

Installation Playwright Test was created specifically to accommodate the needs of end-to-end testing. Playwright supports all modern rendering engines including Chromium, WebKit, and Firefox. Test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation of Google Chrome for Android and Mobile Safari. You will learn How to install Playwright What's Installed How to run the example test How to open the HTML test report Installing Playwright Get started by installing Playwright using npm or yarn. Alternatively you can also get started and run your tests using the VS Code Extension. `npm install playwright` `yarn add playwright` `npm init playwright@latest` `yarn create playwright` `npm dlx create-playwright` Run the install command and select the following to get started: Choose between TypeScript or JavaScript (default is TypeScript) Name of your Tests folder (default is tests or e2e if you already have a tests folder in your project) Add a GitHub Actions workflow to easily run tests on CI Install Playwright browsers (default is true) What's Installed Playwright will download the browsers needed as well as create the following files. `playwright.config.ts` `package.json` `package-lock.json` `tests/example.spec.ts` `tests/examples/demo-todo-app.spec.ts` The `playwright.config` is where you can add configuration for Playwright including modifying which browsers you would like to run Playwright on. If you are running tests inside an already existing project then dependencies will be added directly to your `package.json`. The tests folder contains a basic example test to help you get started with testing. For a more detailed example check out the tests-examples folder which contains tests written to test a todo app. Running the Example Test By default tests will be run on all 3 browsers, chromium, firefox and webkit using 3 workers. This can be configured in the `playwright.config` file. Tests are run in headless mode meaning no browser will open up when running the tests. Results of the tests and test logs will be shown in the terminal. `npx playwright test` See our doc on Running Tests to learn more about running tests in headed mode, running multiple tests, running specific tests etc. Run your tests with UI Mode for a better developer experience with time travel debugging, watch mode and more. `npx playwright test --ui` HTML Test Reports Once your test has finished running a HTML Reporter will have been created which shows you a full report of your tests allowing you to filter the report by browsers, passed tests, failed tests, skipped tests and flaky tests. You can click on each test and explore the test's errors as well as each step of the test. By default, the HTML report is opened automatically if some of the tests failed. `npx playwright show-report` System requirements Node.js 16+ Windows 10+, Windows Server 2016+ or Windows Subsystem for Linux (WSL). MacOS 12 Monterey or MacOS 13 Ventura. Debian 11, Ubuntu 20.04 or Ubuntu 22.04. What's next Write tests using web first assertions, page fixtures and locators Run single test, multiple tests, headed mode Generate tests with Codegen See a trace of your tests

Writing tests Playwright tests are simple, they perform actions, and assert the state against expectations. There is no need to wait for anything prior to performing an action: Playwright automatically waits for the wide range of actionability checks to pass prior to performing each action. There is also no need to deal with the race conditions when performing the checks - Playwright assertions are designed in a way that they describe the expectations that need to be eventually met. That's it! These design choices allow Playwright users to forget about flaky timeouts and racy checks in their tests altogether. You will learn How to write the first test How to perform actions How to use

assertionsHow tests run in isolationHow to use test hooksFirst test Take a look at the following example to see how to write a test.

```

import { test, expect } from '@playwright/test';
test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();
  // Expects the URL to contain intro.
  await expect(page).toHaveURL(/.*intro/);
});

```

noteAdd // @ts-check at the start of each test file when using JavaScript in VS Code to get automatic type checking.

Actions

Navigation

Most of the tests will start with navigating page to the URL. After that, test will be able to interact with the page elements.

```

await page.goto('https://playwright.dev/');

```

Playwright will wait for page to reach the load state prior to moving forward. Learn more about the `page.goto()` options.

Interactions

Performing actions starts with locating the elements. Playwright uses Locators API for that. Locators represent a way to find element(s) on the page at any moment, learn more about the different types of locators available. Playwright will wait for the element to be actionable prior to performing the action, so there is no need to wait for it to become available.

```

// Create a locator
const getStarted = page.getByRole('link', { name: 'Get started' });
// Click it.
await getStarted.click();

```

In most cases, it'll be written in one line:

```

await page.getByRole('link', { name: 'Get started' }).click();

```

Basic actions

This is the list of the most popular Playwright actions. Note that there are many more, so make sure to check the Locator API section to learn more about them.

ActionDescription

- `locator.check()` Check the input checkbox
- `locator.click()` Click the element
- `locator.uncheck()` Uncheck the input checkbox
- `locator.hover()` Hover mouse over the element
- `locator.fill()` Fill the form field (fast)
- `locator.focus()` Focus the element
- `locator.press()` Press single key
- `locator.setInputFiles()` Pick files to upload
- `locator.selectOption()` Select option in the drop down
- `locator.type()` Type text character by character (slow)

Assertions

Playwright includes test assertions in the form of expect function. To make an assertion, call `expect(value)` and choose a matcher that reflects the expectation. There are many generic matchers like `toEqual`, `toContain`, `toBeTruthy` that can be used to assert any conditions.

```

expect(success).toBeTruthy();

```

Playwright also includes async matchers that will wait until the expected condition is met. Using these matchers allows making the tests non-flaky and resilient. For example, this code will wait until the page gets the title containing "Playwright":

```

await expect(page).toHaveTitle(/Playwright/);

```

Here is the list of the most popular async assertions. Note that there are many more to get familiar with:

ActionDescription

- `expect(locator).toBeChecked()` Checkbox is checked
- `expect(locator).toBeEnabled()` Control is enabled
- `expect(locator).toBeVisible()` Element is visible
- `expect(locator).toContainText()` Element contains text
- `expect(locator).toHaveAttribute()` Element has attribute
- `expect(locator).toHaveCount()` List of elements has given length
- `expect(locator).toHaveText()` Element matches text
- `expect(locator).toHaveValue()` Input element has value
- `expect(page).toHaveTitle()` Page has title
- `expect(page).toHaveURL()` Page has URL
- `expect(page).toHaveScreenshot()` Page has screenshot

Test Isolation

Playwright Test is based on the concept of test fixtures such as the built in page fixture, which is

passed into your test. Pages are isolated between tests due to the Browser Context, which is equivalent to a brand new browser profile, where every test gets a fresh environment, even when multiple tests run in a single Browser.

tests/example.spec.ts

```
test('basic test', async ({ page }) => { // ...});
```

Using Test Hooks You can use various test hooks such as `test.describe` to declare a group of tests and `test.beforeEach` and `test.afterEach` which are executed before/after each test. Other hooks include the `test.beforeAll` and `test.afterAll` which are executed once per worker before/after all tests.

tests/example.spec.ts

```
import { test, expect } from '@playwright/test';
test.describe('navigation', () => {
  test.beforeEach(async ({ page }) => { // Go to the starting url before each test.
    await page.goto('https://playwright.dev/');
  });
  test('main navigation', async ({ page }) => { // Assertions use the expect API.
    await expect(page).toHaveURL('https://playwright.dev/');
  });
});
```

What's Next Run single test, multiple tests, headed mode

Generate tests with Codegen See a trace of your tests

Running tests You can run a single test, a set of tests or all tests. Tests can be run on one browser or multiple browsers. By default tests are run in a headless manner meaning no browser window will be opened while running the tests and results will be seen in the terminal. You will learn

How to run tests from the command line

How to debug tests

How to open the HTML test reporter

Run tests in UI Mode Run your tests with UI Mode for a better developer experience with time travel debugging, watch mode and more.

Command Line Running all tests

```
npx playwright test
```

Running a single test

```
npx playwright test landing-page.spec.ts
```

Run a set of test files

```
npx playwright test tests/todo-page/ tests/landing-page/
```

Run files that have landing or login in the file name

```
npx playwright test landing login
```

Run the test with the title

```
npx playwright test -g "add a todo item"
```

Running tests in headed mode

```
npx playwright test landing-page.spec.ts --headed
```

Running tests on a specific project

```
npx playwright test landing-page.ts --project=chromium
```

Debugging Tests Since Playwright runs in Node.js, you can debug it with your debugger of choice e.g. using `console.log` or inside your IDE or directly in VS Code with the VS Code Extension. Playwright comes with the Playwright Inspector which allows you to step through Playwright API calls, see their debug logs and explore locators.

Debugging all tests

```
npx playwright test --debug
```

Debugging one test file

```
npx playwright test example.spec.ts --debug
```

Debugging a test from the line number where the test(.. is defined

```
npx playwright test example.spec.ts:10 --debug
```

Check out our debugging guide to learn more about the Playwright Inspector as well as debugging with Browser Developer tools.

Test Reports The HTML Reporter shows you a full report of your tests allowing you to filter the report by browsers, passed tests, failed tests, skipped tests and flaky tests. By default, the HTML report is opened automatically if some of the tests failed.

```
npx playwright show-report
```

You can click on each test and explore the tests errors as well as each step of the test.

What's Next Generate tests with Codegen

See a trace of your tests

Test generator Playwright comes with the ability to generate tests out of the box and is a great way to quickly get started with testing. It will open two windows, a browser window where you interact with the website you wish to test and the Playwright Inspector window where you can record your tests, copy the tests, clear your tests as well as change the language of your tests. You will learn

How to record a test

How to generate locators Your browser does not support the video tag.

Running Codegen Use the `codegen` command to run the test generator followed by the URL of the website you

want to generate tests for. The URL is optional and you can always run the command without it and then add the URL directly into the browser window instead.`npx playwright codegen demo.playwright.dev/todomvc`

Recording a test

Run codegen and perform actions in the browser. Playwright will generate the code for the user interactions. Codegen will look at the rendered page and figure out the recommended locator, prioritizing role, text and test id locators. If the generator identifies multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, therefore eliminating and reducing test(s) failing and flaking due to locators.

Generating locators

You can generate locators with the test generator. Press the 'Record' button to stop the recording and the 'Pick Locator' button will appear. Click on the 'Pick Locator' button and then hover over elements in the browser window to see the locator highlighted underneath each element. To choose a locator click on the element you would like to locate and the code for that locator will appear in the field next to the Pick Locator button. You can then edit the locator in this field to fine tune it or use the copy button to copy it and paste it into your code.

Emulation

You can also generate tests using emulation so as to generate a test for a specific viewport, device, color scheme, as well as emulate the geolocation, language or timezone. The test generator can also generate a test while preserving authenticated state. Check out the Test Generator guide to learn more.

What's Next

See a trace of your tests

Trace viewer

Playwright Trace Viewer is a GUI tool that lets you explore recorded Playwright traces of your tests meaning you can go back and forward through each action of your test and visually see what was happening during each action. You will learn

How to record a trace

How to open the HTML report

How to open and view the trace

Recording a Trace

By default the `playwright.config` file will contain the configuration needed to create a trace.zip file for each test. Traces are setup to run on-first-retry meaning they will be run on the first retry of a failed test. Also retries are set to 2 when running on CI and 0 locally. This means the traces will be recorded on the first retry of a failed test but not on the first run and not on the second retry.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({  retries: process.env.CI ? 2 : 0, // set to 2 when running on CI  // ...  use: {    trace: 'on-first-retry', // record traces on first retry of each test  },});
```

To learn more about available options to record a trace check out our detailed guide on Trace Viewer.

Traces are normally run in a Continuous Integration(CI) environment as locally you can use debugging methods to debug tests. However should you want to run traces locally you can force tracing to be on with `--trace on`. ``` npx playwright test --trace on ``` **note** The trace-on flag was introduced in Playwright v1.25. Check your package.json to make sure you have at least this version of Playwright installed. Opening the HTML Report The HTML report shows you a report of all your tests that have been ran and on which browsers as well as how long they took. Tests can be filtered by passed tests, failed, flakey or skipped tests. You can also search for a particular test. Clicking on a test will open the detailed view where you can see more information on your tests such as the errors, the test steps and the trace. ``` npx playwright show-report ``` In the HTML report click on the trace icon next to the test name file name to directly open the trace for the required test. You can also click open the detailed view of the test and scroll down to the 'Traces' tab and open the trace by clicking on the trace screenshot. To learn more about reporters check out our detailed guide on reporters including the HTML

Reporter.Viewing the Trace View traces of your test by clicking through each action or hovering using the timeline and see the state of the page before and after the action. Inspect the log, source and network during each step of the test. The trace viewer creates a DOM snapshot so you can fully interact with it, open devtools etc.To learn more about traces check out our detailed guide on Trace Viewer.

UI ModeUI Mode let's you explore, run and debug tests with a time travel experience complete with watch mode. All test files are loaded into the testing sidebar where you can expand each file and describe block to individually run, view, watch and debug each test. Filter tests by text or @tag or by passed, failed and skipped tests as well as by projects as set in your playwright.config file. See a full trace of your tests and hover back and forward over each action to see what was happening during each step and pop out the DOM snapshot to a separate window for a better debugging experience.

Running tests in UI Mode To open UI mode, run the following command:`npx playwright test --ui`Filtering tests Filter tests by text or @tag or by passed, failed or skipped tests. You can also filter by projects as set in your playwright.config file. If you are using project dependencies make sure to run your setup tests first before running the tests that depend on them. The UI mode will not take into consideration the setup tests and therefore you will have to manually run them first.

Running your tests Once you launch UI Mode you will see a list of all your test files. You can run all your tests by clicking the triangle icon in the sidebar. You can also run a single test file, a block of tests or a single test by hovering over the name and clicking on the triangle next to it.

Viewing test traces Traces are shown for each test that has been run, so to see the trace, click on one of the test names. Note that you won't see any trace results if you click on the name of the test file or the name of a describe block.

Actions and metadata In the Actions tab you can see what locator was used for every action and how long each one took to run. Hover over each action of your test and visually see the change in the DOM snapshot. Go back and forward in time and click an action to inspect and debug. Use the Before and After tabs to visually see what happened before and after the action. Next to the Actions tab you will find the Metadata tab which will show you more information on your test such as the Browser, viewport size, test duration and more.

Source, console, log and network As you hover over each action of your test the source code for the test is highlighted below. Click on the source tab to see the source code for the entire test. Click on the console tab to see the console logs for each action. Click on the log tab to see the logs for each action. Click on the network tab to see the network logs for each action.

Attachments The "Attachments" tab allows you to explore attachments. If you're doing visual regression testing, you'll be able to compare screenshots by examining the image diff, the actual image and the expected image. When you click on the expected image you can use the slider to slide one image over the other so you can easily see the differences in your screenshots.

Pop out and inspect the DOM Pop out the DOM snapshot into it's own window for a better debugging experience by clicking on the pop out icon above the DOM snapshot. From there you can open the browser DevTools and inspect the HTML, CSS, Console etc. Go back to UI Mode and click on another action and pop that one out to easily compare the two side by side or debug each individually.

Timeline view At the top of the trace you can see a timeline view of each action of your test. Hover back and forth to see an image snapshot for each action.

Pick locator Click on the pick locator button and hover over the

DOM snapshot to see the locator for each element highlighted as you hover. Click on an element to save the locator into the pick locator field. You can then copy the locator and paste it into your test.

Watch mode Next to the name of each test in the sidebar you will find an eye icon. Clicking on the icon will activate watch mode which will re-run the test when you make changes to it. You can watch a number of tests at the same time by clicking the eye icon next to each one or all tests by clicking the eye icon at the top of the sidebar. If you are using VS Code then you can easily open your test by clicking on the file icon next to the eye icon. This will open your test in VS Code right at the line of code that you clicked on.

Docker & GitHub Codespaces For Docker and GitHub Codespaces environments, you can run UI mode in the browser. In order for an endpoint to be accessible outside of the container, it needs to be bound to the 0.0.0.0 interface: `npx playwright test --ui-host=0.0.0.0` In the case of GitHub Codespaces, the port gets forwarded automatically, so you can open UI mode in the browser by clicking on the link in the terminal. To have a static port, you can pass the `--ui-port` flag: `npx playwright test --ui-port=8080 --ui-host=0.0.0.0`

note Be aware that when specifying the `--ui-host=0.0.0.0` flag, UI Mode with your traces, the passwords and secrets is accessible from other machines inside your network. In the case of GitHub Codespaces, the ports are only accessible from your account by default.

CI GitHub Actions When installing Playwright you are given the option to add a GitHub Actions. This creates a `playwright.yml` file inside a `.github/workflows` folder containing everything you need so that your tests run on each push and pull request into the main/master branch.

What you will learn: GitHub Actions Create a Repo and Push to GitHub Opening the Workflows Viewing Test Logs HTML Report Downloading the HTML Report Viewing the HTML Report Viewing the Trace What's Next

GitHub Actions Tests will run on push or pull request on branches main/master. The workflow will install all dependencies, install Playwright and then run the tests. It will also create the HTML report.

```
name: Playwright Test
on: push:
  branches: [main, master]
pull_request:
  branches: [main, master]
jobs:
  test:
    timeout-minutes: 60
    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 Browsers
        run: npx playwright install --with-deps
      - name: Run Playwright tests
        run: npx playwright test
      - uses: actions/upload-artifact@v3
        if: always()
        with:
          name: playwright-report
          path: playwright-report/
          retention-days: 30
```

Create a Repo and Push to GitHub Create a repo on GitHub and create a new repository or push an existing repository. Follow the instructions on GitHub and don't forget to initialize a git repository using the `git init` command so you can add, commit and push your code.

Opening the Workflows Click on the Actions tab to see the workflows. Here you will see if your tests have passed or failed.

On Pull Requests you can also click on the Details link in the PR status check.

Viewing Test Logs Clicking on the workflow run will show you all the actions that GitHub performed and clicking on Run Playwright tests will show the error messages, what was expected and what was received as well as the call log.

HTML Report The HTML Report shows you a full report of your tests. You can filter the report by browsers, passed tests, failed tests, skipped tests and flaky tests.

Downloading the HTML Report In the Artifacts section click on the `playwright-report` to download your report in the format of a zip file.

Viewing the HTML Report Locally opening the report will not work as expected as

you need a web server in order for everything to work correctly. First, extract the zip, preferably in a folder that already has Playwright installed. Using the command line change into the directory where the report is and use `npx playwright show-report` followed by the name of the extracted folder. This will serve up the report and enable you to view it in your browser. `npx playwright show-report name-of-my-extracted-playwright-report` To learn more about reports check out our detailed guide on [HTML ReporterViewing the Trace](#) Once you have served the report using `npx playwright show-report`, click on the trace icon next to the test's file name as seen in the image above. You can then view the trace of your tests and inspect each action to try to find out why the tests are failing. To learn more about traces check out our detailed guide on [Trace Viewer](#). To learn more about running tests on CI check out our detailed guide on [Continuous Integration](#). What's Next Learn how to use [Locators](#) Learn how to perform [Actions](#) Learn how to write [Assertions](#)

Getting started - VS Code Playwright Test was created specifically to accommodate the needs of end-to-end testing. Playwright supports all modern rendering engines including Chromium, WebKit, and Firefox. Test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation of Google Chrome for Android and Mobile Safari. Get started by installing Playwright and generating a test to see it in action. Alternatively you can also get started and run your tests using the CLI. **Installation** Install the VS Code extension from the marketplace or from the extensions tab in VS Code. Once installed, open the command panel and type: `Install Playwright` **Select Test:** Install Playwright and Choose the browsers you would like to run your tests on. These can be later configured in the `playwright.config` file. You can also choose if you would like to have a GitHub Actions setup to run your tests on CI. **Running Tests** You can run a single test by clicking the green triangle next to your test block to run your test. Playwright will run through each line of the test and when it finishes you will see a green tick next to your test block as well as the time it took to run the test. **Run Tests and Show Browsers** You can also run your tests and show the browsers by selecting the option `Show Browsers` in the testing sidebar. Then when you click the green triangle to run your test the browser will open and you will visually see it run through your test. Leave this selected if you want browsers open for all your tests or uncheck it if you prefer your tests to run in headless mode with no browser open. Use the `Close all browsers` button to close all browsers. **View and Run All Tests** View all tests in the testing sidebar and extend the tests by clicking on each test. Tests that have not been run will not have the green check next to them. Run all tests by clicking on the white triangle as you hover over the tests in the testing sidebar. **Run Tests on Specific Browsers** The VS Code test runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown and choose another profile or modify the default profile by clicking `Select Default Profile` and select the browsers you wish to run your tests on. Choose various or all profiles to run tests on multiple profiles. These profiles are read from the `playwright.config` file. To add more profiles such as a mobile profile, first add it to your config file and it will then be available here. **Debugging Tests** With the VS Code extension you can debug your tests right in VS Code see error messages, create breakpoints and live debug your tests. **Error Messages** If your test fails VS Code will show you error messages right in the editor showing what was expected, what was received as well as a complete call

log.Live Debugging You can debug your test live in VS Code. After running a test with the Show Browser option checked, click on any of the locators in VS Code and it will be highlighted in the Browser window. Playwright will highlight it if it exists and show you if there is more than one result. You can also edit the locators in VS Code and Playwright will show you the changes live in the browser window.

Run in Debug Mode To set a breakpoint click next to the line number where you want the breakpoint to be until a red dot appears. Run the tests in debug mode by right clicking on the line next to the test you want to run. A browser window will open and the test will run and pause at where the breakpoint is set. You can step through the tests, pause the test and rerun the tests from the menu in VS Code.

Debug in different Browsers By default debugging is done using the Chromium profile. You can debug your tests on different browsers by right clicking on the debug icon in the testing sidebar and clicking on the 'Select Default Profile' option from the dropdown. Then choose the test profile you would like to use for debugging your tests. Each time you run your test in debug mode it will use the profile you selected. You can run tests in debug mode by right clicking the line number where your test is and selecting 'Debug Test' from the menu. To learn more about debugging, see [Debugging in Visual Studio Code](#).

Generating Tests CodeGen will auto generate your tests for you as you perform actions in the browser and is a great way to quickly get started. The viewport for the browser window is set to a specific width and height. See the [configuration guide](#) to change the viewport or emulate different environments.

Record a New Test To record a test click on the Record new button from the Testing sidebar. This will create a test-1.spec.ts file as well as open up a browser window. In the browser go to the URL you wish to test and start clicking around. Playwright will record your actions and generate a test for you. Once you are done recording click the cancel button or close the browser window. You can then inspect your test-1.spec.ts file and see your generated test. Your browser does not support the video tag.

Record at Cursor To record from a specific point in your test file click the Record at cursor button from the Testing sidebar. This generates actions into the existing test at the current cursor position. You can run the test, position the cursor at the end of the test and continue generating the test.

Picking a Locator Pick a locator and copy it into your test file by clicking the Pick locator button from the testing sidebar. Then in the browser click the element you require and it will now show up in the Pick locator box in VS Code. Press 'enter' on your keyboard to copy the locator into the clipboard and then paste anywhere in your code. Or press 'escape' if you want to cancel. Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If the generator finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.

What's next Write tests using web first assertions, page fixtures and locators. See test reports. See a trace of your tests.

Release notes Version 1.36 Summer maintenance release.

Browser Versions Chromium 115.0.5790.75 Mozilla Firefox 115.0 WebKit 17.0 This version was also tested against the following stable channels: Google Chrome 114 Microsoft Edge 114 Version 1.35

Highlights UI mode is now available in VS Code Playwright extension via a new "Show trace viewer" button: UI mode and trace viewer mark network requests handled with `page.route()` and `browserContext.route()` handlers, as well as those issued via the API

testing: New option `maskColor` for methods `page.screenshot()`, `locator.screenshot()`,

`expect(page).toHaveScreenshot()` and `expect(locator).toHaveScreenshot()` to change default masking color:
`await page.goto('https://playwright.dev');`
`await expect(page).toHaveScreenshot({ mask: [page.locator('img')], maskColor: '#00FF00', // green});`
New uninstall CLI command to uninstall browser binaries:
`$ npx playwright uninstall # remove browsers installed by this installation`
`$ npx playwright uninstall --all # remove all ever-install Playwright browsers`
Both UI mode and trace viewer now could be opened in a browser tab:
`$ npx playwright test --ui-port 0 # open UI mode in a tab on a random port`
`$ npx playwright show-trace --port 0 # open trace viewer in tab on a random port`

Breaking changes `playwright-core` binary got renamed from `playwright` to `playwright-core`. So if you use `playwright-core` CLI, make sure to update the name:
`$ npx playwright-core install # the new way to install browsers when using playwright-core`
 This change does not affect `@playwright/test` and `playwright` package users.

Browser Versions Chromium 115.0.5790.13 Mozilla Firefox 113.0 WebKit 16.4
 This version was also tested against the following stable channels: Google Chrome 114 Microsoft Edge 114 Version 1.34

Highlights UI Mode now shows steps, fixtures and attachments: New property `testProject.teardown` to specify a project that needs to run after this and all dependent projects have finished. Teardown is useful to cleanup any resources acquired by this project. A common pattern would be a setup dependency with a corresponding teardown:
`playwright.config.ts`
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ projects: [{ name: 'setup', testMatch: /global.setup\.ts/, teardown: 'teardown', }, { name: 'teardown', testMatch: /global.teardown\.ts/, }, { name: 'chromium', use: devices['Desktop Chrome'], dependencies: ['setup'], }, { name: 'firefox', use: devices['Desktop Firefox'], dependencies: ['setup'], }, { name: 'webkit', use: devices['Desktop Safari'], dependencies: ['setup'], },], });`
New method `expect.configure` to create pre-configured `expect` instance with its own defaults such as timeout and soft.
`const slowExpect = expect.configure({ timeout: 10000 });`
`await slowExpect(locator).toHaveText('Submit');`
Always do soft assertions.
`const softExpect = expect.configure({ soft: true });`
New options `stderr` and `stdout` in `testConfig.webServer` to configure output handling:
`playwright.config.ts`
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ // Run your local dev server before starting the tests webServer: { command: 'npm run start', url: 'http://127.0.0.1:3000', reuseExistingServer: !process.env.CI, stdout: 'pipe', stderr: 'pipe', }, });`
New locator `locator.and()` to create a locator that matches both locators.
`const button = page.getByRole('button').and(page.getByTitle('Subscribe'));`
New events `browserContext.on('console')` and `browserContext.on('dialog')` to subscribe to any dialogs and console messages from any page from the given browser context. Use the new methods `consoleMessage.page()` and `dialog.page()` to pin-point event source.

Breaking changes `npx playwright test` no longer works if you install both `playwright` and `@playwright/test`. There's no need to install both, since you can always import browser automation APIs from `@playwright/test` directly:
`automation.ts`
`import { chromium, firefox, webkit } from '@playwright/test';`
 /* ... */
Node.js 14 is no longer supported since it reached its end-of-life on April 30, 2023.

Browser Versions Chromium 114.0.5735.26 Mozilla Firefox 113.0 WebKit 16.4
 This version was also tested against the following stable channels: Google Chrome 113 Microsoft Edge 113 Version 1.33

Locators Update Use `locator.or()` to create a locator that matches either of the two locators.

Consider a scenario where you'd like to click on a "New email" button, but sometimes a security settings dialog shows up instead. In this case, you can wait for either a "New email" button, or a dialog and act accordingly:

```
const newEmail =
page.getByRole('button', { name: 'New email' });const dialog = page.getByText('Confirm
security settings');await expect(newEmail.or(dialog)).toBeVisible();if (await
dialog.isVisible()) await page.getByRole('button', { name: 'Dismiss' }).click();await
newEmail.click();Use new options hasNot and hasNotText in locator.filter() to find
elements that do not match certain conditions.const rowLocator =
page.locator('tr');await rowLocator .filter({ hasNotText: 'text in column
1' }) .filter({ hasNot: page.getByRole('button', { name: 'column 2
button' }) }) .screenshot();Use new web-first assertion expect(locator).toBeAttached()
to ensure that the element is present in the page's DOM. Do not confuse with the
expect(locator).toBeVisible() that ensures that element is both attached & visible.New
APIs locator.or()New option hasNot in locator.filter()New option hasNotText in
locator.filter()expect(locator).toBeAttached()New option timeout in
route.fetch()reporter.onExit()& p Breaking change The mcr.microsoft.com/
playwright:v1.33.0 now serves a Playwright image based on Ubuntu Jammy. To use the
focal-based image, please use mcr.microsoft.com/playwright:v1.33.0-focal
instead.Browser Versions Chromium 113.0.5672.53Mozilla Firefox 112.0WebKit
16.4This version was also tested against the following stable channels:Google Chrome
112Microsoft Edge 112Version 1.32 Introducing UI Mode (preview) New UI Mode lets
you explore, run and debug tests. Comes with a built-in watch mode.Engage with a new
flag --ui:npx playwright test --uiNew APIs New options updateMode and updateContent
in page.routeFromHAR() and browserContext.routeFromHAR().Chaining existing
locator objects, see locator docs for details.New property testInfo.testId.New option
name in method tracing.startChunk().& p Breaking change in component tests Note:
component tests only, does not affect end-to-end tests.@playwright/experimental-ct-
react now supports React 18 only.If you're running component tests with React 16 or
17, please replace @playwright/experimental-ct-react with @playwright/experimental-ct-
react17.Browser Versions Chromium 112.0.5615.29Mozilla Firefox 111.0WebKit
16.4This version was also tested against the following stable channels:Google Chrome
111Microsoft Edge 111Version 1.31 New APIs New property testProject.dependencies
to configure dependencies between projects.Using dependencies allows global setup to
produce traces and other artifacts, see the setup steps in the test report and
more.playwright.config.tsimport { defineConfig } from '@playwright/test';export default
defineConfig({ projects: [ { name: 'setup', testMatch: /global.setup\.ts/, },
{ name: 'chromium', use: devices['Desktop Chrome'], dependencies:
['setup'], }, { name: 'firefox', use: devices['Desktop Firefox'], dependencies:
['setup'], }, { name: 'webkit', use: devices['Desktop Safari'], dependencies:
['setup'], }, ],});New assertion expect(locator).toBeInViewport() ensures that locator
points to an element that intersects viewport, according to the intersection observer
API.const button = page.getByRole('button');// Make sure at least some part of element
intersects viewport.await expect(button).toBeInViewport();// Make sure element is fully
outside of viewport.await expect(button).not.toBeInViewport();// Make sure that at least
half of the element intersects viewport.await expect(button).toBeInViewport({ ratio:
0.5 });Miscellaneous DOM snapshots in trace viewer can be now opened in a separate
```

window.New method `defineConfig` to be used in `playwright.config`. New option `Route.fetch.maxRedirects` for method `route.fetch()`. Playwright now supports Debian 11 arm64. Official docker images now include Node 18 instead of Node 16.

Breaking change in component tests Note: component tests only, does not affect end-to-end tests. `playwright-ct.config` configuration file for component testing now requires calling `defineConfig`.

```

// Before
import { type PlaywrightTestConfig, devices } from '@playwright/experimental-ct-react';
const config: PlaywrightTestConfig = { // ... config goes here ... };
export default config;

// After
import { defineConfig, devices } from '@playwright/experimental-ct-react';
export default defineConfig({ // ... config goes here ... });
  
```

Browser Versions Chromium 111.0.5563.19 Mozilla Firefox 109.0 WebKit 16.4 This version was also tested against the following stable channels: Google Chrome 110 Microsoft Edge 110 Version 1.30 Browser Versions Chromium 110.0.5481.38 Mozilla Firefox 108.0.2 WebKit 16.4 This version was also tested against the following stable channels: Google Chrome 109 Microsoft Edge 109 Version 1.29

New APIs New method `route.fetch()` and new option `json` for `route.fulfill()`:


```

await page.route('**/api/settings', async route => { // Fetch original settings.
  const response = await route.fetch(); // Force settings theme to a predefined value.
  const json = await response.json(); json.theme = 'Solorized'; // Fulfill with modified data.
  await route.fulfill({ json });
});
  
```

New method `locator.all()` to iterate over all matching elements:


```

// Check all checkboxes!
const checkboxes = page.getByRole('checkbox');
for (const checkbox of await checkboxes.all()) await checkbox.check();
  
```

`locator.selectOption()` matches now by value or label:


```

<select multiple>
  <option value="red">Red</div>
  <option value="green">Green</div>
  <option value="blue">Blue</div>
</select>
await element.selectOption('Red');
  
```

Retry blocks of code until all assertions pass:


```

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

Read more in our documentation. Automatically capture full page screenshot on test failure:


```

import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    screenshot: { mode: 'only-on-failure', fullPage: true, },
  },
});
  
```

Miscellaneous Playwright Test now respects `jsconfig.json`. New options `args` and `proxy` for `androidDevice.launchBrowser()`. Option `postData` in method `route.continue()` now supports Serializable values.

Browser Versions Chromium 109.0.5414.46 Mozilla Firefox 107.0 WebKit 16.4 This version was also tested against the following stable channels: Google Chrome 108 Microsoft Edge 108 Version 1.28

Playwright Tools Record at Cursor in VSCode. You can run the test, position the cursor at the end of the test and continue generating the test. Live Locators in VSCode. You can hover and edit locators in VSCode to get them highlighted in the opened browser. Live Locators in CodeGen. Generate a locator for any element on the page using "Explore" tool. Codegen and Trace Viewer Dark Theme. Automatically picked up from operating system settings.

Test Runner Configure retries and test timeout for a file or a test with `test.describe.configure()`. Each test in the file will be retried twice and have a timeout of 20 seconds.


```

test.describe.configure({ retries: 2, timeout: 20_000 });
test('runs first', async ({ page }) => {});
test('runs second', async ({ page }) => {});
  
```

Use `testProject.snapshotPathTemplate` and `testConfig.snapshotPathTemplate` to configure a template controlling location of snapshots generated by `expect(page).toHaveScreenshot()` and

`expect(snapshot).toMatchSnapshot().playwright.config.ts` import { defineConfig } from '@playwright/test'; export default defineConfig({ testDir: './tests', snapshotPathTemplate: '{testDir}/__screenshots__/{testFilePath}/{arg}{ext}', });

New APIs
`locator.blur()`, `locator.clear()`, `android.launchServer()` and `android.connect()`, `androidDevice.on('close')`

Browser Versions
 Chromium 108.0.5359.29, Mozilla Firefox 106.0, WebKit 16.4. This version was also tested against the following stable channels: Google Chrome 107, Microsoft Edge 107, Version 1.27

Locators
 With these new APIs writing locators is a joy: `page.getByText()` to locate by text content, `page.getByRole()` to locate by ARIA role, ARIA attributes and accessible name, `page.getByLabel()` to locate a form control by associated label's text, `page.getByTestId()` to locate an element based on its data-testid attribute (other attribute can be configured), `page.getByPlaceholder()` to locate an input by placeholder, `page.getByAltText()` to locate an element, usually image, by its text alternative, `page.getByTitle()` to locate an element by its title.

await
`page.getByLabel('User Name').fill('John');`, `await page.getByLabel('Password').fill('secret-password');`, `await page.getByRole('button', { name: 'Sign in' }).click();`, `await expect(page.getByText('Welcome, John!')).toBeVisible();`

All the same methods are also available on `Locator`, `FrameLocator` and `Frame` classes.

Other highlights
workers option
 in the `playwright.config.ts` now accepts a percentage string to use some of the available CPUs. You can also pass it in the command line: `npx playwright test --workers=20%`

New options
`host` and `port` for the html reporter.

`import { defineConfig } from '@playwright/test'; export default defineConfig({ reporter: [['html'], { host: 'localhost', port: '9223' }], });`

New field
`FullConfig.configFile` is available to test reporters, specifying the path to the config file if any.

As announced in v1.25, Ubuntu 18 will not be supported as of Dec 2022. In addition to that, there will be no WebKit updates on Ubuntu 18 starting from the next Playwright release.

Behavior Changes
`expect(locator).toHaveAttribute()` with an empty value does not match missing attribute anymore. For example, the following snippet will succeed when button does not have a disabled attribute.

`await expect(page.getByRole('button')).toHaveAttribute('disabled', '');`

Command line options
`--grep` and `--grep-invert` previously incorrectly ignored `grep` and `grepInvert` options specified in the config. Now all of them are applied together.

Browser Versions
 Chromium 107.0.5304.18, Mozilla Firefox 105.0.1, WebKit 16.0. This version was also tested against the following stable channels: Google Chrome 106, Microsoft Edge 106, Version 1.26

Assertions
 New option enabled for `expect(locator).toBeEnabled()`.

`expect(locator).toHaveText()` now pierces open shadow roots.

New option `editable` for `expect(locator).toBeEditable()`.

New option `visible` for `expect(locator).toBeVisible()`.

Other highlights
 New option `maxRedirects` for `apiRequestContext.get()` and others to limit redirect count.

New command-line flag
`--pass-with-no-tests` that allows the test suite to pass when no files are found.

New command-line flag
`--ignore-snapshots` to skip snapshot expectations, such as `expect(value).toMatchSnapshot()` and `expect(page).toHaveScreenshot()`.

Behavior Change
 A bunch of Playwright APIs already support the `waitUntil: 'domcontentloaded'` option. For example: `await page.goto('https://playwright.dev', { waitUntil: 'domcontentloaded' });`

Prior to 1.26, this would wait for all iframes to fire the `DOMContentLoaded` event. To align with web specification, the `'domcontentloaded'` value only waits for the target frame to fire the `'DOMContentLoaded'` event. Use

waitUntil: 'load' to wait for all iframes. Browser Versions Chromium 106.0.5249.30 Mozilla Firefox 104.0 WebKit 16.0 This version was also tested against the following stable channels: Google Chrome 105 Microsoft Edge 105 Version 1.25 VSCode Extension Watch your tests running live & keep devtools open. Pick selector. Record new test from current page state. Test Runner test.step() now returns the value of the step function: test('should work', async ({ page }) => { const pageTitle = await test.step('get title', async () => { await page.goto('https://playwright.dev'); return await page.title(); }); console.log(pageTitle); }); Added test.describe.fixme(). New 'interrupted' test status. Enable tracing via CLI flag: npx playwright test --trace=on. Announcements

- We now ship Ubuntu 22.04 Jammy Jellyfish docker image: mcr.microsoft.com/playwright:v1.34.0-jammy.
- This is the last release with macOS 10.15 support (deprecated as of 1.21).
- This is the last release with Node.js 12 support, we recommend upgrading to Node.js LTS (16).
- Ubuntu 18 is now deprecated and will not be supported as of Dec 2022.

Browser Versions Chromium 105.0.5195.19 Mozilla Firefox 103.0 WebKit 16.0 This version was also tested against the following stable channels: Google Chrome 104 Microsoft Edge 104 Version 1.24

Multiple Web Servers in playwright.config.ts

Launch multiple web servers, databases, or other processes by passing an array of configurations:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  webServer: [
    {
      command: 'npm run start',
      url: 'http://127.0.0.1:3000',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
    {
      command: 'npm run backend',
      url: 'http://127.0.0.1:3333',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
  ],
  use: {
    baseURL: 'http://localhost:3000/',
  },
});
```

Debian 11 Bullseye Support

Playwright now supports Debian 11 Bullseye on x86_64 for Chromium, Firefox and WebKit. Let us know if you encounter any issues! Linux support looks like this: Ubuntu 20.04 Ubuntu 22.04 Debian 11 Chromium WebKit Firefox

Anonymous Describe

It is now possible to call test.describe() to create suites without a title. This is useful for giving a group of tests a common option with test.use().

```
test.describe(() => {
  test.use({ colorScheme: 'dark' });
  test('one', async ({ page }) => { // ... });
  test('two', async ({ page }) => { // ... });
});
```

Component Tests Update

Playwright 1.24 Component Tests introduce beforeMount and afterMount hooks. Use these to configure your app for tests. For example, this could be used to setup App router in Vue.js:

```
src/component.spec.ts
import { test } from '@playwright/experimental-ct-vue';
import { Component } from './mycomponent';
test('should work', async ({ mount }) => {
  const component = await mount(Component, {
    hooksConfig: { /* anything to configure your app */ },
  });
  playwright/index.ts
import { router } from './router';
import { beforeMount } from '@playwright/experimental-ct-vue/hooks';
beforeMount(async ({ app, hooksConfig }) => {
  app.use(router);
});
```

A similar configuration in Next.js would look like this:

```
src/component.spec.jsx
import { test } from '@playwright/experimental-ct-react';
import { Component } from './mycomponent';
test('should work', async ({ mount }) => {
  const component = await mount(
    <Component>,
    {
      // Pass mock value from test into `beforeMount`.
      hooksConfig: {
        router: {
          query: { page: 1, per_page: 10 },
          asPath: '/posts'
        }
      },
    },
  );
  playwright/index.js
import router from 'next/router';
import { beforeMount } from '@playwright/experimental-ct-react/hooks';
beforeMount(async ({ hooksConfig }) => {
  // Before mount, redefine useRouter to return mock value from test.
  router.useRouter = () => hooksConfig.router;
});
```

Version

1.23 Network Replay Now you can record network traffic into a HAR file and re-use this traffic in your tests. To record network into HAR file: `npx playwright open --save-har=github.har.zip https://github.com/microsoft` Alternatively, you can record HAR programmatically: `const context = await browser.newContext({ recordHar: { path: 'github.har.zip' } });` // ... do stuff ... `await context.close()`; Use the new methods `page.routeFromHAR()` or `browserContext.routeFromHAR()` to serve matching responses from the HAR file: `await context.routeFromHAR('github.har.zip')`; Read more in our documentation.

Advanced Routing You can now use `route.fallback()` to defer routing to other handlers. Consider the following example: // Remove a header from all requests. `test.beforeEach(async ({ page }) => { await page.route('**/*', async route => { const headers = await route.request().allHeaders(); delete headers['if-none-match']; route.fallback({ headers }); });});` `test('should work', async ({ page }) => { await page.route('**/*', route => { if (route.request().resourceType() === 'image') route.abort(); else route.fallback(); });});` Note that the new methods `page.routeFromHAR()` and `browserContext.routeFromHAR()` also participate in routing and could be deferred to.

Web-First Assertions Update New method `expect(locator).toHaveValues()` that asserts all selected values of `<select multiple>` element. Methods `expect(locator).toContainText()` and `expect(locator).toHaveText()` now accept `ignoreCase` option.

Component Tests Update Support for Vue2 via the `@playwright/experimental-ct-vue2` package. Support for component tests for create-react-app with components in .js files. Read more about component testing with Playwright.

Miscellaneous If there's a service worker that's in your way, you can now easily disable it with a new context option `serviceWorkers: 'block'`; `playwright.config.ts` export default `{ use: { serviceWorkers: 'block', } }`; Using .zip path for `recordHar` context option automatically zips the resulting HAR: `const context = await browser.newContext({ recordHar: { path: 'github.har.zip', } });` If you intend to edit HAR by hand, consider using the "minimal" HAR recording mode that only records information that is essential for replaying: `const context = await browser.newContext({ recordHar: { path: 'github.har', mode: 'minimal', } });`

Playwright now runs on Ubuntu 22 amd64 and Ubuntu 22 arm64. We also publish new docker image `mcr.microsoft.com/playwright:v1.34.0-jammy`.

Breaking Changes & WebServer is now considered "ready" if request to the specified url has any of the following HTTP status codes: 200-299, 300-399 (new), 400, 401, 402, 403 (new).

Version 1.22 Highlights Components Testing (preview) Playwright Test can now test your React, Vue.js or Svelte components. You can use all the features of Playwright Test (such as parallelization, emulation & debugging) while running components in real browsers. Here is what a typical component test looks like: `App.spec.tsx` `import { test, expect } from '@playwright/experimental-ct-react'; import App from './App';` // Let's test component in a dark scheme! `test.use({ colorScheme: 'dark' });` `test('should render', async ({ mount }) => { const component = await mount(<App></App>);` // As with any Playwright test, assert locator text. `await expect(component).toContainText('React');` // Or do a screenshot `await expect(component).toHaveScreenshot();` // Or use any Playwright method `await component.click();` Read more in our documentation.

Role selectors that allow selecting elements by their ARIA role, ARIA attributes and accessible name. // Click a button with accessible name "log in" `await page.locator('role=button[name="log in"]').click();` Read more in our documentation.

New

locator.filter() API to filter an existing locator

```
const buttons = page.locator('role=button');// ...const submitButton = buttons.filter({ hasText: 'Submit' });await submitButton.click();
```

New web-first assertions `expect(page).toHaveScreenshot()` and `expect(locator).toHaveScreenshot()` that wait for screenshot stabilization and enhances test reliability. The new assertions has screenshot-specific defaults, such as: disables animations, uses CSS scale option, `await page.goto('https://playwright.dev');`, `await expect(page).toHaveScreenshot();`

The new `expect(page).toHaveScreenshot()` saves screenshots at the same location as `expect(snapshot).toMatchSnapshot()`.

Version 1.21 Highlights

New role selectors that allow selecting elements by their ARIA role, ARIA attributes and accessible name.

Click a button with accessible name "log in"

```
await page.locator('role=button[name="log in"]').click();
```

Read more in our documentation.

New scale option in `page.screenshot()` for smaller sized screenshots.

New caret option in `page.screenshot()` to control text caret. Defaults to "hide".

New method `expect.poll` to wait for an arbitrary condition.

```
// Poll the method until it returns an expected result.await expect.poll(async () => { const response = await page.request.get('https://api.example.com'); return response.status();}).toBe(200);
```

`expect.poll` supports most synchronous matchers, like `.toBe()`, `.toContain()`, etc. Read more in our documentation.

Behavior Changes

ESM support when running TypeScript tests is now enabled by default. The `PLAYWRIGHT_EXPERIMENTAL_TS_ESM` env variable is no longer required.

The `mcr.microsoft.com/playwright` docker image no longer contains Python. Please use `mcr.microsoft.com/playwright/python` as a Playwright-ready docker image with pre-installed Python.

Playwright now supports large file uploads (100s of MBs) via `locator.setInputFiles()` API.

Browser Versions

Chromium 101.0.4951.26
Mozilla Firefox 98.0.2
WebKit 15.4

This version was also tested against the following stable channels: Google Chrome 100, Microsoft Edge 100

Version 1.20 Highlights

New options for methods `page.screenshot()`, `locator.screenshot()` and `elementHandle.screenshot()`:

- Option `animations`: "disabled" rewinds all CSS animations and transitions to a consistent state
- Option `mask`: `Locator[]` masks given elements, overlaying them with pink #FF00FF boxes.
- `expect().toMatchSnapshot()` now supports anonymous snapshots: when snapshot name is missing, Playwright Test will generate one automatically: `expect('Web is Awesome <3').toMatchSnapshot();`
- New `maxDiffPixels` and `maxDiffPixelRatio` options for fine-grained screenshot comparison using `expect().toMatchSnapshot()`: `expect(await page.screenshot()).toMatchSnapshot({ maxDiffPixels: 27, // allow no more than 27 different pixels.});`

It is most convenient to specify `maxDiffPixels` or `maxDiffPixelRatio` once in `testConfig.expect`.

Playwright Test now adds `testConfig.fullyParallel` mode. By default, Playwright Test parallelizes between files. In fully parallel mode, tests inside a single file are also run in parallel. You can also use `--fully-parallel` command line flag.

```
playwright.config.tsexport default { fullyParallel: true,};
```

`testProject.grep` and `testProject.grepInvert` are now configurable per project. For example, you can now configure smoke tests project using `grep`:

```
playwright.config.tsexport default { projects: [ { name: 'smoke tests', grep: /@smoke/, }, ],};
```

Trace Viewer now shows API testing requests.

`locator.highlight()` visually reveals element(s) for easier debugging.

Announcements

We now ship a designated Python docker image `mcr.microsoft.com/playwright/python`. Please switch over to it if you use Python. This is

the last release that includes Python inside our javascript mcr.microsoft.com/playwright docker image.v1.20 is the last release to receive WebKit update for macOS 10.15 Catalina. Please update MacOS to keep using latest & greatest WebKit! Browser Versions Chromium 101.0.4921.0 Mozilla Firefox 97.0.1 WebKit 15.4 This version was also tested against the following stable channels: Google Chrome 99 Microsoft Edge 99 Version 1.19 Playwright Test Update Playwright Test v1.19 now 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.locator('#status')).toHaveText('Success');await expect.soft(page.locator('#eta')).toHaveText('1 day');// ... and continue the test to check more things.await page.locator('#next-page').click();await expect.soft(page.locator('#title')).toHaveText('Make another order');Read more in our documentation You can now specify a custom error message as a second argument to the expect and expect.soft functions, for example:await expect(page.locator('text=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.locator('text=Name'), 'should be logged in').toBeVisible(); | ^ 5 | }); 6 | Read more in our documentation By default, tests in a single file are run in order. If you have many independent tests in a single file, you can now run them in parallel with test.describe.configure(). Other Updates Locator now supports a has option that makes sure it contains another locator inside:await page.locator('article', { has: page.locator('.highlight'), }).click();Read more in locator documentation New locator.page() page.screenshot() and locator.screenshot() now automatically hide blinking caret Playwright Codegen now generates locators and frame locators New option url in testConfig.webServer to ensure your web server is ready before running the tests New testInfo.errors and testResult.errors that contain all failed assertions and soft assertions. Potentially breaking change in Playwright Test Global Setup It is unlikely that this change will affect you, no action is required if your tests keep running as they did. We've noticed that in rare cases, the set of tests to be executed was configured in the global setup by means of the environment variables. We also noticed some applications that were post processing the reporters' output in the global teardown. If you are doing one of the two, learn more Browser Versions Chromium 100.0.4863.0 Mozilla Firefox 96.0.1 WebKit 15.4 This version was also tested against the following stable channels: Google Chrome 98 Microsoft Edge 98 Version 1.18 Locator Improvements locator.dragTo() expect(locator).toBeChecked({ checked }) Each locator can now be optionally filtered by the text it contains:await page.locator('li', { hasText: 'my item' }).locator('button').click();Read more in locator documentation Testing API improvements expect(response).toBeOK() testInfo.attach() test.info() Improved TypeScript Support Playwright Test now respects tsconfig.json's baseUrl and paths, so you can use aliases There is a new environment variable PW_EXPERIMENTAL_TS_ESM that allows importing ESM modules in your TS code, without the need for the compile step. Don't forget the .js suffix when you are importing your esm modules. Run your tests as follows:npm i --save-dev @playwright/test@1.18.0-rc1 PW_EXPERIMENTAL_TS_ESM=1 npx playwright test Create Playwright The npm

init playwright command is now generally available for your use: # Run from your project's root directory `npm init playwright@latest` # Or create a new project `npm init playwright@latest new-project` This will create a Playwright Test configuration file, optionally add examples, a GitHub Action workflow and a first test example `spec.ts`. New APIs & changes

- `new TestCase.repeatEachIndex` API
- `acceptDownloads` option now defaults to `true`
- Breaking change: custom config options

Custom config options are a convenient way to parametrize projects with different values. Learn more in this guide. Previously, any fixture introduced through `test.extend()` could be overridden in the `testProject.use` config section. For example, // WRONG: THIS SNIPPET DOES NOT WORK SINCE v1.18. // fixtures.js

```
const test = base.extend({ myParameter: 'default' }); // playwright.config.js module.exports = { use: { myParameter: 'value', }, }; 
```

The proper way to make a fixture parametrized in the config file is to specify `option: true` when defining the fixture. For example, // CORRECT: THIS SNIPPET WORKS SINCE v1.18. // fixtures.js

```
const test = base.extend({ // Fixtures marked as "option: true" will get a value specified in the config, // or fallback to the default value. myParameter: ['default', { option: true }], }); // playwright.config.js module.exports = { use: { myParameter: 'value', }, }; 
```

Browser Versions Chromium 99.0.4812.0 Mozilla Firefox 95.0 WebKit 15.4 This version was also tested against the following stable channels: Google Chrome 97 Microsoft Edge 97

Version 1.17

Frame Locators Playwright 1.17 introduces frame locators - a locator to the iframe on the page. Frame locators capture the logic sufficient to retrieve the iframe and then locate elements in that iframe. Frame locators are strict by default, will wait for iframe to appear and can be used in Web-First assertions. Frame locators can be created with either `page.frameLocator()` or `locator.frameLocator()`

```
method. const locator = page.frameLocator('#my-iframe').locator('text=Submit'); await locator.click(); 
```

Read more at our documentation.

Trace Viewer Update Playwright Trace Viewer is now available online at <https://trace.playwright.dev>! Just drag-and-drop your trace.zip file to inspect its contents. NOTE: trace files are not uploaded anywhere; trace.playwright.dev is a progressive web application that processes traces locally.

Playwright Test traces now include sources by default (these could be turned off with tracing option)

Trace Viewer now shows test name

New trace metadata tab with browser details

Snapshots now have URL bar

HTML Report Update HTML report now supports dynamic filtering

Report is now a single static HTML file that could be sent by e-mail or as a slack attachment.

Ubuntu ARM64 support + more Playwright now supports Ubuntu 20.04 ARM64. You can now run Playwright tests inside Docker on Apple M1 and on Raspberry Pi. You can now use Playwright to install stable version of Edge on Linux: `npx playwright install msedge`

New APIs

- Tracing now supports a 'title' option
- Page navigations support a new 'commit' waiting option
- HTML reporter got new configuration option `testConfig.snapshotDir`
- `testInfo.parallelIndex`
- `testInfo.titlePath`
- `testOptions.trace` has new option `expect.toMatchSnapshot`
- supports `subdirectories`
- `reporter.printsToStdio()`

Version 1.16

Playwright Test API Testing Playwright 1.16 introduces new API Testing that lets you send requests to the server directly from Node.js! Now you can:

- test your server API
- prepare server side state before visiting the web application in a test
- validate server side post-conditions after running some actions in the browser

To do a request on behalf of Playwright's Page, use new `page.request` API:

```
import { test, expect } from '@playwright/test'; test('context fetch', async ({ page }) => { // Do a GET request on
```

behalf of page const response = await page.request.get('http://example.com/foo.json'); // ...});To do a stand-alone request from node.js to an API endpoint, use new request fixture:import { test, expect } from '@playwright/test';test('context fetch', async ({ request }) => { // Do a GET request on behalf of page const response = await request.get('http://example.com/foo.json'); // ...});Read more about it in our API testing guide.
 Response Interception It is now possible to do response interception by combining API Testing with request interception.For example, we can blur all the images on the page:import { test, expect } from '@playwright/test';import jimp from 'jimp'; // image processing librarytest('response interception', async ({ page }) => { await page.route('**/*.jpeg', async route => { const response = await page._request.fetch(route.request()); const image = await jimp.read(await response.body()); await image.blur(5); route.fulfill({ response, body: await image.getBufferAsync('image/jpeg'), }); }); const response = await page.goto('https://playwright.dev'); expect(response.status()).toBe(200);});Read more about response interception.
 New HTML reporter Try it out new HTML reporter with either --reporter=html or a reporter entry in playwright.config.ts file:\$ npx playwright test --reporter=htmlThe HTML reporter has all the information about tests and their failures, including surfacing trace and image artifacts.Read more about our reporters.
 Playwright Library locator.waitFor Wait for a locator to resolve to a single element with a given state. Defaults to the state: 'visible'.Comes especially handy when working with lists:import { test, expect } from '@playwright/test';test('context fetch', async ({ page }) => { const completeness = page.locator('text=Success'); await completeness.waitFor(); expect(await page.screenshot()).toMatchSnapshot('screen.png');});Read more about locator.waitFor().
 Docker support for Arm64 Playwright Docker image is now published for Arm64 so it can be used on Apple Silicon.Read more about Docker integration.
 Playwright Trace Viewer web-first assertions inside trace viewer
 run trace viewer with npx playwright show-trace and drop trace files to the trace viewer
 PWA API testing is integrated with trace viewer
 better visual attribution of action targets
 Read more about Trace Viewer.
 Browser Versions Chromium 97.0.4666.0Mozilla Firefox 93.0WebKit 15.4This version of Playwright was also tested against the following stable channels:Google Chrome 94Microsoft Edge 94Version 1.15
 Playwright Library  Mouse Wheel By using Page.mouse.wheel you are now able to scroll vertically or horizontally.
 New Headers API Previously it was not possible to get multiple header values of a response. This is now possible and additional helper functions are available:Request.allHeaders()Request.headersArray()Request.headerValue(name: string)Response.allHeaders()Response.headersArray()Response.headerValue(name: string)Response.headerValues(name: string)
 Forced-Colors emulation Its now possible to emulate the forced-colors CSS media feature by passing it in the context options or calling Page.emulateMedia().
 New APIs Page.route() accepts new times option to specify how many times this route should be matched.Page.setChecked(selector: string, checked: boolean) and Locator.setChecked(selector: string, checked: boolean) was introduced to set the checked state of a checkbox.
 Request.sizes() Returns resource size information for given http request.
 BrowserContext.tracing.startChunk() - Start a new trace chunk.
 BrowserContext.tracing.stopChunk() - Stops a new trace chunk.
 Playwright Test  test.parallel() run tests in the same file in parallel
 test.describe.parallel('group', () => { test('runs in parallel 1', async ({ page }) => { });

test('runs in parallel 2', async ({ page }) => { });});By default, tests in a single file are run in order. If you have many independent tests in a single file, you can now run them in parallel with test.describe.parallel(title, callback).
 Add --debug CLI flag By using npx playwright test --debug it will enable the Playwright Inspector for you to debug your tests.
 Browser Versions Chromium 96.0.4641.0Mozilla Firefox 92.0WebKit 15.0Version 1.14
 Playwright Library New "strict" mode Selector ambiguity is a common problem in automation testing. "strict" mode ensures that your selector points to a single element and throws otherwise.
 Pass strict: true into your action calls to opt in.// This will throw if you have more than one button!await page.click('button', { strict: true });
 New Locators API Locator represents a view to the element(s) on the page. It captures the logic sufficient to retrieve the element at any given moment. The difference between the Locator and ElementHandle is that the latter points to a particular element, while Locator captures the logic of how to retrieve that element. Also, locators are "strict" by default!
 const locator = page.locator('button');await locator.click();Learn more in the documentation.
 Experimental React and Vue selector engines React and Vue selectors allow selecting elements by its component name and/or property values. The syntax is very similar to attribute selectors and supports all attribute selector operators.
 await page.locator('_react=SubmitButton[enabled=true]').click();await page.locator('_vue=submit-button[enabled=true]').click();Learn more in the react selectors documentation and the vue selectors documentation.
 New nth and visible selector engines nth selector engine is equivalent to the :nth-match pseudo class, but could be combined with other selector engines.
 visible selector engine is equivalent to the :visible pseudo class, but could be combined with other selector engines.// select the first button among all buttonsawait button.click('button >> nth=0');// or if you are using locators, you can use first(), nth() and last()await page.locator('button').first().click();// click a visible buttonawait button.click('button >> visible=true');
 Playwright Test Web-First Assertions expect now supports lots of new web-first assertions. Consider the following example:await expect(page.locator('.status')).toHaveText('Submitted');
 Playwright Test will be re-testing the node with the selector .status until fetched Node has the "Submitted" text. It will be re-fetching the node and checking 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 testProject.expect value in test config.
 By default, the timeout for assertions is not set, so it'll wait forever, until the whole test times out.
 List of all new assertions:
 expect(locator).toBeChecked()
 expect(locator).toBeDisabled()
 expect(locator).toBeEditable()
 expect(locator).toBeEmpty()
 expect(locator).toBeEnabled()
 expect(locator).toBeFocused()
 expect(locator).toBeHidden()
 expect(locator).toBeVisible()
 expect(locator).toContainText(text, options?)
 expect(locator).toHaveAttribute(name, value)
 expect(locator).toHaveClass(expected)
 expect(locator).toHaveCount(count)
 expect(locator).toHaveCSS(name, value)
 expect(locator).toHaveId(id)
 expect(locator).toHaveJSProperty(name, value)
 expect(locator).toHaveText(expected, options)
 expect(page).toHaveTitle(title)
 expect(page).toHaveURL(url)
 expect(locator).toHaveValue(value)
 Serial mode with describe.serial Declares a group of tests that should always be run serially. If one of the tests fails, all subsequent tests are skipped. All tests in a group are retried together.
 test.describe.serial('group', () => { test('runs first', async ({ page }) => { /* ... */ }); test('runs second', async ({ page }) => { /* ... */ });});
 Learn more in the

documentation. [Steps API with test.step](#) Split long tests into multiple steps using `test.step()` API: `import { test, expect } from '@playwright/test'; test('test', async ({ page }) => { await test.step('Log in', async () => { // ... }); await test.step('news feed', async () => { // ... }); });` Step information is exposed in reporters API. [Launch web server before running tests](#) To launch a server during the tests, use the `webServer` option in the configuration file. The server will wait for a given url to be available before running the tests, and the url will be passed over to Playwright as a `baseURL` when creating a context. `playwright.config.ts` `import { defineConfig } from '@playwright/test'; export default defineConfig({ webServer: { command: 'npm run start', // command to launch url: 'http://127.0.0.1:3000', // url to await for timeout: 120 * 1000, reuseExistingServer: !process.env.CI, }, });` Learn more in the documentation. [Browser Versions](#) Chromium 94.0.4595.0 Mozilla Firefox 91.0 WebKit 15.0 Version 1.13 [Playwright Test & Reporter API](#) which is already used to create an Allure Playwright reporter. [New baseURL fixture to support relative paths in tests](#). [Playwright – Programmatic drag-and-drop support via the `page.dragAndDrop\(\)` API](#). [Enhanced HAR with body sizes for requests and responses](#). Use via `recordHar` option in `browser.newContext()`. [Tools](#) [Playwright Trace Viewer](#) now shows parameters, returned values and `console.log()` calls. [Playwright Inspector](#) can generate Playwright Test tests. [New and Overhauled Guides](#) [Intro Authentication Chrome Extensions](#) [Playwright Test Annotations](#) [Playwright Test Configuration](#) [Playwright Test Fixtures](#) [Browser Versions](#) Chromium 93.0.4576.0 Mozilla Firefox 90.0 WebKit 14.2 [New Playwright APIs](#) [new baseURL option in `browser.newContext\(\)` and `browser.newPage\(\)`](#) [response.securityDetails\(\)](#) and [response.serverAddr\(\)](#) [page.dragAndDrop\(\)](#) and [frame.dragAndDrop\(\)](#) [download.cancel\(\)](#) [page.inputValue\(\)](#), [frame.inputValue\(\)](#) and [elementHandle.inputValue\(\)](#) [new force option in `page.fill\(\)`, `frame.fill\(\)`, and `elementHandle.fill\(\)`](#) [new force option in `page.selectOption\(\)`, `frame.selectOption\(\)`, and `elementHandle.selectOption\(\)`](#) Version 1.12 [Introducing Playwright Test](#) Playwright Test is a new test runner built from scratch by Playwright team specifically to accommodate end-to-end testing needs: Run tests across all browsers. Execute tests in parallel. Enjoy context isolation and sensible defaults out of the box. Capture videos, screenshots and other artifacts on failure. Integrate your POMs as extensible fixtures. Installation: `npm i -D @playwright/test` Simple test `tests/foo.spec.ts`: `import { test, expect } from '@playwright/test'; test('basic test', async ({ page }) => { await page.goto('https://playwright.dev/'); const name = await page.innerText('.navbar__title'); expect(name).toBe('Playwright'); });` Running: `npx playwright test` [Read more in Playwright Test documentation](#). [Introducing Playwright Trace Viewer](#) Playwright Trace Viewer is a new GUI tool that helps exploring recorded Playwright traces after the script ran. Playwright traces let you examine: page DOM before and after each Playwright action page rendering before and after each Playwright action browser network during script execution Traces are recorded using the new `browserContext.tracing` API: `const browser = await chromium.launch(); const context = await browser.newContext(); // Start tracing before creating / navigating a page. await context.tracing.start({ screenshots: true, snapshots: true }); const page = await context.newPage(); await page.goto('https://playwright.dev/'); // Stop tracing and export it into a zip archive. await context.tracing.stop({ path: 'trace.zip' });` Traces are examined later with the Playwright CLI: `npx playwright show-trace trace.zip` That will open the following GUI: [Read more in](#)

trace viewer documentation. Browser Versions Chromium 93.0.4530.0 Mozilla Firefox 89.0 WebKit 14.2 This version of Playwright was also tested against the following stable channels: Google Chrome 91 Microsoft Edge 91 New APIs `reducedMotion` option in `page.emulateMedia()`, `browserType.launchPersistentContext()`, `browser.newContext()` and `browser.newPage()` `browserContext.on('request')` `browserContext.on('requestfailed')` `browserContext.on('requestfinished')` `browserContext.on('response')` `tracesDir` option in `browserType.launch()` and `browserType.launchPersistentContext()` new `browserContext.tracing` API namespace new `download.page()` method Version 1.11
 New video: Playwright: A New Test Automation Framework for the Modern Web (slides) We talked about Playwright Showed engineering work behind the scenes Did live demos with new features ' (Special thanks to applitools for hosting the event and inviting us! Browser Versions Chromium 92.0.4498.0 Mozilla Firefox 89.0b6 WebKit 14.2 New APIs support for async predicates across the API in methods such as `page.waitForRequest()` and others new emulation devices: Galaxy S8, Galaxy S9+, Galaxy Tab S4, Pixel 3, Pixel 4 new methods: `page.waitForURL()` to await navigations to URL `video.delete()` and `video.saveAs()` to manage screen recording new options: `screen` option in the `browser.newContext()` method to emulate window `screen dimensions` `position` option in `page.check()` and `page.uncheck()` method `trial` option to dry-run actions in `page.check()`, `page.uncheck()`, `page.click()`, `page.dblclick()`, `page.hover()` and `page.tap()` Version 1.10 Playwright for Java v1.10 is now stable! Run Playwright against Google Chrome and Microsoft Edge stable channels with the new channels API. Chromium screenshots are fast on Mac & Windows. Bundled Browser Versions Chromium 90.0.4430.0 Mozilla Firefox 87.0b10 WebKit 14.2 This version of Playwright was also tested against the following stable channels: Google Chrome 89 Microsoft Edge 89 New APIs `browserType.launch()` now accepts the new 'channel' option. Read more in our documentation. Version 1.9 Playwright Inspector is a new GUI tool to author and debug your tests. Line-by-line debugging of your Playwright scripts, with play, pause and step-through. Author new scripts by recording user actions. Generate element selectors for your script by hovering over elements. Set the `PWDEBUG=1` environment variable to launch the Inspector Pause script execution with `page.pause()` in headed mode. Pausing the page launches Playwright Inspector for debugging. New has-text pseudo-class for CSS selectors. `:has-text("example")` matches any element containing "example" somewhere inside, possibly in a child or a descendant element. See more examples. Page dialogs are now auto-dismissed during execution, unless a listener for dialog event is configured. Learn more about this. Playwright for Python is now stable with an idiomatic snake case API and pre-built Docker image to run tests in CI/CD. Browser Versions Chromium 90.0.4421.0 Mozilla Firefox 86.0b10 WebKit 14.1 New APIs `page.pause()`. Version 1.8 Selecting elements based on layout with `:left-of()`, `:right-of()`, `:above()` and `:below()`. Playwright now includes command line interface, former `playwright-cli.npx playwright --help` `page.selectOption()` now waits for the options to be present. New methods to assert element state like `page.isEditable()`. New APIs `elementHandle.isChecked()` `elementHandle.isDisabled()` `elementHandle.isEditable()` `elementHandle.isEnabled()` `elementHandle.isHidden()` `elementHandle.isVisible()` `page.isChecked()` `page.isDisabled()` `page.isEditable()` `page.isEnabled()` `page.isHidden()` `page.isVisible()`. New option 'editable' in `elementHandle.waitForElementState()`. Browser Versions Chromium 90.0.4392.0 Mozilla Firefox 85.0b5 WebKit 14.1 Version 1.7 New Java SDK: Playwright for

Java is now on par with JavaScript, Python and .NET bindings. Browser storage API: New convenience APIs to save and load browser storage state (cookies, local storage) to simplify automation scenarios with authentication. New CSS selectors: We heard your feedback for more flexible selectors and have revamped the selectors implementation. Playwright 1.7 introduces new CSS extensions and there's more coming soon. New website: The docs website at playwright.dev has been updated and is now built with Docusaurus. Support for Apple Silicon: Playwright browser binaries for WebKit and Chromium are now built for Apple Silicon. New APIs `browserContext.storageState()` to get current state for later reuse. `storageState` option in `browser.newContext()` and `browser.newPage()` to setup browser context state. Browser Versions Chromium 89.0.4344.0 Mozilla Firefox 84.0b9 WebKit 14.1

Canary releases Playwright for Node.js has a canary releases system. It permits you to test new unreleased features instead of waiting for a full release. They get released daily on the next NPM tag of Playwright. It is a good way to give feedback to maintainers, ensuring the newly implemented feature works as intended. **note** Using a canary release in production might seem risky, but in practice, it's not. A canary release passes all automated tests and is used to test e.g. the HTML report, Trace Viewer, or Playwright Inspector with end-to-end tests. `npm install -D @playwright/test@next` Next npm Dist Tag For any code-related commit on main, the continuous integration will publish a daily canary release under the `@next` npm dist tag. You can see on npm the current dist tags: `latest`: stable releases `next`: next releases, published daily `beta`: after a release-branch was cut, usually a week before a stable release each commit gets published under this tag **Using a Canary Release** `npm install -D @playwright/test@next` Documentation The stable and the next documentation is published on playwright.dev. To see the next documentation, press Shift on the keyboard 5 times.

Test configuration Playwright has many options to configure how your tests are run. You can specify these options in the configuration file. Note that test runner options are top-level, do not put them into the `use` section. **Basic Configuration** Here are some of the most common configuration options.

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  // Look for test files in the "tests" directory, relative to this configuration file.
  testDir: 'tests',
  // Run all tests in parallel.
  fullyParallel: true,
  // Fail the build on CI if you accidentally left test.only in the source code.
  forbidOnly: !!process.env.CI,
  // Retry on CI only.
  retries: process.env.CI ? 2 : 0,
  // Opt out of parallel tests on CI.
  workers: process.env.CI ? 1 : undefined,
  // Reporter to use
  reporter: 'html',
  use: {
    // Base URL to use in actions like `await page.goto('/')`.
    baseURL: 'http://127.0.0.1:3000',
    // Collect trace when retrying the failed test.
    trace: 'on-first-retry',
  },
  // Configure projects for major browsers.
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
  ],
  // Run your local dev server before starting the tests.
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
  },
});
```

OptionDescription `testConfig.forbidOnly` Whether to exit with an error if any tests are marked as `test.only`. Useful on CI. `testConfig.fullyParallel` have all tests in all files to run in parallel. See /Parallelism and sharding for more details. `testConfig.projects` Run tests in multiple configurations or on multiple browsers `testConfig.reporter` Reporter to use. See Test Reporters to learn more about

which reporters are available.

testConfig.retries The maximum number of retry attempts per test. See [Test Retries](#) to learn more about retries.

testConfig.testDir Directory with the test files.

testConfig.useOptions with `use()`

testConfig.webServer To launch a server during the tests, use the `webServer` option

testConfig.workers The maximum number of concurrent worker processes to use for parallelizing tests. Can also be set as percentage of logical CPU cores, e.g. '50%'. See [Parallelism and sharding](#) for more details.

Filtering Tests

Filter tests by glob patterns or regular expressions.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Glob patterns or regular expressions to ignore test files.
  testIgnore: ['*test-assets'], // Glob patterns or regular expressions that match test files.
  testMatch: ['*todo-tests/*.spec.ts'],
});
```

testConfig.testIgnore Glob patterns or regular expressions that should be ignored when looking for the test files. For example, `'*test-assets'`

testConfig.testMatch Glob patterns or regular expressions that match test files. For example, `'*todo-tests/*.spec.ts'`. By default, Playwright runs `.(test|spec).(js|ts|mjs)` files.

Advanced Configuration

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Folder for test artifacts such as screenshots, videos, traces, etc.
  outputDir: 'test-results', // path to the global setup files.
  globalSetup: require.resolve('./global-setup'), // path to the global teardown files.
  globalTeardown: require.resolve('./global-teardown'), // Each test is given 30 seconds.
  timeout: 30000,
});
```

testConfig.globalSetup Path to the global setup file. This file will be required and run before all the tests. It must export a single function.

testConfig.globalTeardown Path to the global teardown file. This file will be required and run after all the tests. It must export a single function.

testConfig.outputDir Folder for test artifacts such as screenshots, videos, traces, etc.

testConfig.timeout Playwright enforces a timeout for each test, 30 seconds by default. Time spent by the test function, fixtures, `beforeEach` and `afterEach` hooks is included in the test timeout.

Expect Options

Configuration for the `expect` assertion library.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  expect: {
    // Maximum time expect() should wait for the condition to be met.
    timeout: 5000,
    toHaveScreenshot: {
      // An acceptable amount of pixels that could be different, unset by default.
      maxDiffPixels: 10,
    },
    toMatchSnapshot: {
      // An acceptable ratio of pixels that are different to the total amount of pixels, between 0 and 1.
      maxDiffPixelRatio: 0.1,
    },
  },
});
```

testConfig.expect Web first assertions like `expect(locator).toHaveText()` have a separate timeout of 5 seconds by default. This is the maximum time the `expect()` should wait for the condition to be met. Learn more about test and expect timeouts and how to set them for a single test.

testConfig.expect `page().toHaveScreenshot()`

testConfig.expect `locator().toHaveScreenshot()`

testConfig.expect `snapshot().toMatchSnapshot()`

testConfig.expect `locator().toMatchSnapshot()` method.

Add custom matchers using `expect.extend`

You can extend Playwright assertions by providing custom matchers. These matchers will be available on the `expect` object. In this example we add a custom `toBeWithinRange` function in the configuration file. Custom matcher should return a message callback and a pass flag indicating whether the assertion passed.

```
TypeScript
JavaScript
playwright.config.ts
import { expect, defineConfig } from '@playwright/test';
expect.extend({
  toBeWithinRange(received: number, floor: number,
```

```

ceiling: number) {  const pass = received >= floor && received <= ceiling;  if (pass)
{    return {    message: () => 'passed',    pass: true,    };  } else {    return
{    message: () => 'failed',    pass: false,    };  } },,);export default
defineConfig({});playwright.config.tsconst { expect, defineConfig } =
require('@playwright/test');expect.extend({  toBeWithinRange(received, floor, ceiling)
{    const pass = received >= floor && received <= ceiling;  if (pass) {    return
{    message: () => 'passed',    pass: true,    };  } else {    return {    message:
() => 'failed',    pass: false,    };  } },,);module.exports = defineConfig({});Now we
can use toBeWithinRange in the test.example.spec.tsimport { test, expect } from
'@playwright/test';test('numeric ranges', () => {  expect(100).toBeWithinRange(90,
110);  expect(101).not.toBeWithinRange(0, 100);,});noteDo not confuse Playwright's
expect with the expect library. The latter is not fully integrated with Playwright test
runner, so make sure to use Playwright's own expect.For TypeScript, also add the
following to your global.d.ts. If it does not exist, you need to create it inside your
repository. Make sure that your global.d.ts gets included inside your tsconfig.json via
the include or compilerOptions.typeRoots option so that your IDE will pick it up.You
don't need it for JavaScript.global.d.tsexport {};declare global { namespace
PlaywrightTest {  interface Matchers<R, T> {    toBeWithinRange(a: number, b:
number): R;  } }}

```

Test use optionsIn addition to configuring the test runner you can also configure Emulation, Network and Recording for the Browser or BrowserContext. These options are passed to the use: {} object in the Playwright config.

Basic Options Set the base URL and storage state for all tests:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Base URL to use in actions like `await page.goto('/')`. baseURL: 'http://127.0.0.1:3000', // Populates context with given storage state. storageState:

'state.json', },,);OptionDescriptiontestOptions.baseURLBase URL used for all pages in the context. Allows navigating by using just the path, for example page.goto('/settings').testOptions.storageStatePopulates context with given storage state. Useful for easy authentication, learn more.

Emulation Options With Playwright you can emulate a real device such as a mobile phone or tablet. See our guide on projects for more info on emulating devices. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme". See our Emulation guide to learn

more.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Emulates `prefers-colors-scheme` media feature. colorScheme: 'dark', // Context geolocation. geolocation: { longitude: 12.492507, latitude: 41.889938 }, // Emulates the user locale. locale: 'en-GB', // Grants specified permissions to the browser context. permissions: ['geolocation'], // Emulates the user timezone. timezoneId: 'Europe/Paris', // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, },,);OptionDescriptiontestOptions.colorSchemeEmulates 'prefers-colors-scheme' media feature, supported values are 'light', 'dark', 'no-preference'testOptions.geolocationContext geolocation.testOptions.localeEmulates the user locale, for example en-GB, de-DE, etc.testOptions.permissionsA list of permissions to grant to all pages in the context.testOptions.timezoneIdChanges the

timezone of the context.`testOptions.viewport`Viewport used for all pages in the context.
Network Options Available options to configure
`networking: playwright.config.ts``import { defineConfig } from '@playwright/test';``export default defineConfig({ use: { // Whether to automatically download all the attachments. acceptDownloads: false, // An object containing additional HTTP headers to be sent with every request. extraHTTPHeaders: { 'X-My-Header': 'value', }, // Credentials for HTTP authentication. httpCredentials: { username: 'user', password: 'pass', }, // Whether to ignore HTTPS errors during navigation. ignoreHTTPSErrors: true, // Whether to emulate network being offline. offline: true, // Proxy settings used for all pages in the test. proxy: { server: 'http://myproxy.com:3128', bypass: 'localhost', }, }, });`
Option Description
`testOptions.acceptDownloads`Whether to automatically download all the attachments, defaults to true. Learn more about working with downloads.
`testOptions.extraHTTPHeaders`An object containing additional HTTP headers to be sent with every request. All header values must be strings.
`testOptions.httpCredentials`Credentials for HTTP authentication.
`testOptions.ignoreHTTPSErrors`Whether to ignore HTTPS errors during navigation.
`testOptions.offline`Whether to emulate network being offline.
`testOptions.proxy`Proxy settings used for all pages in the test.
noteYou don't have to configure anything to mock network requests. Just define a custom Route that mocks the network for a browser context. See our network mocking guide to learn more.
Recording Options With Playwright you can capture screenshots, record videos as well as traces of your test. By default these are turned off but you can enable them by setting the screenshot, video and trace options in your `playwright.config.js` file. Trace files, screenshots and videos will appear in the test output directory, typically `test-results`.
`playwright.config.ts``import { defineConfig } from '@playwright/test';``export default defineConfig({ use: { // Capture screenshot after each test failure. screenshot: 'only-on-failure', // Record trace only when retrying a test for the first time. trace: 'on-first-retry', // Record video only when retrying a test for the first time. video: 'on-first-retry' }, });`
Option Description
`testOptions.screenshot`Capture screenshots of your test. Options include 'off', 'on' and 'only-on-failure'.
`testOptions.trace`Playwright can produce test traces while running the tests. Later on, you can view the trace and get detailed information about Playwright execution by opening Trace Viewer. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'.
`testOptions.video`Playwright can record videos for your tests. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'.
Other Options
`playwright.config.ts``import { defineConfig } from '@playwright/test';``export default defineConfig({ use: { // Maximum time each action such as `click()` can take. Defaults to 0 (no limit). actionTimeout: 0, // Name of the browser that runs tests. For example `chromium`, `firefox`, `webkit`. browserName: 'chromium', // Toggles bypassing Content-Security-Policy. bypassCSP: true, // Channel to use, for example "chrome", "chrome-beta", "msedge", "msedge-beta". channel: 'chrome', // Run browser in headless mode. headless: false, // Change the default data-testid attribute. testIdAttribute: 'pw-test-id', }, });`
Option Description
`testOptions.actionTimeout`Timeout for each Playwright action in milliseconds. Defaults to 0 (no timeout). Learn more about timeouts and how to set them for a single test.
`testOptions.browserName`Name of the browser that runs tests. Defaults to 'chromium'. Options include chromium, firefox, or

`webkit.testOptions.bypassCSPToggles` bypassing Content-Security-Policy. Useful when CSP includes the production origin. Defaults to `false`.

`testOptions.channel` Browser channel to use. Learn more about different browsers and channels.

`testOptions.headless` Whether to run the browser in headless mode meaning no browser is shown when running tests. Defaults to `true`.

`testOptions.testIdAttribute` Changes the default data-testid attribute used by Playwright locators.

More browser and context options Any options accepted by `browserType.launch()` or `browser.newContext()` can be put into `launchOptions` or `contextOptions` respectively in the `use` section.

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    launchOptions: {
      slowMo: 50,
    },
  },
});
    
```

However, most common ones like `headless` or `viewport` are available directly in the `use` section - see basic options, emulation or network.

Explicit Context Creation and Option Inheritance If using the built-in browser fixture, calling `browser.newContext()` will create a context with options inherited from the config.

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    userAgent: 'some custom ua',
    viewport: { width: 100, height: 100 },
  },
});
    
```

An example test illustrating the initial context options are set:

```

test('should inherit use options on context when using built-in browser fixture', async ({ browser }) => {
  const context = await browser.newContext();
  const page = await context.newPage();
  expect(await page.evaluate(() => navigator.userAgent)).toBe('some custom ua');
  expect(await page.evaluate(() => window.innerWidth)).toBe(100);
  await context.close();
});
    
```

Configuration Scopes You can configure Playwright globally, per project, or per test. For example, you can set the locale to be used globally by adding `locale` to the `use` option of the Playwright config, and then override it for a specific project using the `project` option in the config. You can also override it for a specific test by adding `test.use({})` in the test file and passing in the options.

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    locale: 'en-GB',
  },
});
    
```

You can override options for a specific project using the `project` option in the Playwright config.

```

import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    {
      name: 'chromium',
      use: {
        ...devices['Desktop Chrome'],
        locale: 'de-DE',
      },
    },
  ],
});
    
```

You can override options for a specific test file by using the `test.use()` method and passing in the options. For example to run tests with the French locale for a specific test:

```

import { test, expect } from '@playwright/test';
test.use({ locale: 'fr-FR' });
test('example', async ({ page }) => {
  // ...
});
    
```

The same works inside a `describe` block. For example to run tests in a `describe` block with the French locale:

```

import { test, expect } from '@playwright/test';
test.describe('french language block', () => {
  test.use({ locale: 'fr-FR' });
  test('example', async ({ page }) => {
    // ...
  });
});
    
```

Test use options In addition to configuring the test runner you can also configure Emulation, Network and Recording for the Browser or BrowserContext. These options are passed to the `use: {}` object in the Playwright config.

Basic Options Set the base URL and storage state for all tests:

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    // Base URL to use in actions like `await page.goto('/')`.
    baseURL: 'http://127.0.0.1:3000',
    // Populates context with given storage state.
    storageState:
    
```

'state.json', },,});OptionDescriptiontestOptions.baseUrlBase URL used for all pages in the context. Allows navigating by using just the path, for example page.goto('/settings').testOptions.storageStatePopulates context with given storage state. Useful for easy authentication, learn more.

Emulation Options

With Playwright you can emulate a real device such as a mobile phone or tablet. See our guide on projects for more info on emulating devices. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme". See our Emulation guide to learn more.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Emulates `prefers-colors-scheme` media feature. colorScheme: 'dark', // Context geolocation. geolocation: { longitude: 12.492507, latitude: 41.889938 }, // Emulates the user locale. locale: 'en-GB', // Grants specified permissions to the browser context. permissions: ['geolocation'], // Emulates the user timezone. timezoneId: 'Europe/Paris', // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, },,});OptionDescriptiontestOptions.colorSchemeEmulates 'prefers-colors-scheme' media feature, supported values are 'light', 'dark', 'no-preference'
```

testOptions.geolocationContext geolocation.

testOptions.localeEmulates the user locale, for example en-GB, de-DE, etc.

testOptions.permissionsA list of permissions to grant to all pages in the context.

testOptions.timezoneIdChanges the timezone of the context.

testOptions.viewportViewport used for all pages in the context.

Network Options

Available options to configure

```
networking:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Whether to automatically download all the attachments. acceptDownloads: false, // An object containing additional HTTP headers to be sent with every request. extraHTTPHeaders: { 'X-My-Header': 'value', }, // Credentials for HTTP authentication. httpCredentials: { username: 'user', password: 'pass', }, // Whether to ignore HTTPS errors during navigation. ignoreHTTPSErrors: true, // Whether to emulate network being offline. offline: true, // Proxy settings used for all pages in the test. proxy: { server: 'http://myproxy.com:3128', bypass: 'localhost', }, },,});OptionDescriptiontestOptions.acceptDownloadsWhether to automatically download all the attachments, defaults to true. Learn more about working with downloads.
```

testOptions.extraHTTPHeadersAn object containing additional HTTP headers to be sent with every request. All header values must be strings.

testOptions.httpCredentialsCredentials for HTTP authentication.

testOptions.ignoreHTTPSErrorsWhether to ignore HTTPS errors during navigation.

testOptions.offlineWhether to emulate network being offline.

testOptions.proxyProxy settings used for all pages in the test.

noteYou don't have to configure anything to mock network requests. Just define a custom Route that mocks the network for a browser context. See our network mocking guide to learn more.

Recording Options

With Playwright you can capture screenshots, record videos as well as traces of your test. By default these are turned off but you can enable them by setting the screenshot, video and trace options in your playwright.config.js file. Trace files, screenshots and videos will appear in the test output directory, typically test-results.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default
```

`defineConfig({ use: { // Capture screenshot after each test failure. screenshot: 'only-on-failure', // Record trace only when retrying a test for the first time. trace: 'on-first-retry', // Record video only when retrying a test for the first time. video: 'on-first-retry' },});`

`OptionDescription` `testOptions.screenshot` Capture screenshots of your test. Options include 'off', 'on' and 'only-on-failure' `testOptions.trace` Playwright can produce test traces while running the tests. Later on, you can view the trace and get detailed information about Playwright execution by opening Trace Viewer. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry' `testOptions.video` Playwright can record videos for your tests. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'

Other Options `playwright.config.ts` `import { defineConfig } from '@playwright/test';` `export default` `defineConfig({ use: { // Maximum time each action such as `click()` can take. Defaults to 0 (no limit). actionTimeout: 0, // Name of the browser that runs tests. For example `chromium`, `firefox`, `webkit`. browserName: 'chromium', // Toggles bypassing Content-Security-Policy. bypassCSP: true, // Channel to use, for example "chrome", "chrome-beta", "msedge", "msedge-beta". channel: 'chrome', // Run browser in headless mode. headless: false, // Change the default data-testid attribute. testIdAttribute: 'pw-test-id', },});`

OptionDescription `testOptions.actionTimeout` Timeout for each Playwright action in milliseconds. Defaults to 0 (no timeout). Learn more about timeouts and how to set them for a single test. `testOptions.browserName` Name of the browser that runs tests. Defaults to 'chromium'. Options include chromium, firefox, or webkit. `testOptions.bypassCSP` Toggles bypassing Content-Security-Policy. Useful when CSP includes the production origin. Defaults to false. `testOptions.channel` Browser channel to use. Learn more about different browsers and channels. `testOptions.headless` Whether to run the browser in headless mode meaning no browser is shown when running tests. Defaults to true. `testOptions.testIdAttribute` Changes the default data-testid attribute used by Playwright locators.

More browser and context options Any options accepted by `browserType.launch()` or `browser.newContext()` can be put into `launchOptions` or `contextOptions` respectively in the use section.

`playwright.config.ts` `import { defineConfig } from '@playwright/test';` `export default` `defineConfig({ use: { launchOptions: { slowMo: 50, }, },});`

However, most common ones like headless or viewport are available directly in the use section - see basic options, emulation or network.

Explicit Context Creation and Option Inheritance If using the built-in browser fixture, calling `browser.newContext()` will create a context with options inherited from the config:

`playwright.config.ts` `import { defineConfig } from '@playwright/test';` `export default` `defineConfig({ use: { userAgent: 'some custom ua', viewport: { width: 100, height: 100 }, },});`

An example test illustrating the initial context options are set:

`import { test, expect } from '@playwright/test';` `test('should inherit use options on context when using built-in browser fixture', async ({ browser }) => { const context = await browser.newContext(); const page = await context.newPage(); expect(await page.evaluate(() => navigator.userAgent)).toBe('some custom ua'); expect(await page.evaluate(() => window.innerWidth)).toBe(100); await context.close();});`

Configuration Scopes You can configure Playwright globally, per project, or per test. For example, you can set the locale to be used globally by adding `locale` to the use option of the Playwright config, and then override it for a specific project using the `project` option in the config. You can also override it for a specific test

by adding `test.use({})` in the test file and passing in the options.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { locale: 'en-GB' },});
```

You can override options for a specific project using the `project` option in the Playwright config.

```
import { defineConfig, devices } from '@playwright/test';export default defineConfig({ projects: [ { name: 'chromium', use: { ...devices['Desktop Chrome'], locale: 'de-DE', }, }, ],});
```

You can override options for a specific test file by using the `test.use()` method and passing in the options. For example to run tests with the French locale for a specific test:

```
import { test, expect } from '@playwright/test';test.use({ locale: 'fr-FR' });test('example', async ({ page }) => { // ...});
```

The same works inside a `describe` block. For example to run tests in a `describe` block with the French locale:

```
import { test, expect } from '@playwright/test';test.describe('french language block', () => { test.use({ locale: 'fr-FR' }); test('example', async ({ page }) => { // ... });});
```

Annotations Playwright Test supports test annotations to deal with failures, flakiness, skip, focus and tag tests: `test.skip()` marks the test as irrelevant. Playwright Test does not run such a test. Use this annotation when the test is not applicable in some configuration. `test.fail()` marks the test as failing. Playwright Test will run this test and ensure it does indeed fail. If the test does not fail, Playwright Test will complain. `test.fixme()` marks the test as failing. Playwright Test will not run this test, as opposed to the fail annotation. Use `fixme` when running the test is slow or crashes. `test.slow()` marks the test as slow and triples the test timeout. Annotations can be used on a single test or a group of tests. Annotations can be conditional, in which case they apply when the condition is truthy. Annotations may depend on test fixtures. There could be multiple annotations on the same test, possibly in different configurations.

Focus a test

You can focus some tests. When there are focused tests, only these tests run. `test.only('focus this test', async ({ page }) => { // Run only focused tests in the entire project. });`

Skip a test

Mark a test as skipped. `test.skip('skip this test', async ({ page }) => { // This test is not run });`

Conditionally skip a test

You can skip certain test based on the condition. `test('skip this test', async ({ page, browserName }) => { test.skip(browserName === 'firefox', 'Still working on it'); });`

Group tests

You can group tests to give them a logical name or to scope before/after hooks to the group. `import { test, expect } from '@playwright/test'; test.describe('two tests', () => { test('one', async ({ page }) => { // ... }); test('two', async ({ page }) => { // ... }); });`

Tag tests

Sometimes you want to tag your tests as `@fast` or `@slow` and only run the tests that have the certain tag. We recommend that you use the `--grep` and `--grep-invert` command line flags for that: `import { test, expect } from '@playwright/test'; test('Test login page @fast', async ({ page }) => { // ... }); test('Test full report @slow', async ({ page }) => { // ... });`

You will then be able to run only that test: `npx playwright test --grep @fast` Or if you want the opposite, you can skip the tests with a certain tag: `npx playwright test --grep-invert @slow`

To run tests containing either tag (logical OR operator): `npx playwright test --grep "@fast|@slow"` Or run tests containing both tags (logical AND operator) using regex lookahead: `npx playwright test --grep "(?=.*@fast)(?=.*@slow)"`

Conditionally skip a group of tests

For example, you can run a group of tests just in Chromium by passing a callback. `example.spec.tstest.describe('chromium only', () => { test.skip(({ browserName }) => browserName !== 'chromium', 'Chromium only!'); test.beforeAll(async () => { // This hook is only run in Chromium. }); test('test 1', async ({ page }) => { // This test is only run in Chromium. }); test('test 2', async ({ page }) => { // This test is only run in Chromium. }); });`

Use `fixme` in `beforeEach` hook

To avoid running `beforeEach` hooks, you can put annotations in the hook itself. `example.spec.tstest.beforeEach(async ({ page, isMobile }) => { test.fixme(isMobile, 'Settings page does not work in mobile yet'); await page.goto('http://localhost:3000/settings'); }); test('user profile', async ({ page }) => { await page.getByText('My Profile').click(); // ... });`

Custom annotations

It's also possible to add custom metadata in the form of annotations to your tests. Annotations are key/value pairs accessible via `test.info().annotations`. Many reporters show annotations, for example `'html'`. `example.spec.tstest('user profile', async ({ page }) => { test.info().annotations.push({ type: 'issue', description: 'https://github.com/microsoft/playwright/issues/<some-issue>' }); // ... });`

Command line Here are the most common options available in the command line.

Run all the tests
`npx playwright test` Run a single test file
`npx playwright test tests/todo-page.spec.ts` Run a set of test files
`npx playwright test tests/todo-page/ tests/landing-page/` Run files that have my-spec or my-spec-2 in the file name
`npx playwright test my-spec my-spec-2` Run tests that are in line 42 in my-spec.ts
`npx playwright test my-spec.ts:42` Run the test with the title
`npx playwright test -g "add a todo item"` Run tests in headed browsers
`npx playwright test --headed` Run all the tests against a specific project
`npx playwright test --project=chromium` Disable parallelization
`npx playwright test --workers=1` Choose a reporter
`npx playwright test --reporter=dot` Run in debug mode with Playwright Inspector
`npx playwright test --debug` Run tests in interactive UI mode, with a built-in watch mode (Preview)
`npx playwright test --ui` Ask for help
`npx playwright test --help`

Reference Complete set of Playwright Test options is available in the configuration file. Following options can be passed to a command line and take priority over the configuration file:

Option	Description
<code>--headed</code>	Run tests in headed browsers. Useful for debugging.
<code>--browser</code>	Run test in a specific browser. Available options are "chromium", "firefox", "webkit" or "all" to run tests in all three browsers at the same time.
<code>--debug</code>	Run tests with Playwright Inspector. Shortcut for <code>PWDEBUG=1</code> environment variable and <code>--timeout=0 --max-failures=1 --headed --workers=1</code> options.
<code>-c <file></code> or <code>--config <file></code>	Configuration file. If not passed, defaults to <code>playwright.config.ts</code> or <code>playwright.config.js</code> in the current directory.
<code>-c <dir></code> or <code>--config <dir></code>	Configuration file. If not passed, defaults to <code>playwright.config.ts</code> or <code>playwright.config.js</code> in the current directory.
<code>--forbid-only</code>	Whether to disallow test.only. Useful on CI.
<code>-g <grep></code> or <code>--grep <grep></code>	Only run tests matching this regular expression. For example, this will run 'should add to cart' when passed <code>-g "add to cart"</code> .
<code>--grep-invert <grep></code>	Only run tests not matching this regular expression. The opposite of <code>--grep</code> .
<code>--global-timeout <number></code>	Total timeout for the whole test run in milliseconds. By default, there is no global timeout. Learn more about various timeouts.
<code>--list</code>	list all the tests, but do not run them.
<code>--max-failures <N></code> or <code>-x</code>	Stop after the first N test failures. Passing <code>-x</code> stops after the first failure.
<code>--output <dir></code>	Directory for artifacts produced by tests, defaults to <code>test-results</code> .
<code>--pass-with-no-tests</code>	Allows the test suite to pass when no files are found.
<code>--project <name></code>	Only run tests from one of the specified projects. Defaults to running all projects defined in the configuration file.
<code>--quiet</code>	Whether to suppress stdout and stderr from the tests.
<code>--repeat-each <N></code>	Run each test N times, defaults to one.
<code>--reporter <reporter></code>	Choose a reporter: minimalist dot, concise line or detailed list. See reporters for more information.
<code>--retries <number></code>	The maximum number of retries for flaky tests, defaults to zero (no retries).
<code>--shard <shard></code>	Shard tests and execute only selected shard, specified in the form <code>current/all</code> , 1-based, for example <code>3/5</code> .
<code>--timeout <number></code>	Maximum timeout in milliseconds for each test, defaults to 30 seconds. Learn more about various timeouts.
<code>--trace <mode></code>	Force tracing mode, can be on, off, on-first-retry, on-all-retries, retain-on-failure
<code>--ignore-snapshots</code>	Whether to ignore snapshots. Use this when snapshot expectations are known to be different, e.g. running tests on Linux against Windows screenshots.
<code>--update-snapshots</code> or <code>-u</code>	Whether to update snapshots with actual results instead of comparing them. Use this when snapshot expectations have changed.
<code>--workers <number></code> or <code>-j <number></code>	The maximum number of concurrent worker processes that run in parallel.

Emulation With Playwright you can test your app on any browser as well as emulate a

real device such as a mobile phone or tablet. Simply configure the devices you would like to emulate and Playwright will simulate the browser behavior such as "userAgent", "screenSize", "viewport" and if it "hasTouch" enabled. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme". Devices Playwright comes with a registry of device parameters using `playwright.devices` for selected desktop, tablet and mobile devices. It can be used to simulate browser behavior for a specific device such as user agent, screen size, viewport and if it has touch enabled. All tests will run with the specified device parameters.

```
TestLibraryplaywright.config.tsimport { defineConfig, devices } from '@playwright/test'; // import devicesexport default defineConfig({ projects: [ { name: 'chromium', use: { ...devices['Desktop Chrome'], }, }, { name: 'Mobile Safari', use: { ...devices['iPhone 13'], }, }, ],});const { chromium, devices } = require('playwright');const browser = await chromium.launch();const iphone13 = devices['iPhone 13'];const context = await browser.newContext({ ...iphone13,});
```

The viewport is included in the device but you can override it for some tests with

```
page.setViewportSize().TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, },});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1280, height: 1024 } });Test file:TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ viewport: { width: 1600, height: 1200 },});test('my test', async ({ page }) => { // ...});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1280, height: 1024 } });// Resize viewport for individual pageawait page.setViewportSize({ width: 1600, height: 1200 });// Emulate high-DPIconst context = await browser.newContext({ viewport: { width: 2560, height: 1440 }, deviceScaleFactor: 2,});
```

The same works inside a test file.

```
TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.describe('specific viewport block', () => { test.use({ viewport: { width: 1600, height: 1200 } }); test('my test', async ({ page }) => { // ... });});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1600, height: 1200 } });const page = await context.newPage();isMobile Whether the meta viewport tag is taken into account and touch events are enabled.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { isMobile: false, }, });Locale & Timezone Emulate the user Locale and Timezone which can be set globally for all tests in the config and then overridden for particular tests.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Emulates the user locale. locale: 'en-GB', // Emulates the user timezone. timezoneId: 'Europe/Paris', }, });TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ locale: 'de-DE', timezoneId: 'Europe/Berlin',});test('my test for de lang in Berlin timezone', async ({ page }) => { await page.goto('https://www.bing.com'); // ...});const context = await browser.newContext({ locale: 'de-DE', timezoneId: 'Europe/Berlin',});Permissions Allow app to show system notifications.TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Grants specified permissions to the browser context. permissions: ['notifications'], }, });const context =
```



```

await browser.newContext({ permissions: ['notifications'],});Allow notifications for a
specific domain.TestLibrarytests/example.spec.tsimport { test } from '@playwright/
test';test.beforeEach(async ({ context }) => { // Runs before each test and signs in each
page. await context.grantPermissions(['notifications'], { origin: 'https://
skype.com' });});test('first', async ({ page }) => { // page has notifications permission for
https://skype.com.});await context.grantPermissions(['notifications'], { origin: 'https://
skype.com' });Revoke all permissions with browserContext.clearPermissions().//
Libraryawait context.clearPermissions();Geolocation Grant "geolocation" permissions
and set geolocation to a specific area.playwright.config.tsimport { defineConfig } from
'@playwright/test';export default defineConfig({ use: { // Context geolocation
geolocation: { longitude: 12.492507, latitude: 41.889938 }, permissions:
['geolocation'], },});TestLibrarytests/example.spec.tsimport { test, expect } from
'@playwright/test';test.use({ geolocation: { longitude: 41.890221, latitude: 12.492348 },
permissions: ['geolocation'],});test('my test with geolocation', async ({ page }) =>
{ // ...});const context = await browser.newContext({ geolocation: { longitude:
41.890221, latitude: 12.492348 }, permissions: ['geolocation']});Change the location
later:TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/
test';test.use({ geolocation: { longitude: 41.890221, latitude: 12.492348 }, permissions:
['geolocation'],});test('my test with geolocation', async ({ page, context }) => { //
overwrite the location for this test await context.setGeolocation({ longitude: 48.858455,
latitude: 2.294474 });});await context.setGeolocation({ longitude: 48.858455, latitude:
2.294474 });Note you can only change geolocation for all pages in the context.Color
Scheme and Media Emulate the users "colorScheme". Supported values are 'light',
'dark', 'no-preference'. You can also emulate the media type with
page.emulateMedia().playwright.config.tsimport { defineConfig } from '@playwright/
test';export default defineConfig({ use: { colorScheme: 'dark', },});TestLibrarytests/
example.spec.tsimport { test, expect } from '@playwright/test';test.use({ colorScheme:
'dark' // or 'light'});test('my test with dark mode', async ({ page }) => { // ...});// Create
context with dark modeconst context = await browser.newContext({ colorScheme:
'dark' // or 'light'});// Create page with dark modeconst page = await
browser.newPage({ colorScheme: 'dark' // or 'light'});// Change color scheme for the
pageawait page.emulateMedia({ colorScheme: 'dark' });// Change media for pageawait
page.emulateMedia({ media: 'print' });User Agent The User Agent is included in the
device and therefore you will rarely need to change it however if you do need to test a
different user agent you can override it with the userAgent property.TestLibrarytests/
example.spec.tsimport { test, expect } from '@playwright/test';test.use({ userAgent: 'My
user agent' });test('my user agent test', async ({ page }) => { // ...});const context =
await browser.newContext({ userAgent: 'My user agent'});Offline Emulate the network
being offline.playwright.config.tsimport { defineConfig } from '@playwright/test';export
default defineConfig({ use: { offline: true },});JavaScript Enabled Emulate a user
scenario where JavaScript is disabled.TestLibrarytests/example.spec.tsimport { test,
expect } from '@playwright/test';test.use({ javascriptEnabled: false });test('test with no
JavaScript', async ({ page }) => { // ...});const context = await
browser.newContext({ javascriptEnabled: false});

```

FixturesPlaywright Test is based on the concept of test fixtures. Test fixtures are used to establish environment for each test, giving the test everything it needs and nothing else.

Test fixtures are isolated between tests. With fixtures, you can group tests based on their meaning, instead of their common setup. Built-in fixtures You have already used test fixtures in your first test.

```
import { test, expect } from '@playwright/test';
test('basic test', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  await expect(page).toHaveTitle(/Playwright/);
});
```

The `{ page }` argument tells Playwright Test to setup the page fixture and provide it to your test function. Here is a list of the pre-defined fixtures that you are likely to use most of the time:

- `FixtureTypeDescription` `page` `Page` `isolated page` for this test
- `run` `context` `BrowserContext` `isolated context` for this test run. The page fixture belongs to this context as well. Learn how to configure context.
- `browser` `Browser` `Browsers` are shared across tests to optimize resources. Learn how to configure browser.
- `browserName` `string` The name of the browser currently running the test. Either `chromium`, `firefox` or `webkit`.
- `requestAPI` `RequestContext` `isolated APIRequestContext` instance for this test run. Without fixtures Here is how typical test environment setup differs between traditional test style and the fixture-based one.

`TodoPage` is a class that helps interacting with a "todo list" page of the web app, following the Page Object Model pattern. It uses Playwright's page internally. Click to expand the code for the `TodoPage`.

```
import type { Page, Locator } from '@playwright/test';
export class TodoPage {
  private readonly inputBox: Locator;
  private readonly todoItems: Locator;
  constructor(public readonly page: Page) {
    this.inputBox = this.page.locator('input.new-todo');
    this.todoItems = this.page.getByTestId('todo-item');
  }
  async goto() {
    await this.page.goto('https://demo.playwright.dev/todomvc/');
  }
  async addToDo(text: string) {
    await this.inputBox.fill(text);
    await this.inputBox.press('Enter');
  }
  async remove(text: string) {
    const todo = this.todoItems.filter({ hasText: text });
    await todo.hover();
    await todo.getByLabel('Delete').click();
  }
  async removeAll() {
    while ((await this.todoItems.count()) > 0) {
      await this.todoItems.first().hover();
      await this.todoItems.getByLabel('Delete').first().click();
    }
  }
}

export class TodoPage {
  /**
   * @param {import('@playwright/test').Page} page
   */
  constructor(page) {
    this.page = page;
    this.inputBox = this.page.locator('input.new-todo');
    this.todoItems = this.page.getByTestId('todo-item');
  }
  async goto() {
    await this.page.goto('https://demo.playwright.dev/todomvc/');
  }
  /**
   * @param {string} text
   */
  async addToDo(text) {
    await this.inputBox.fill(text);
    await this.inputBox.press('Enter');
  }
  /**
   * @param {string} text
   */
  async remove(text) {
    const todo = this.todoItems.filter({ hasText: text });
    await todo.hover();
    await todo.getByLabel('Delete').click();
  }
  async removeAll() {
    while ((await this.todoItems.count()) > 0) {
      await this.todoItems.first().hover();
      await this.todoItems.getByLabel('Delete').first().click();
    }
  }
}

const { test } = require('@playwright/test');
const { TodoPage } = require('./todo-page');
test.describe('todo tests', () => {
  let todoPage;
  test.beforeEach(async ({ page }) => {
    todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo('item1');
    await todoPage.addToDo('item2');
  });
  test.afterEach(async () => {
    await todoPage.removeAll();
  });
  test('should add an item', async () => {
    await todoPage.addToDo('my item');
    // ...
  });
  test('should remove an item', async () => {
    await todoPage.remove('item1');
    // ...
  });
});
```

With fixtures Fixtures have a number of advantages over before/after hooks: Fixtures encapsulate setup and

teardown in the same place so it is easier to write. Fixtures are reusable between test files - you can define them once and use in all your tests. That's how Playwright's built-in page fixture works. Fixtures are on-demand - you can define as many fixtures as you'd like, and Playwright Test will setup only the ones needed by your test and nothing else. Fixtures are composable - they can depend on each other to provide complex behaviors. Fixtures are flexible. Tests can use any combinations of the fixtures to tailor precise environment they need, without affecting other tests. Fixtures simplify grouping. You no longer need to wrap tests in describes that set up environment, and are free to group your tests by their meaning instead. Click to expand the code for the

```

TodoPageTypeScriptJavaScripttodo-page.tsimport type { Page, Locator } from
'@playwright/test';export class TodoPage { private readonly inputBox: Locator; private
readonly todoItems: Locator; constructor(public readonly page: Page) { this.inputBox
= this.page.locator('input.new-todo'); this.todoItems = this.page.getByTestId('todo-
item'); } async goto() { await this.page.goto('https://demo.playwright.dev/
todomvc/'); } async addToDo(text: string) { await this.inputBox.fill(text); await
this.inputBox.press('Enter'); } async remove(text: string) { const todo =
this.todoItems.filter({ hasText: text }); await todo.hover(); await
todo.getByLabel('Delete').click(); } async removeAll() { while ((await
this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await
this.todoItems.getByLabel('Delete').first().click(); } }}todo-page.jsexport class
TodoPage { /** * @param {import('@playwright/test').Page} page */
constructor(page) { this.page = page; this.inputBox = this.page.locator('input.new-
todo'); this.todoItems = this.page.getByTestId('todo-item'); } async goto() { await
this.page.goto('https://demo.playwright.dev/todomvc/'); } /** * @param {string} text
*/ async addToDo(text) { await this.inputBox.fill(text); await
this.inputBox.press('Enter'); } /** * @param {string} text */ async remove(text)
{ const todo = this.todoItems.filter({ hasText: text }); await todo.hover(); await
todo.getByLabel('Delete').click(); } async removeAll() { while ((await
this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await
this.todoItems.getByLabel('Delete').first().click(); } }}
TypeScriptJavaScriptexample.spec.tsimport { test as base } from '@playwright/
test';import { TodoPage } from './todo-page';// Extend basic test by providing a
"todoPage" fixture.const test = base.extend<{ todoPage: TodoPage }>({ todoPage:
async ({ page }, use) => { const todoPage = new TodoPage(page); await
todoPage.goto(); await todoPage.addToDo('item1'); await
todoPage.addToDo('item2'); await use(todoPage); await
todoPage.removeAll(); },});test('should add an item', async ({ todoPage }) => { await
todoPage.addToDo('my item'); // ...});test('should remove an item', async ({ todoPage })
=> { await todoPage.remove('item1'); // ...});todo.spec.jsconst base =
require('@playwright/test');const { TodoPage } = require('./todo-page');// Extend basic
test by providing a "todoPage" fixture.const test = base.test.extend({ todoPage: async
({ page }, use) => { const todoPage = new TodoPage(page); await
todoPage.goto(); await todoPage.addToDo('item1'); await
todoPage.addToDo('item2'); await use(todoPage); await
todoPage.removeAll(); },});test('should add an item', async ({ todoPage }) => { await
todoPage.addToDo('my item'); // ...});test('should remove an item', async ({ todoPage })

```

```

=> { await todoPage.remove('item1'); // ...});Creating a fixture To create your own
fixture, use test.extend() to create a new test object that will include it.Below we create
two fixtures todoPage and settingsPage that follow the Page Object Model pattern.Click
to expand the code for the TodoPage and SettingsPageTypeScriptJavaScripttodo-
page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage
{ private readonly inputBox: Locator; private readonly todoItems: Locator;
constructor(public readonly page: Page) { this.inputBox = this.page.locator('input.new-
todo'); this.todoItems = this.page.getByTestId('todo-item'); } async goto() { await
this.page.goto('https://demo.playwright.dev/todomvc/'); } async addToDo(text: string)
{ await this.inputBox.fill(text); await this.inputBox.press('Enter'); } async
remove(text: string) { const todo = this.todoItems.filter({ hasText: text }); await
todo.hover(); await todo.getByLabel('Delete').click(); } async removeAll() { while
((await this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await
this.todoItems.getByLabel('Delete').first().click(); } }}todo-page.jsexport class
TodoPage { /** * @param {import('@playwright/test').Page} page */
constructor(page) { this.page = page; this.inputBox = this.page.locator('input.new-
todo'); this.todoItems = this.page.getByTestId('todo-item'); } async goto() { await
this.page.goto('https://demo.playwright.dev/todomvc/'); } /** * @param {string} text
*/ async addToDo(text) { await this.inputBox.fill(text); await
this.inputBox.press('Enter'); } /** * @param {string} text */ async remove(text)
{ const todo = this.todoItems.filter({ hasText: text }); await todo.hover(); await
todo.getByLabel('Delete').click(); } async removeAll() { while ((await
this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await
this.todoItems.getByLabel('Delete').first().click(); } }}SettingsPage is
similar:TypeScriptJavaScriptsettings-page.tsimport type { Page } from '@playwright/
test';export class SettingsPage { constructor(public readonly page: Page) { } async
switchToDarkMode() { // ... }}settings-page.jsexport class SettingsPage { /** *
@param {import('@playwright/test').Page} page */ constructor(page) { this.page =
page; } async switchToDarkMode() { // ... }}TypeScriptJavaScriptmy-test.tsimport
{ test as base } from '@playwright/test';import { TodoPage } from './todo-page';import
{ SettingsPage } from './settings-page';// Declare the types of your fixtures.type
MyFixtures = { todoPage: TodoPage; settingsPage: SettingsPage;};// Extend base test
by providing "todoPage" and "settingsPage".// This new "test" can be used in multiple
test files, and each of them will get the fixtures.export const test =
base.extend<MyFixtures>({ todoPage: async ({ page }, use) => { // Set up the
fixture. const todoPage = new TodoPage(page); await todoPage.goto(); await
todoPage.addToDo('item1'); await todoPage.addToDo('item2'); // Use the fixture
value in the test. await use(todoPage); // Clean up the fixture. await
todoPage.removeAll(); }, settingsPage: async ({ page }, use) => { await use(new
SettingsPage(page)); },});export { expect } from '@playwright/test';my-test.jsconst base
= require('@playwright/test');const { TodoPage } = require('./todo-page');const
{ SettingsPage } = require('./settings-page');// Extend base test by providing "todoPage"
and "settingsPage".// This new "test" can be used in multiple test files, and each of
them will get the fixtures.exports.test = base.test.extend({ todoPage: async ({ page },
use) => { // Set up the fixture. const todoPage = new TodoPage(page); await
todoPage.goto(); await todoPage.addToDo('item1'); await

```

```

todoPage.addToDo('item2'); // Use the fixture value in the test.  await
use(todoPage); // Clean up the fixture.  await todoPage.removeAll(); },
settingsPage: async ({ page }, use) => {  await use(new
SettingsPage(page)); },});exports.expect = base.expect;noteCustom fixture names
should start with a letter or underscore, and can contain only letters, numbers,
underscores.Using a fixture Just mention fixture in your test function argument, and test
runner will take care of it. Fixtures are also available in hooks and other fixtures. If you
use TypeScript, fixtures will have the right type.Below we use the todoPage and
settingsPage fixtures defined above.TypeScriptJavaScriptimport { test, expect } from './
my-test';test.beforeEach(async ({ settingsPage }) => {  await
settingsPage.switchToDarkMode();});test('basic test', async ({ todoPage, page }) =>
{  await todoPage.addToDo('something nice');  await expect(page.getByTestId('todo-
title')).toContainText(['something nice']);});const { test, expect } = require('./my-
test');test.beforeEach(async ({ settingsPage }) => {  await
settingsPage.switchToDarkMode();});test('basic test', async ({ todoPage, page }) =>
{  await todoPage.addToDo('something nice');  await expect(page.getByTestId('todo-
title')).toContainText(['something nice']);});Overriding fixtures In addition to creating your
own fixtures, you can also override existing fixtures to fit your needs. Consider the
following example which overrides the page fixture by automatically navigating to some
baseUrl:import { test as base } from '@playwright/test';export const test =
base.extend({  page: async ({ baseUrl, page }, use) => {  await
page.goto(baseUrl);  await use(page); },});Notice that in this example, the page
fixture is able to depend on other built-in fixtures such as testOptions.baseUrl. We can
now configure baseUrl in the configuration file, or locally in the test file with
test.use().example.spec.tstest.use({ baseUrl: 'https://playwright.dev' });Fixtures can
also be overridden where the base fixture is completely replaced with something
different. For example, we could override the testOptions.storageState fixture to provide
our own data.import { test as base } from '@playwright/test';export const test =
base.extend({  storageState: async ({}, use) => {  const cookie = await
getAuthCookie();  await use({ cookies: [cookie] }); },});Worker-scoped fixtures
Playwright Test uses worker processes to run test files. Similarly to how test fixtures are
set up for individual test runs, worker fixtures are set up for each worker process. That's
where you can set up services, run servers, etc. Playwright Test will reuse the worker
process for as many test files as it can, provided their worker fixtures match and hence
environments are identical.Below we'll create an account fixture that will be shared by
all tests in the same worker, and override the page fixture to login into this account for
each test. To generate unique accounts, we'll use the workerInfo.workerIndex that is
available to any test or fixture. Note the tuple-like syntax for the worker fixture - we have
to pass {scope: 'worker'} so that test runner sets up this fixture once per
worker.TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/
test';type Account = {  username: string;  password: string;};// Note that we pass worker
fixture types as a second template parameter.export const test = base.extend<{
{ account: Account }>({  account: [async ({ browser }, use, workerInfo) => {  // Unique
username.  const username = 'user' + workerInfo.workerIndex;  const password =
'verysecure';  // Create the account with Playwright.  const page = await
browser.newPage();  await page.goto('/signup');  await page.getByLabel('User

```

```

Name').fill(username); await page.getByLabel('Password').fill(password); await
page.getByText('Sign up').click(); // Make sure everything is ok. await
expect(page.getByTestId('result')).toHaveText('Success'); // Do not forget to cleanup.
await page.close(); // Use the account value. await use({ username, password }); },
{ scope: 'worker' }], page: async ({ page, account }, use) => { // Sign in with our
account. const { username, password } = account; await page.goto('/signin'); await
page.getByLabel('User Name').fill(username); await
page.getByLabel('Password').fill(password); await page.getByText('Sign in').click();
await expect(page.getByTestId('userinfo')).toHaveText(username); // Use signed-in
page in the test. await use(page); },,);export { expect } from '@playwright/test';my-
test.jsconst base = require('@playwright/test');exports.test =
base.test.extend({ account: [async ({ browser }, use, workerInfo) => { // Unique
username. const username = 'user' + workerInfo.workerIndex; const password =
'verysecure'; // Create the account with Playwright. const page = await
browser.newPage(); await page.goto('/signup'); await page.getByLabel('User
Name').fill(username); await page.getByLabel('Password').fill(password); await
page.getByText('Sign up').click(); // Make sure everything is ok. await
expect(page.locator('#result')).toHaveText('Success'); // Do not forget to cleanup.
await page.close(); // Use the account value. await use({ username, password }); },
{ scope: 'worker' }], page: async ({ page, account }, use) => { // Sign in with our
account. const { username, password } = account; await page.goto('/signin'); await
page.getByLabel('User Name').fill(username); await
page.getByLabel('Password').fill(password); await page.getByText('Sign in').click();
await expect(page.getByTestId('userinfo')).toHaveText(username); // Use signed-in
page in the test. await use(page); },,);exports.expect = base.expect;Automatic fixtures
Automatic fixtures are set up for each test/worker, even when the test does not list them
directly. To create an automatic fixture, use the tuple syntax and pass { auto: true }.Here
is an example fixture that automatically attaches debug logs when the test fails, so we
can later review the logs in the reporter. Note how it uses TestInfo object that is
available in each test/fixture to retrieve metadata about the test being
run.TypeScriptJavaScriptmy-test.tsimport * as debug from 'debug';import * as fs from
'fs';import { test as base } from '@playwright/test';export const test =
base.extend<{ saveLogs: void }>({ saveLogs: [async ({}, use, testInfo) => { //
Collecting logs during the test. const logs = []; debug.log = (...args) =>
logs.push(args.map(String).join("")); debug.enable('myserver'); await use(); // After
the test we can check whether the test passed or failed. if (testInfo.status !==
testInfo.expectedStatus) { // outputPath() API guarantees a unique file name.
const logFile = testInfo.outputPath('logs.txt'); await fs.promises.writeFile(logFile,
logs.join("\n"), 'utf8'); testInfo.attachments.push({ name: 'logs', contentType: 'text/
plain', path: logFile }); }, { auto: true }],,);export { expect } from '@playwright/test';my-
test.jsconst debug = require('debug');const fs = require('fs');const base =
require('@playwright/test');exports.test = base.test.extend({ saveLogs: [async ({}, use,
testInfo) => { // Collecting logs during the test. const logs = []; debug.log = (...args)
=> logs.push(args.map(String).join("")); debug.enable('myserver'); await use(); //
After the test we can check whether the test passed or failed. if (testInfo.status !==
testInfo.expectedStatus) { // outputPath() API guarantees a unique file name.

```

```

const logFile = testInfo.outputPath('logs.txt');    await fs.promises.writeFile(logFile,
logs.join('\n'), 'utf8');    testInfo.attachments.push({ name: 'logs', contentType: 'text/
plain', path: logFile });    } }, { auto: true }]);});
Fixture timeout By default, fixture shares
timeout with the test. However, for slow fixtures, especially worker-scoped ones, it is
convenient to have a separate timeout. This way you can keep the overall test timeout
small, and give the slow fixture more time.
TypeScriptJavaScriptimport { test as base,
expect } from '@playwright/test';const test = base.extend<{ slowFixture: string }
>({ slowFixture: [async ({}, use) => {    // ... perform a slow operation ...    await
use('hello'); }, { timeout: 60000 }]);test('example test', async ({ slowFixture }) =>
{    // ...});const { test: base, expect } = require('@playwright/test');const test =
base.extend({ slowFixture: [async ({}, use) => {    // ... perform a slow operation ...
await use('hello'); }, { timeout: 60000 }]);test('example test', async ({ slowFixture }) =>
{    // ...});
Fixtures-options noteOverriding custom fixtures in the config file has changed in
version 1.18. Learn more.
Playwright Test supports running multiple test projects that
can be separately configured. You can use "option" fixtures to make your configuration
options declarative and type-checked. Learn more about parametrizing tests.
Below we'll create a defaultItem option in addition to the todoPage fixture from other examples.
This option will be set in configuration file. Note the tuple syntax and { option: true }
argument.
Click to expand the code for the TodoPageTypeScriptJavaScripttodo-
page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage
{    private readonly inputBox: Locator;    private readonly todoItems: Locator;
    constructor(public readonly page: Page) {        this.inputBox = this.page.locator('input.new-
todo');        this.todoItems = this.page.getByTestId('todo-item');    }    async goto() {        await
this.page.goto('https://demo.playwright.dev/todomvc/');    }    async addToDo(text: string)
{        await this.inputBox.fill(text);        await this.inputBox.press('Enter');    }    async
remove(text: string) {        const todo = this.todoItems.filter({ hasText: text });        await
todo.hover();        await todo.getByLabel('Delete').click();    }    async removeAll() {        while
((await this.todoItems.count()) > 0) {            await this.todoItems.first().hover();            await
this.todoItems.getByLabel('Delete').first().click();        }    }
}
export class
TodoPage {    /**     * @param {import('@playwright/test').Page} page     */
    constructor(page) {        this.page = page;        this.inputBox = this.page.locator('input.new-
todo');        this.todoItems = this.page.getByTestId('todo-item');    }    async goto() {        await
this.page.goto('https://demo.playwright.dev/todomvc/');    }    /**     * @param {string} text     */
    async addToDo(text) {        await this.inputBox.fill(text);        await
this.inputBox.press('Enter');    }    /**     * @param {string} text     */
    async remove(text)
    {        const todo = this.todoItems.filter({ hasText: text });        await todo.hover();        await
todo.getByLabel('Delete').click();    }    async removeAll() {        while ((await
this.todoItems.count()) > 0) {            await this.todoItems.first().hover();            await
this.todoItems.getByLabel('Delete').first().click();        }    }
}
TypeScriptJavaScriptmy-
test.tsimport { test as base } from '@playwright/test';import { TodoPage } from './todo-
page';// Declare your options to type-check your configuration.export type MyOptions =
{    defaultItem: string;};type MyFixtures = {    todoPage: TodoPage;};// Specify both option
and fixture types.export const test = base.extend<MyOptions & MyFixtures>({    // Define
an option and provide a default value.    // We can later override it in the config.
    defaultItem: ['Something nice', { option: true }],    // Our "todoPage" fixture depends on
the option.    todoPage: async ({ page, defaultItem }, use) => {        const todoPage = new

```

```

    TodoPage(page); await todoPage.goto(); await todoPage.addToDo(defaultItem);
    await use(todoPage); await todoPage.removeAll(); },,);export { expect } from
    '@playwright/test';const base = require('@playwright/test');const { TodoPage }
    = require('./todo-page');exports.test = base.test.extend({ // Define an option and
    provide a default value. // We can later override it in the config. defaultItem:
    ['Something nice', { option: true }], // Our "todoPage" fixture depends on the option.
    todoPage: async ({ page, defaultItem }, use) => { const todoPage = new
    TodoPage(page); await todoPage.goto(); await todoPage.addToDo(defaultItem);
    await use(todoPage); await todoPage.removeAll(); },,);exports.expect =
    base.expect;We can now use todoPage fixture as usual, and set the defaultItem option
    in the config file.TypeScriptJavaScriptplaywright.config.tsimport { defineConfig } from
    '@playwright/test';import type { MyOptions } from './my-test';export default
    defineConfig<MyOptions>({ projects: [ { name: 'shopping', use: { defaultItem:
    'Buy milk' }, }, { name: 'wellbeing', use: { defaultItem:
    'Exercise!' }, }, ]});playwright.config.ts// @ts-checkconst { defineConfig } =
    require('@playwright/test');module.exports = defineConfig({ projects: [ { name:
    'shopping', use: { defaultItem: 'Buy milk' }, }, { name: 'wellbeing', use:
    { defaultItem: 'Exercise!' }, }, ]});Execution order Each fixture has a setup and
    teardown phase separated by the await use() call in the fixture. Setup is executed
    before the fixture is used by the test/hook, and teardown is executed when the fixture
    will not be used by the test/hook anymore.Fixtures follow these rules to determine the
    execution order:When fixture A depends on fixture B: B is always set up before A and
    teared down after A.Non-automatic fixtures are executed lazily, only when the test/hook
    needs them.Test-scoped fixtures are teared down after each test, while worker-scoped
    fixtures are only teared down when the worker process executing tests is
    shutdown.Consider the following example.TypeScriptJavaScriptimport { test as base }
    from '@playwright/test';const test = base.extend<{ testFixture: string, autoTestFixture:
    string, unusedFixture: string, }, { workerFixture: string, autoWorkerFixture: string, }
    >({ workerFixture: [async ({ browser }) => { // workerFixture setup... await
    use('workerFixture'); // workerFixture teardown... }, { scope: 'worker' }],
    autoWorkerFixture: [async ({ browser }) => { // autoWorkerFixture setup... await
    use('autoWorkerFixture'); // autoWorkerFixture teardown... }, { scope: 'worker', auto:
    true }], testFixture: [async ({ page, workerFixture }) => { // testFixture setup... await
    use('testFixture'); // testFixture teardown... }, { scope: 'test' }], autoTestFixture: [async
    () => { // autoTestFixture setup... await use('autoTestFixture'); // autoTestFixture
    teardown... }, { scope: 'test', auto: true }], unusedFixture: [async ({ page }) => { //
    unusedFixture setup... await use('unusedFixture'); // unusedFixture teardown... },
    { scope: 'test' }],,);test.beforeAll(async () => { /* ... */});test.beforeEach(async ({ page })
    => { /* ... */});test('first test', async ({ page }) => { /* ... */});test('second test', async
    ({ testFixture }) => { /* ... */});test.afterEach(async () => { /* ... */});test.afterAll(async ()
    => { /* ... */});const { test: base } = require('@playwright/test');const test =
    base.extend({ workerFixture: [async ({ browser }) => { // workerFixture setup... await
    use('workerFixture'); // workerFixture teardown... }, { scope: 'worker' }],
    autoWorkerFixture: [async ({ browser }) => { // autoWorkerFixture setup... await
    use('autoWorkerFixture'); // autoWorkerFixture teardown... }, { scope: 'worker', auto:
    true }], testFixture: [async ({ page, workerFixture }) => { // testFixture setup... await

```



```

use('testFixture'); // testFixture teardown... }, { scope: 'test' }], autoTestFixture: [async
() => { // autoTestFixture setup... await use('autoTestFixture'); // autoTestFixture
teardown... }, { scope: 'test', auto: true }], unusedFixture: [async ({ page }) => { //
unusedFixture setup... await use('unusedFixture'); // unusedFixture teardown... },
{ scope: 'test' }],,);test.beforeAll(async () => { /* ... */ });test.beforeEach(async ({ page })
=> { /* ... */ });test('first test', async ({ page }) => { /* ... */ });test('second test', async
({ testFixture }) => { /* ... */ });test.afterEach(async () => { /* ... */ });test.afterAll(async ()
=> { /* ... */ });Normally, if all tests pass and no errors are thrown, the order of execution
is as following.worker setup and beforeAll section:browser setup because it is required
by autoWorkerFixture.autoWorkerFixture setup because automatic worker fixtures are
always set up before anything else.beforeAll runs.first test section:autoTestFixture setup
because automatic test fixtures are always set up before test and beforeEach
hooks.page setup because it is required in beforeEach hook.beforeEach runs.first test
runs.afterEach runs.page teardown because it is a test-scoped fixture and should be
teared down after the test finishes.autoTestFixture teardown because it is a test-scoped
fixture and should be teared down after the test finishes.second test
section:autoTestFixture setup because automatic test fixtures are always set up before
test and beforeEach hooks.page setup because it is required in beforeEach
hook.beforeEach runs.workerFixture setup because it is required by testFixture that is
required by the second test.testFixture setup because it is required by the second
test.second test runs.afterEach runs.testFixture teardown because it is a test-scoped
fixture and should be teared down after the test finishes.page teardown because it is a
test-scoped fixture and should be teared down after the test finishes.autoTestFixture
teardown because it is a test-scoped fixture and should be teared down after the test
finishes.afterAll and worker teardown section:afterAll runs.workerFixture teardown
because it is a workers-scoped fixture and should be teared down once at the
end.autoWorkerFixture teardown because it is a workers-scoped fixture and should be
teared down once at the end.browser teardown because it is a workers-scoped fixture
and should be teared down once at the end.A few observations:page and
autoTestFixture are set up and teared down for each test, as test-scoped
fixtures.unusedFixture is never set up because it is not used by any tests/
hooks.testFixture depends on workerFixture and triggers its setup.workerFixture is
lazily set up before the second test, but teared down once during worker shutdown, as
a worker-scoped fixture.autoWorkerFixture is set up for beforeAll hook, but
autoTestFixture is not.

```

Global setup and teardownThere are two ways to configure global setup and teardown: using a global setup file and setting it in the config under globalSetup or using project dependencies. With project dependencies, you define a project that runs before all other projects. This is the recommended way to configure global setup as with Project dependencies your HTML report will show the global setup, trace viewer will record a trace of the setup and fixtures can be used.

Project Dependencies Project dependencies are a list of projects that need to run before the tests in another project run. They can be useful for configuring the global setup actions so that one project depends on this running first. Using dependencies allows global setup to produce traces and other artifacts.

Setup First we add a new project with the name 'setup'. We then give it a testProject.testMatch property in order to match the file called

global.setup.ts:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: /global.setup\.ts/, }, // { // other project // }]});Then we add the testProject.dependencies property to our projects that depend on the setup project and pass into the array the name of our dependency project, which we defined in the previous step:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: /global.setup\.ts/, }, { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup'], },]});Setup Example This example will show you how to use project dependencies to create a global setup that logs into an application and saves the state in storage state. This is useful if you want to run multiple tests that require a sign-in state and you want to avoid login for each test. The setup project will write the storage state into an 'playwright/.auth/user.json' file next to your playwright.config. By exporting a const of STORAGE_STATE we can then easily share the location of the storage file between projects with the StorageState method. This applies the storage state on the browser context with its cookies and a local storage snapshot. In this example the 'logged in chromium' project depends on the setup project whereas the 'logged out chromium' project does not depend on the setup project, and does not use the storageState option.playwright.config.tsimport { defineConfig } from '@playwright/test';export const STORAGE_STATE = path.join(__dirname, 'playwright/.auth/user.json');export default defineConfig({ use: { baseURL: 'http://localhost:3000/', }, projects: [{ name: 'setup', testMatch: /global.setup\.ts/, }, { name: 'logged in chromium', testMatch: '**/*.loggedin.spec.ts', dependencies: ['setup'], use: { ...devices['Desktop Chrome'], storageState: STORAGE_STATE, }, }, { name: 'logged out chromium', use: { ...devices['Desktop Chrome'] }, testIgnore: ['**/*.loggedin.spec.ts'], },]});We then create a setup test, stored at root level of your project, that logs in to an application and populates the context with the storage state after the login actions have been performed. By doing this you only have to log in once and the credentials will be stored in the STORAGE_STATE file, meaning you don't need to log in again for every test. Start by importing the STORAGE_STATE from the Playwright config file and then use this as the path to save your storage state to the page's context.global.setup.tsimport { test as setup, expect } from '@playwright/test';import { STORAGE_STATE } from '../playwright.config';setup('do login', async ({ page }) => { await page.goto('/'); await page.getByLabel('User Name').fill('user'); await page.getByLabel('Password').fill('password'); await page.getByText('Sign in').click(); // Wait until the page actually signs in. await expect(page.getByText('Hello, user!')).toBeVisible(); await page.context().storageState({ path: STORAGE_STATE });});tests/menu.loggedin.spec.tsimport { test, expect } from '@playwright/test';test.beforeEach(async ({ page }) => { await page.goto('/');});test('menu', async ({ page }) => { // You are signed in!});For a more detailed example check out our blog post: A better global setup in Playwright reusing login with project dependencies or check the v1.31 release video to see the demo.Tear down You can tear down your setup by adding a testProject.tearDown property to your setup project. This will run after all dependent projects have run. First we add a new project into the projects array and give it a name such as 'cleanup db'. We then

give it a `testProject.testMatch` property in order to match the file called `global.teardown.ts`:

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [ // { // setup project // }, { name: 'cleanup db', testMatch: /global.teardown\.ts/, }, // { // other project // } ]});
```

Then we add the `testProject.teardown` property to our setup project with the name 'cleanup db' which is the name we gave to our teardown project in the previous step:

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [ { name: 'setup db', testMatch: /global\.setup\.ts/, teardown: 'cleanup db', }, { name: 'cleanup db', testMatch: /global\.teardown\.ts/, }, // { // other project // } ]});
```

Teardown Example

Start by creating a `global.setup.ts` file at the root level of your project. This will be used to seed the database with some data before all tests have run.

`global.setup.ts` // seed the database with some data

Then create a `global.teardown.ts` file at the root level of your project. This will be used to delete the data from the database after all tests have run.

`global.teardown.ts` // delete the data from the database

In the Playwright config file: Add a project into the projects array and give it a name such as 'setup db'. Give it a `testProject.testMatch` property in order to match the file called `global.setup.ts`. Create another project and give it a name such as 'cleanup db'. Give it a `testProject.testMatch` property in order to match the file called `global.teardown.ts`. Add the `testProject.teardown` property to our setup project with the name 'cleanup db' which is the name given to our teardown project in the previous step. Finally add the 'chromium' project with the `testProject.dependencies` on the 'setup db' project.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [ { name: 'setup db', testMatch: /global\.setup\.ts/, teardown: 'cleanup db', }, { name: 'cleanup db', testMatch: /global\.teardown\.ts/, }, { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup db'], }, ]});
```

Configure globalSetup and globalTeardown

You can use the `globalSetup` option in the configuration file to set something up once before running all tests. The global setup file must export a single function that takes a config object. This function will be run once before all the tests. Similarly, use `globalTeardown` to run something once after all the tests. Alternatively, let `globalSetup` return a function that will be used as a global teardown. You can pass data such as port number, authentication tokens, etc. from your global setup to your tests using environment variables.

note: Using `globalSetup` and `globalTeardown` will not produce traces or artifacts. If you want to produce traces and artifacts, use project dependencies.

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ globalSetup: require.resolve('./global-setup'), globalTeardown: require.resolve('./global-teardown'),});
```

Example

Here is a global setup example that authenticates once and reuses authentication state in tests. It uses the `baseURL` and `storageState` options from the configuration file.

```
global-setup.tsimport { chromium, type FullConfig } from '@playwright/test';async function globalSetup(config: FullConfig) { const { baseURL, storageState } = config.projects[0].use; const browser = await chromium.launch(); const page = await browser.newPage(); await page.goto(baseURL!); await page.getByLabel('User Name').fill('user'); await page.getByLabel('Password').fill('password'); await page.getByText('Sign in').click(); await page.context().storageState({ path: storageState as string }); await
```

```

browser.close();}export default globalSetup;Specify globalSetup, baseUrl and
storageState in the configuration file.playwright.config.tsimport { defineConfig } from
'@playwright/test';export default defineConfig({ globalSetup: require.resolve('./global-
setup'), use: { baseUrl: 'http://localhost:3000/', storageState: 'state.json', },});Tests
start already authenticated because we specify storageState that was populated by
global setup.import { test } from '@playwright/test';test('test', async ({ page }) => { await
page.goto('/'); // You are signed in!});You can make arbitrary data available in your tests
from your global setup file by setting them as environment variables via
process.env.global-setup.tsimport type { FullConfig } from '@playwright/test';async
function globalSetup(config: FullConfig) { process.env.FOO = 'some data'; // Or a more
complicated data structure as JSON: process.env.BAR = JSON.stringify({ some:
'data' });}export default globalSetup;Tests have access to the process.env properties set
in the global setup.import { test } from '@playwright/test';test('test', async ({ page }) =>
{ // environment variables which are set in globalSetup are only available inside test().
const { FOO, BAR } = process.env; // FOO and BAR properties are populated.
expect(FOO).toEqual('some data'); const complexData = JSON.parse(BAR);
expect(BAR).toEqual({ some: 'data' });});Capturing trace of failures during global setup In
some instances, it may be useful to capture a trace of failures encountered during the
global setup. In order to do this, you must start tracing in your setup, and you must
ensure that you stop tracing if an error occurs before that error is thrown. This can be
achieved by wrapping your setup in a try...catch block. Here is an example that
expands the global setup example to capture a trace.global-setup.tsimport { chromium,
type FullConfig } from '@playwright/test';async function globalSetup(config: FullConfig)
{ const { baseUrl, storageState } = config.projects[0].use; const browser = await
chromium.launch(); const context = await browser.newContext(); const page = await
context.newPage(); try { await context.tracing.start({ screenshots: true, snapshots:
true }); await page.goto(baseUrl!); await page.getByLabel('User
Name').fill('user'); await page.getByLabel('Password').fill('password'); await
page.getByText('Sign in').click(); await context.storageState({ path: storageState as
string }); await context.tracing.stop({ path: './test-results/setup-trace.zip', });
await browser.close(); } catch (error) { await context.tracing.stop({ path: './test-
results/failed-setup-trace.zip', }); await browser.close(); throw error; }}export
default globalSetup;

```

Parallelism and sharding Playwright Test runs tests in parallel. In order to achieve that, it runs several worker processes that run at the same time. By default, test files are run in parallel. Tests in a single file are run in order, in the same worker process. You can configure tests using `test.describe.configure` to run tests in a single file in parallel. You can configure entire project to have all tests in all files to run in parallel using `testProject.fullyParallel` or `testConfig.fullyParallel`. To disable parallelism limit the number of workers to one. You can control the number of parallel worker processes and limit the number of failures in the whole test suite for efficiency. Worker processes All tests run in worker processes. These processes are OS processes, running independently, orchestrated by the test runner. All workers have identical environments and each starts its own browser. You can't communicate between the workers. Playwright Test reuses a single worker as much as it can to make testing faster, so multiple test files are usually run in a single worker one after another. Workers are always shutdown after a test

failure to guarantee pristine environment for following tests.

Limit workers

You can control the maximum number of parallel worker processes via command line or in the configuration file.

From the command line: `npx playwright test --workers 4`

In the configuration file: `playwright.config.ts`

```
import { defineConfig } from '@playwright/test';
export default defineConfig({ // Limit the number of workers on CI, use default locally
  workers: process.env.CI ? 2 : undefined,
});
```

Disable parallelism

You can disable any parallelism by allowing just a single worker at any time. Either set `workers: 1` option in the configuration file or pass `--workers=1` to the command line.

`npx playwright test --workers=1`

Parallelize tests in a single file

By default, tests in a single file are run in order. If you have many independent tests in a single file, you might want to run them in parallel with `test.describe.configure()`. Note that parallel tests are executed in separate worker processes and cannot share any state or global variables. Each test executes all relevant hooks just for itself, including `beforeAll` and `afterAll`.

```
import { test } from '@playwright/test';
test.describe.configure({ mode: 'parallel' });
test('runs in parallel 1', async ({ page }) => { /* ... */ });
test('runs in parallel 2', async ({ page }) => { /* ... */ });
```

Alternatively, you can opt-in all tests into this fully-parallel mode in the configuration file:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({ fullyParallel: true, });
```

You can also opt in for fully-parallel mode for just a few projects:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({ // runs all tests in all files of a specific project in parallel
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] }, fullyParallel: true },
  ],
});
```

Serial mode

You can annotate inter-dependent tests as serial. If one of the serial tests fails, all subsequent tests are skipped. All tests in a group are retried together. `noteUsing serial` is not recommended. It is usually better to make your tests isolated, so they can be run independently.

```
import { test, type Page } from '@playwright/test';
// Annotate entire file as serial.
test.describe.configure({ mode: 'serial' });
let page;
test.beforeAll(async ({ browser }) => { page = await browser.newPage(); });
test.afterAll(async () => { await page.close(); });
test('runs first', async () => { await page.goto('https://playwright.dev/'); });
test('runs second', async () => { await page.getByText('Get Started').click(); });
```

Shard tests between multiple machines

Playwright Test can shard a test suite, so that it can be executed on multiple machines. For that, pass `--shard=x/y` to the command line. For example, to split the suite into three shards, each running one third of the tests:

```
npx playwright test --shard=1/3
npx playwright test --shard=2/3
npx playwright test --shard=3/3
```

That way your test suite completes 3 times faster.

Limit failures and fail fast

You can limit the number of failed tests in the whole test suite by setting `maxFailures` config option or passing `--max-failures` command line flag. When running with "max failures" set, Playwright Test will stop after reaching this number of failed tests and skip any tests that were not executed yet. This is useful to avoid wasting resources on broken test suites.

Passing command line option: `npx playwright test --max-failures=10`

Setting in the configuration file:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({ // Limit the number of failures on CI to save resources
  maxFailures: process.env.CI ? 10 : undefined,
});
```

Worker index and parallel index

Each worker process is assigned two ids: a unique worker index that starts with 1, and a parallel index that is between 0 and `workers - 1`. When a worker is restarted, for example after a failure, the new worker process has the same `parallelIndex` and a new `workerIndex`. You can read

an index from environment variables `process.env.TEST_WORKER_INDEX` and `process.env.TEST_PARALLEL_INDEX`, or access them through `testInfo.workerIndex` and `testInfo.parallelIndex`.

Control test order Playwright Test runs tests from a single file in the order of declaration, unless you parallelize tests in a single file. There is no guarantee about the order of test execution across the files, because Playwright Test runs test files in parallel by default. However, if you disable parallelism, you can control test order by either naming your files in alphabetical order or using a "test list" file. Sort test files alphabetically. When you disable parallel test execution, Playwright Test runs test files in alphabetical order. You can use some naming convention to control the test order, for example `001-user-signin-flow.spec.ts`, `002-create-new-document.spec.ts` and so on. Use a "test list" file. `dangerTests` lists are discouraged and supported as a best-effort only. Some features such as VS Code Extension and tracing may not work properly with test lists. You can put your tests in helper functions in multiple files. Consider the following example where tests are not defined directly in the file, but rather in a wrapper function.

```
feature-a.spec.ts
import { test, expect } from '@playwright/test';
export default function createTests() {
  test('feature-a example test', async ({ page }) => {
    // ... test goes here
  });
}

feature-b.spec.ts
import { test, expect } from '@playwright/test';
export default function createTests() {
  test.use({ viewport: { width: 500, height: 500 } });
  test('feature-b example test', async ({ page }) => {
    // ... test goes here
  });
}
```

You can create a test list file that will control the order of tests - first run feature-b tests, then feature-a tests. Note how each test file is wrapped in a `test.describe()` block that calls the function where tests are defined. This way `test.use()` calls only affect tests from a single file.

```
test.list.ts
import { test } from '@playwright/test';
import featureBTests from './feature-b.spec.ts';
import featureATests from './feature-a.spec.ts';
test.describe(featureBTests);
test.describe(featureATests);
```

Now disable parallel execution by setting workers to one, and specify your test list file.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  workers: 1,
  testMatch: 'test.list.ts',
});
```

Do not define your tests directly in a helper file. This could lead to unexpected results because your tests are now dependent on the order of import/require statements. Instead, wrap tests in a function that will be explicitly called by a test list file, as in the example above.

Parameterize tests You can either parameterize tests on a test level or on a project level.

Parameterized Tests

```
example.spec.ts
const people = ['Alice', 'Bob'];
for (const name of people) {
  test(`testing with ${name}`, async () => {
    // ...
  });
}
// You can also do it with test.describe() or with multiple tests as long the test name is unique.
```

Parameterized Projects Playwright Test supports running multiple test projects at the same time. In the following example, we'll run two projects with different options. We declare the option `person` and set the value in the config. The first project runs with the value `Alice` and the second with the value `Bob`.

```
my-test.ts
import { test as base } from '@playwright/test';
export type TestOptions = {
  person: string;
};
export const test = base.extend<TestOptions>({
  // Define an option and provide a default value. // We can later override it in the config.
  person: ['John', { option: true }],
});

my-test.js
const base = require('@playwright/test').test;
const test = base.extend({
  // Define an option and provide a default value. // We can later override it in the config.
  person: ['John', { option: true }],
});

// We can use this option in the test, similarly to fixtures.
example.spec.ts
import { test } from './my-test';
test('test 1',
```

```

async ({ page, person }) => { await page.goto('/index.html'); await
expect(page.locator('#node')).toContainText(person); // ...});

```

Now, we can run tests in multiple configurations by using projects. TypeScript JavaScript playwright.config.ts

```

import { defineConfig } from '@playwright/test';
import type { TestOptions } from './my-test';
export default defineConfig<TestOptions>({
  projects: [
    { name: 'alice', use: { person: 'Alice' } },
    { name: 'bob', use: { person: 'Bob' } },
  ]
});

```

playwright.config.ts

```

// @ts-check
module.exports = defineConfig({
  projects: [
    { name: 'alice', use: { person: 'Alice' } },
    { name: 'bob', use: { person: 'Bob' } },
  ]
});

```

We can also use the option in a fixture. Learn more about fixtures. TypeScript JavaScript my-test.ts

```

import { test as base } from '@playwright/test';
export type TestOptions = { person: string };
export const test = base.extend<TestOptions>({
  // Define an option and provide a default value. // We can later override it in the config.
  person: ['John', { option: true }], // Override default "page" fixture.
  page: async ({ page, person }, use) => {
    await page.goto('/chat'); // We use "person" parameter as a "name" for the chat room.
    await page.getByLabel('User Name').fill(person);
    await page.getByText('Enter chat room').click(); // Each test will get a "page" that already has the person name.
    await use(page);
  },
});

```

my-test.js

```

const base = require('@playwright/test');
exports.test = base.test.extend({
  // Define an option and provide a default value. // We can later override it in the config.
  person: ['John', { option: true }], // Override default "page" fixture.
  page: async ({ page, person }, use) => {
    await page.goto('/chat'); // We use "person" parameter as a "name" for the chat room.
    await page.getByLabel('User Name').fill(person);
    await page.getByText('Enter chat room').click(); // Each test will get a "page" that already has the person name.
    await use(page);
  },
});

```

note Parameterized projects behavior has changed in version 1.18. Learn more. Passing Environment Variables

You can use environment variables to configure tests from the command line. For example, consider the following test file that needs a username and a password. It is usually a good idea not to store your secrets in the source code, so we'll need a way to pass secrets from outside.

```

example.spec.ts
test('example test', async ({ page }) => {
  // ...
  await page.getByLabel('User Name').fill(process.env.USERNAME);
  await page.getByLabel('Password').fill(process.env.PASSWORD);
});

```

You can run this test with your secret username and password set in the command line.

```

Bash
PowerShell
Batch
USERNAME=me PASSWORD=secret npx playwright test
$env:USERNAME=me$env:PASSWORD=secret npx playwright test
set USERNAME=me set PASSWORD=secret npx playwright test

```

Similarly, configuration file can also read environment variables passed through the command line.

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    baseURL: process.env.STAGING === '1' ? 'http://staging.example.test/' : 'http://example.test/',
  },
});

```

Now, you can run tests against a staging or a production environment.

```

Bash
PowerShell
Batch
STAGING=1 npx playwright test
$env:STAGING=1 npx playwright test
set STAGING=1 npx playwright test

```

env files To make environment variables easier to manage, consider something like .env files. Here is an example that uses dotenv package to read environment variables directly in the configuration file.

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
import dotenv from 'dotenv';
import path from 'path'; // Read from default ".env" file.
dotenv.config(); // Alternatively, read from "../my.env" file.
dotenv.config({ path:

```

path.resolve(__dirname, '..', 'my.env') });export default defineConfig({ use: { baseUrl: process.env.STAGING === '1' ? 'http://staging.example.test/' : 'http://example.test/', });});Now, you can just edit .env file to set any variables you'd like.# .env fileSTAGING=0USERNAME=mePASSWORD=secretRun tests as usual, your environment variables should be picked up.npx playwright testCreate tests via a CSV file The Playwright test-runner runs in Node.js, this means you can directly read files from the file system and parse them with your preferred CSV library. See for example this CSV file, in our example

```
input.csv:"test_case","some_value","some_other_value"
"value 1","value 11","foobar1"
"value 2","value 22","foobar21"
"value 3","value 33","foobar321"
"value 4","value 44","foobar4321"
Based on this we'll generate some tests by using the csv-parse library from NPM:
test.spec.ts
import fs from 'fs';
import path from 'path';
import { test } from '@playwright/test';
import { parse } from 'csv-parse/sync';
const records = parse(fs.readFileSync(path.join(__dirname, 'input.csv')), { columns: true, skip_empty_lines: true });
for (const record of records) {
  test(`foo: ${record.test_case}`, async ({ page }) => {
    console.log(record.test_case, record.some_value, record.some_other_value);
  });
}
```

ProjectsA project is logical group of tests running with the same configuration. We use projects so we can run tests on different browsers and devices. Projects are configured in the playwright.config.ts file and once configured you can then run your tests on all projects or only on a specific project. You can also use projects to run the same tests in different configurations. For example, you can run the same tests in a logged-in and logged-out state. By setting up projects you can also run a group of tests with different timeouts or retries or a group of tests against different environments such as staging and production, splitting tests per package/functionality and more. Configure projects for multiple browsers By using projects you can run your tests in multiple browsers such as chromium, webkit and firefox as well as branded browsers such as Google Chrome and Microsoft Edge. Playwright can also run on emulated tablet and mobile devices. See the registry of device parameters for a complete list of selected desktop, tablet and mobile devices.

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } },
    /* Test against mobile viewports. */
    { name: 'Mobile Chrome', use: { ...devices['Pixel 5'] } },
    { name: 'Mobile Safari', use: { ...devices['iPhone 12'] } },
    /* Test against branded browsers. */
    { name: 'Microsoft Edge', use: { ...devices['Desktop Edge'], channel: 'msedge' } },
    { name: 'Google Chrome', use: { ...devices['Desktop Chrome'], channel: 'chrome' } },
  ],
});
```

Run projects Playwright will run all projects by default.npx playwright testRunning 7 tests using 5 workers

```
[chromium] › example.spec.ts:3:1 › basic test (2s)
[firefox] › example.spec.ts:3:1 › basic test (2s)
[webkit] › example.spec.ts:3:1 › basic test (2s)
[Mobile Chrome] › example.spec.ts:3:1 › basic test (2s)
[Mobile Safari] › example.spec.ts:3:1 › basic test (2s)
[Microsoft Edge] › example.spec.ts:3:1 › basic test (2s)
[Google Chrome] › example.spec.ts:3:1 › basic test (2s)
```

Use the --project command line option to run a single project.npx playwright test --project=firefoxRunning 1 test using 1 worker

```
[firefox] › example.spec.ts:3:1 › basic test (2s)
```

The VS Code test

runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown from the testing sidebar and choose another profile or modify the default profile by clicking Select Default Profile and select the browsers you wish to run your tests on. Choose a specific profile, various profiles or all profiles to run tests on. Configure projects for multiple environments By setting up projects we can also run a group of tests with different timeouts or retries or run a group of tests against different environments. For example we can run our tests against a staging environment with 2 retries as well as against a production environment with 0 retries.

```
playwright.config.ts import { defineConfig } from '@playwright/test'; export default defineConfig({
  timeout: 60000, // Timeout is shared between all tests.
  projects: [
    { name: 'staging', use: { baseURL: 'staging.example.com', }, retries: 2, },
    { name: 'production', use: { baseURL: 'production.example.com', }, retries: 0, },
  ],
});
```

Splitting tests into projects We can split tests into projects and use filters to run a subset of tests. For example, we can create a project that runs tests using a filter matching all tests with a specific file name. We can then have another group of tests that ignore specific test files. Here is an example that defines a common timeout and two projects. The "Smoke" project runs a small subset of tests without retries, and "Default" project runs all other tests with retries.

```
playwright.config.ts import { defineConfig } from '@playwright/test'; export default defineConfig({
  timeout: 60000, // Timeout is shared between all tests.
  projects: [
    { name: 'Smoke', testMatch: /\.smoke.spec.ts/, retries: 0, },
    { name: 'Default', testIgnore: /\.smoke.spec.ts/, retries: 2, },
  ],
});
```

Dependencies Dependencies are a list of projects that need to run before the tests in another project run. They can be useful for configuring the global setup actions so that one project depends on this running first. When using project dependencies, test reporters will show the setup tests and the trace viewer will record traces of the setup. You can use the inspector to inspect the DOM snapshot of the trace of your setup tests and you can also use fixtures inside your setup. In this example the chromium, firefox and webkit projects depend on the setup project.

```
playwright.config.ts import { defineConfig } from '@playwright/test'; export default defineConfig({
  projects: [
    { name: 'setup', testMatch: '**/*.setup.ts', },
    { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup'], },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] }, dependencies: ['setup'], },
    { name: 'webkit', use: { ...devices['Desktop Safari'] }, dependencies: ['setup'], },
  ],
});
```

Running Sequence When working with tests that have a dependency, the dependency will always run first and once all tests from this project have passed, then the other projects will run in parallel. Running order: Tests in 'setup' project run Tests in 'chromium', 'webkit' and 'firefox' projects run in parallel. If there are more than one dependency then these project dependencies will be run first and in parallel. If the tests from a dependency fails then the tests that rely on this project will not be run. Running order: Tests in 'Browser Login' and 'Data Base' projects run in parallel 'Browser Login' passes 'Data Base' fails! "e2e tests" is not run!

Teardown You can also test Project.teardown your setup by adding a teardown property to your setup project. This will run after all dependent projects have run. See the teardown guide for more information.

Custom project parameters Projects can be also used to parametrize tests with your custom configuration - take a look at this separate guide.

Reporters Playwright Test comes with a few built-in reporters for different needs and ability to provide custom reporters. The easiest way to try out built-in reporters is to pass `--reporter` command line option. `npx playwright test --reporter=line` For more control, you can specify reporters programmatically in the configuration

```
file.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'line',});Multiple reporters You can use multiple reporters at the same time. For example you can use 'list' for nice terminal output and 'json' to get a comprehensive json file with the test results.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [ ['list'], ['json', { outputFile: 'test-results.json' } ] ],});Reporters on CI You can use different reporters locally and on CI. For example, using concise 'dot' reporter avoids too much output.
```

This is the default on CI.`playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Concise 'dot' for CI, default 'list' when running locally reporter: process.env.CI ? 'dot' : 'list',});`Built-in reporters All built-in reporters show detailed information about failures, and mostly differ in verbosity for successful runs.**List reporter** List reporter is default (except on CI where the dot reporter is default). It prints a line for each test being run.`npx playwright test --reporter=list`

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'list',});
```

Here is an example output in the middle of a test run. Failures will be listed at the end.`npx playwright test --reporter=list`

Running 124 tests using 6 workers

- 1 ' should access error in env (438ms)
- 2 ' handle long test names (515ms)
- 3 x 1) render expected (691ms)
- 4 ' should timeout (932ms)
- 5 should repeat each:
- 6 ' should respect enclosing .gitignore (569ms)
- 7 should teardown env after timeout:
- 8 should respect excluded tests:
- 9 ' should handle env beforeEach error (638ms)
- 10 should respect enclosing .gitignore:

You can opt into the step rendering via passing the following config option:`playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['list', { printSteps: true }]],});`**Line reporter** Line reporter is more concise than the list reporter. It uses a single line to report last finished test, and prints failures when they occur. Line reporter is useful for large test suites where it shows the progress but does not spam the output by listing all the tests.`npx playwright test --reporter=line`

```
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'line',});
```

Here is an example output in the middle of a test run. Failures are reported inline.`npx playwright test --reporter=line`

Running 124 tests using 6 workers

1) dot-reporter.spec.ts:20:1 › render expected

```
===== Error:
expect(received).toBe(expected) // Object.is equality
Expected: 1
Received: 0[23/124]
gitignore.spec.ts - should respect nested .gitignore
Dot reporter
Dot reporter is very concise - it only produces a single character per successful test run. It is the default on CI and useful where you don't want a lot of output.
npx playwright test --reporter=dot
playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'dot',});
```

Here is an example output in the middle of a test run. Failures will be listed at the end.`npx playwright test --reporter=dot`

Running 124 tests using 6 workers.....F.....

HTML reporter HTML reporter produces a self-contained folder that contains report for the test run that can be served as a web page.`npx playwright test --reporter=html` By default, HTML report is opened

automatically if some of the tests failed. You can control this behavior via the `open` property in the Playwright config. The possible values for that property are `always`, `never` and `on-failure` (default). You can also configure host and port that are used to serve the HTML report.

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['html', { open: 'never' }]] });
```

By default, report is written into the `playwright-report` folder in the current working directory. One can override that location using the `PLAYWRIGHT_HTML_REPORT` environment variable or a reporter configuration. In configuration file, pass options directly:

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['html', { outputFolder: 'my-report' }]] });
```

If you are uploading attachments from data folder to other location, you can use `attachmentsBaseURL` option to let html report where to look for them.

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['html', { attachmentsBaseURL: 'https://external-storage.com/' }]] });
```

A quick way of opening the last test run report is:

```
npx playwright show-report
```

Or if there is a custom folder name:

```
npx playwright show-report my-report
```

The html reporter currently does not support merging reports generated across multiple `--shards` into a single report. See this issue for available third party solutions.

JSON reporter

JSON reporter produces an object with all information about the test run. Most likely you want to write the JSON to a file. When running with `--reporter=json`, use `PLAYWRIGHT_JSON_OUTPUT_NAME` environment variable:

```
BashPowerShellBatchPLAYWRIGHT_JSON_OUTPUT_NAME=results.json
npx playwright test --reporter=json
```

Or in configuration file, pass options directly:

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['json', { outputFile: 'results.json' }]] });
```

JUnit reporter

JUnit reporter produces a JUnit-style xml report. Most likely you want to write the report to an xml file. When running with `--reporter=junit`, use `PLAYWRIGHT_JUNIT_OUTPUT_NAME` environment variable:

```
BashPowerShellBatchPLAYWRIGHT_JUNIT_OUTPUT_NAME=results.xml
npx playwright test --reporter=junit
```

Or in configuration file, pass options directly:

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['junit', { outputFile: 'results.xml' }]] });
```

GitHub Actions annotations

You can use the built in github reporter to get automatic failure annotations when running in GitHub actions. Note that all other reporters work on GitHub Actions as well, but do not provide annotations.

```
playwright.config.ts import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: [['github', { concise: 'dot' }]] });
```

// default 'list' when running locally

```
reporter: process.env.CI ? 'github' : 'list',
```

Custom reporters

You can create a custom reporter by implementing a class with some of the reporter methods. Learn more about the Reporter API.

```
my-awesome-reporter.ts import type { FullConfig, FullResult, Reporter, Suite, TestCase, TestResult }
```

```

from '@playwright/test/reporter';class MyReporter implements Reporter
{ onBegin(config: FullConfig, suite: Suite) { console.log(`Starting the run with
${suite.allTests().length} tests`); } onTestBegin(test: TestCase, result: TestResult)
{ console.log(`Starting test ${test.title}`); } onTestEnd(test: TestCase, result:
TestResult) { console.log(`Finished test ${test.title}: ${result.status}`); } onEnd(result:
FullResult) { console.log(`Finished the run: ${result.status}`); }}export default
MyReporter;Now use this reporter with testConfig.reporter.playwright.config.tsimport
{ defineConfig } from '@playwright/test';export default defineConfig({ reporter: './my-
awesome-reporter.ts',});Third party reporter showcase

```

AllureMonocartTesultsReportPortalCurrentsSerenity/JS

RetriesTest retries are a way to automatically re-run a test when it fails. This is useful when a test is flaky and fails intermittently. Test retries are configured in the configuration file.

Failures Playwright Test runs tests in worker processes. These processes are OS processes, running independently, orchestrated by the test runner.

All workers have identical environments and each starts its own browser. Consider the following snippet:

```

import { test } from '@playwright/test';test.describe('suite', () =>
{ test.beforeAll(async () => { /* ... */ }); test('first good', async ({ page }) => { /* ... */ });
test('second flaky', async ({ page }) => { /* ... */ }); test('third good', async ({ page }) =>
{ /* ... */ });});When all tests pass, they will run in order in the same worker

```

process. Worker process startsbeforeAll hook runsfirst good passessecond flaky

passesthird good passesShould any test fail, Playwright Test will discard the entire

worker process along with the browser and will start a new one. Testing will continue in

the new worker process starting with the next test. Worker process #1 startsbeforeAll

hook runsfirst good passessecond flaky failsWorker process #2 startsbeforeAll hook

runs againthird good passesIf you enable retries, second worker process will start by

retrying the failed test and continue from there. Worker process #1 startsbeforeAll hook

runsfirst good passessecond flaky failsWorker process #2 startsbeforeAll hook runs

againsecond flaky is retried and passesthird good passesThis scheme works perfectly

for independent tests and guarantees that failing tests can't affect healthy ones.

Retries Playwright supports test retries. When enabled, failing tests will be retried multiple times

until they pass, or until the maximum number of retries is reached. By default failing

tests are not retried. # Give failing tests 3 retry attemptsnpx playwright test --

retries=3You can configure retries in the configuration file:playwright.config.tsimport

```

{ defineConfig } from '@playwright/test';export default defineConfig({ // Give failing tests
3 retry attempts  retries: 3,});Playwright Test will categorize tests as follows:"passed" -

```

tests that passed on the first run;"flaky" - tests that failed on the first run, but passed

when retried;"failed" - tests that failed on the first run and failed all retries.

Running 3 tests using 1 worker ' example.spec.ts:4:2 › first passes (438ms) x example.spec.ts:5:2

› second flaky (691ms) ' example.spec.ts:5:2 › second flaky (522ms) ' example.spec.ts:6:2

› third passes (932ms) 1 flaky example.spec.ts:5:2 › second flaky 2 passed (4s)

You can detect retries at runtime with testInfo.retry, which is accessible to any test, hook or fixture.

Here is an example that clears some server-side state before a retry.

```

import { test, expect } from '@playwright/test';test('my test', async
({ page }, testInfo) => { if (testInfo.retry) await

```

```

cleanSomeCachesOnTheServer(); // ...});You can specify retries for a specific group of

```

```

tests or a single file with test.describe.configure().import { test, expect } from

```

```

test.describe.configure({ retries: 3 });

```

'@playwright/test';test.describe(() => { // All tests in this describe group will get 2 retry attempts. test.describe.configure({ retries: 2 }); test('test 1', async ({ page }) => { // ... }); test('test 2', async ({ page }) => { // ... });});Serial mode Use test.describe.serial() to group dependent tests to ensure they will always run together and in order. If one of the tests fails, all subsequent tests are skipped. All tests in the group are retried together.Consider the following snippet that uses test.describe.serial:import { test } from '@playwright/test';test.describe.configure({ mode: 'serial' });test.beforeAll(async () => { /* ... */ });test('first good', async ({ page }) => { /* ... */ });test('second flaky', async ({ page }) => { /* ... */ });test('third good', async ({ page }) => { /* ... */ });When running without retries, all tests after the failure are skipped:Worker process #1:beforeAll hook runsfirst good passessecond flaky failsthird good is skipped entirelyWhen running with retries, all tests are retried together:Worker process #1:beforeAll hook runsfirst good passessecond flaky failsthird good is skippedWorker process #2:beforeAll hook runs againfirst good passes againsecond flaky passesthird good passesnotelt is usually better to make your tests isolated, so they can be efficiently run and retried independently.Reuse single page between tests Playwright Test creates an isolated Page object for each test. However, if you'd like to reuse a single Page object between multiple tests, you can create your own in test.beforeAll() and close it in test.afterAll().TypeScriptJavaScriptexample.spec.tsimport { test, type Page } from '@playwright/test';test.describe.configure({ mode: 'serial' });let page: Page;test.beforeAll(async ({ browser }) => { page = await browser.newPage();});test.afterAll(async () => { await page.close();});test('runs first', async () => { await page.goto('https://playwright.dev/');});test('runs second', async () => { await page.getByText('Get Started').click();});example.spec.js// @ts-checkconst { test } = require('@playwright/test');test.describe.configure({ mode: 'serial' });/* @type {import('@playwright/test').Page} */let page;test.beforeAll(async ({ browser }) => { page = await browser.newPage();});test.afterAll(async () => { await page.close();});test('runs first', async () => { await page.goto('https://playwright.dev/');});test('runs second', async () => { await page.getByText('Get Started').click();});

TimeoutsPlaywright Test has multiple configurable timeouts for various tasks.**Timeout**DefaultDescriptionTest timeout30000 msTimeout for each test, includes test, hooks and fixtures:Set defaultconfig = { timeout: 60000 }**Override**test.setTimeout(120000)Expect timeout5000 msTimeout for each assertion:Set defaultconfig = { expect: { timeout: 10000 } }**Override**expect(locator).toBeVisible({ timeout: 10000 })Test timeout Playwright Test enforces a timeout for each test, 30 seconds by default. Time spent by the test function, fixtures, beforeEach and afterEach hooks is included in the test timeout.Timed out test produces the following error:example.spec.ts:3:1 › basic test

```
=====
Timeout of 30000ms exceeded.The same timeout value also applies to beforeAll and afterAll hooks, but they do not share time with any test.Set test timeout in the config playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ timeout: 5 * 60 * 1000,});API reference: testConfig.timeout.Set timeout for a single test import { test, expect } from '@playwright/test';test('slow test', async ({ page }) => { test.slow(); // Easy way to triple the default timeout // ...});test('very slow test', async ({ page }) =>
```

`{ test.setTimeout(120000); // ...});`API reference: `test.setTimeout()` and `test.slow()`. Change timeout from a `beforeEach` hook `import { test, expect } from '@playwright/test'; test.beforeEach(async ({ page }, testInfo) => { // Extend timeout for all tests running this hook by 30 seconds. testInfo.setTimeout(testInfo.timeout + 30000);});`API reference: `testInfo.setTimeout()`. Change timeout for `beforeAll/afterAll` hook `beforeAll` and `afterAll` hooks have a separate timeout, by default equal to test timeout. You can change it separately for each hook by calling `testInfo.setTimeout()` inside the hook.`import { test, expect } from '@playwright/test'; test.beforeAll(async () => { // Set timeout for this hook. test.setTimeout(60000);});`API reference: `testInfo.setTimeout()`. Expect timeout Web-first assertions like `expect(locator).toHaveText()` have a separate timeout, 5 seconds by default. Assertion timeout is unrelated to the test timeout. It produces the following error:
`example.spec.ts:3:1 › basic test =====Error: expect(received).toHaveText(expected)Expected string: "my text"Received string: ""Call log: - expect.toHaveText with timeout 5000ms - waiting for "locator('button')"`Set expect timeout in the config `playwright.config.ts``import { defineConfig } from '@playwright/test'; export default defineConfig({ expect: { timeout: 10 * 1000, },});`Global timeout Playwright Test supports a timeout for the whole test run. This prevents excess resource usage when everything went wrong. There is no default global timeout, but you can set a reasonable one in the config, for example one hour. Global timeout produces the following error:Running 1000 tests using 10 workers 514 skipped 486 passed Timed out waiting 3600s for the entire test runYou can set global timeout in the config.`// playwright.config.tsimport { defineConfig } from '@playwright/test'; export default defineConfig({ globalTimeout: 60 * 60 * 1000,});`API reference: `testConfig.globalTimeout`. Advanced: low level timeouts These are the low-level timeouts that are pre-configured by the test runner, you should not need to change these. If you happen to be in this section because your test are flaky, it is very likely that you should be looking for the solution elsewhere.`TimeoutDefaultDescriptionAction timeoutno timeoutTimeout for each action:Set defaultconfig = { use: { actionTimeout: 10000 } } Overridelocator.click({ timeout: 10000 })Navigation timeoutno timeoutTimeout for each navigation action:Set defaultconfig = { use: { navigationTimeout: 30000 } } Overridepage.goto('/', { timeout: 30000 })Global timeoutno timeoutGlobal timeout for the whole test run:Set in configconfig = { globalTimeout: 60*60*1000 }beforeAll/afterAll timeout30000 msTimeout for the hook:Set in hooktest.setTimeout(60000)Fixture timeoutno timeoutTimeout for an individual fixture:Set in fixture{ scope: 'test', timeout: 30000 }Set timeout for a single assertion import { test, expect } from '@playwright/test'; test('basic test', async ({ page }) => { await expect(page.getByRole('button')).toHaveText('Sign in', { timeout: 10000 });});Set action and navigation timeouts in the config playwright.config.tsimport { defineConfig } from '@playwright/test'; export default defineConfig({ use: { actionTimeout: 10 * 1000, navigationTimeout: 30 * 1000, },});API reference: testOptions.actionTimeout and testOptions.navigationTimeout. Set timeout for a single action import { test, expect } from '@playwright/test'; test('basic test', async ({ page }) => { await page.goto('https://playwright.dev', { timeout: 30000 }); await page.getByText('Get Started').click({ timeout: 10000 });});Fixture timeout By default, fixture shares timeout with the test. However, for slow fixtures, especially worker-scoped ones, it is convenient to have a separate`

timeout. This way you can keep the overall test timeout small, and give the slow fixture more time.

```
TypeScriptJavaScriptimport { test as base, expect } from '@playwright/test';const test = base.extend<{ slowFixture: string }>({ slowFixture: [async ({}, use) => { // ... perform a slow operation ... await use('hello'); }, { timeout: 60000 }]);test('example test', async ({ slowFixture }) => { // ...});const { test: base, expect } = require('@playwright/test');const test = base.extend({ slowFixture: [async ({}, use) => { // ... perform a slow operation ... await use('hello'); }, { timeout: 60000 }]);test('example test', async ({ slowFixture }) => { // ...});API reference: test.extend().
```

TypeScriptPlaywright supports TypeScript out of the box. You just write tests in TypeScript, and Playwright will read them, transform to JavaScript and run.tsconfig.json Playwright will pick up tsconfig.json for each source file it loads. Note that Playwright only supports the following tsconfig options: paths and baseUrl.We recommend setting up a separate tsconfig.json in the tests directory so that you can change some preferences specifically for the tests. Here is an example directory structure.

```
src/
source.tstests/  tsconfig.json # test-specific tsconfig  example.spec.tststconfig.json # generic tsconfig for all typescript sourcesplaywright.config.tststconfig path mapping
```

Playwright supports path mapping declared in the tsconfig.json. Make sure that baseUrl is also set. Here is an example tsconfig.json that works with Playwright Test:

```
{ "compilerOptions": { "baseUrl": ".", // This must be specified if "paths" is. "paths": { "@myhelper/*": ["packages/myhelper/*"] // This mapping is relative to "baseUrl". } }}You can now import using the mapped paths:example.spec.tsimport { test, expect } from '@playwright/test';import { username, password } from '@myhelper/credentials';test('example', async ({ page }) => { await page.getByLabel('User Name').fill(username); await page.getByLabel('Password').fill(password);});Manually compile tests with TypeScript Sometimes, Playwright Test will not be able to transform your TypeScript code correctly, for example when you are using experimental or very recent features of TypeScript, usually configured in tsconfig.json. In this case, you can perform your own TypeScript compilation before sending the tests to Playwright. First add a tsconfig.json file inside the tests directory:{ "compilerOptions": { "target": "ESNext", "module": "commonjs", "moduleResolution": "Node", "sourceMap": true, "outDir": "../tests-out", }}In package.json, add two scripts:{ "scripts": { "pretest": "tsc --incremental -p tests/tsconfig.json", "test": "playwright test -c tests-out" }}The pretest script runs typescript on the tests. test will run the tests that have been generated to the tests-out directory. The -c argument configures the test runner to look for tests inside the tests-out directory. Then npm run test will build the tests and run them.


Transpilation issues Using dynamic imports inside a function passed to various evaluate() methods is not supported. This is because Playwright uses Function.prototype.toString() to serialize functions, and transpiler will sometimes replace dynamic imports with require() calls, which are not valid inside the web page. To work around this issue, use a string template instead of a function:



```
await page.evaluate(`async () => { const { value } = await import('some-module'); console.log(value);});
```


```

UI ModeUI Mode let's you explore, run and debug tests with a time travel experience complete with watch mode. All test files are loaded into the testing sidebar where you can expand each file and describe block to individually run, view, watch and debug

each test. Filter tests by text or @tag or by passed, failed and skipped tests as well as by projects as set in your playwright.config file. See a full trace of your tests and hover back and forward over each action to see what was happening during each step and pop out the DOM snapshot to a separate window for a better debugging experience.

Running tests in UI Mode

To open UI mode, run the following command:
`npx playwright test --ui`

Filtering tests

Filter tests by text or @tag or by passed, failed or skipped tests. You can also filter by projects as set in your playwright.config file. If you are using project dependencies make sure to run your setup tests first before running the tests that depend on them. The UI mode will not take into consideration the setup tests and therefore you will have to manually run them first.

Running your tests

Once you launch UI Mode you will see a list of all your test files. You can run all your tests by clicking the triangle icon in the sidebar. You can also run a single test file, a block of tests or a single test by hovering over the name and clicking on the triangle next to it.

Viewing test traces

Traces are shown for each test that has been run, so to see the trace, click on one of the test names. Note that you won't see any trace results if you click on the name of the test file or the name of a describe block.

Actions and metadata

In the Actions tab you can see what locator was used for every action and how long each one took to run. Hover over each action of your test and visually see the change in the DOM snapshot. Go back and forward in time and click an action to inspect and debug. Use the Before and After tabs to visually see what happened before and after the action. Next to the Actions tab you will find the Metadata tab which will show you more information on your test such as the Browser, viewport size, test duration and more.

Source, console, log and network

As you hover over each action of your test the source code for the test is highlighted below. Click on the source tab to see the source code for the entire test. Click on the console tab to see the console logs for each action. Click on the log tab to see the logs for each action. Click on the network tab to see the network logs for each action.

Attachments

The "Attachments" tab allows you to explore attachments. If you're doing visual regression testing, you'll be able to compare screenshots by examining the image diff, the actual image and the expected image. When you click on the expected image you can use the slider to slide one image over the other so you can easily see the differences in your screenshots.

Pop out and inspect the DOM

Pop out the DOM snapshot into it's own window for a better debugging experience by clicking on the pop out icon above the DOM snapshot. From there you can open the browser DevTools and inspect the HTML, CSS, Console etc. Go back to UI Mode and click on another action and pop that one out to easily compare the two side by side or debug each individually.

Timeline view

At the top of the trace you can see a timeline view of each action of your test. Hover back and forth to see an image snapshot for each action.

Pick locator

Click on the pick locator button and hover over the DOM snapshot to see the locator for each element highlighted as you hover. Click on an element to save the locator into the pick locator field. You can then copy the locator and paste it into your test.

Watch mode

Next to the name of each test in the sidebar you will find an eye icon. Clicking on the icon will activate watch mode which will re-run the test when you make changes to it. You can watch a number of tests at the same time by clicking the eye icon next to each one or all tests by clicking the eye icon at the top of the sidebar. If you are using VS Code then you can easily open your test by clicking on the file icon next to the eye icon. This will open your test in VS Code right at the line

of code that you clicked on. Docker & GitHub Codespaces For Docker and GitHub Codespaces environments, you can run UI mode in the browser. In order for an endpoint to be accessible outside of the container, it needs to be bound to the 0.0.0.0 interface: `npx playwright test --ui-host=0.0.0.0` In the case of GitHub Codespaces, the port gets forwarded automatically, so you can open UI mode in the browser by clicking on the link in the terminal. To have a static port, you can pass the `--ui-port` flag: `npx playwright test --ui-port=8080 --ui-host=0.0.0.0` Be aware that when specifying the `--ui-host=0.0.0.0` flag, UI Mode with your traces, the passwords and secrets is accessible from other machines inside your network. In the case of GitHub Codespaces, the ports are only accessible from your account by default.

CI GitHub Actions When installing Playwright you are given the option to add a GitHub Actions. This creates a `playwright.yml` file inside a `.github/workflows` folder containing everything you need so that your tests run on each push and pull request into the main/master branch. What you will learn: GitHub Actions Create a Repo and Push to GitHub Opening the Workflows Viewing Test Logs HTML Report Downloading the HTML Report Viewing the HTML Report Viewing the Trace What's Next GitHub Actions Tests will run on push or pull request on branches main/master. The workflow will install all dependencies, install Playwright and then run the tests. It will also create the HTML report.

```
name: Playwright Test
on: push
  branches: [main, master]
pull_request:
  branches: [main, master]
jobs:
  test:
    timeout-minutes: 60
    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 Browsers
        run: npx playwright install --with-deps
      - name: Run Playwright tests
        run: npx playwright test
      - uses: actions/upload-artifact@v3
        if: always()
        with:
          name: playwright-report
          path: playwright-report/
          retention-days: 30
```

Create a Repo and Push to GitHub Create a repo on GitHub and create a new repository or push an existing repository. Follow the instructions on GitHub and don't forget to initialize a git repository using the `git init` command so you can add, commit and push your code. Opening the Workflows Click on the Actions tab to see the workflows. Here you will see if your tests have passed or failed. On Pull Requests you can also click on the Details link in the PR status check. Viewing Test Logs Clicking on the workflow run will show you the all the actions that GitHub performed and clicking on Run Playwright tests will show the error messages, what was expected and what was received as well as the call log. HTML Report The HTML Report shows you a full report of your tests. You can filter the report by browsers, passed tests, failed tests, skipped tests and flaky tests. Downloading the HTML Report In the Artifacts section click on the `playwright-report` to download your report in the format of a zip file. Viewing the HTML Report Locally opening the report will not work as expected as you need a web server in order for everything to work correctly. First, extract the zip, preferably in a folder that already has Playwright installed. Using the command line change into the directory where the report is and use `npx playwright show-report` followed by the name of the extracted folder. This will serve up the report and enable you to view it in your browser. `npx playwright show-report name-of-my-extracted-playwright-report` To learn more about reports check out our detailed guide on HTML Reporter Viewing the Trace Once you have served the report using `npx playwright show-report`, click on the trace icon next to the test's file name as seen in the image above.

You can then view the trace of your tests and inspect each action to try to find out why the tests are failing. To learn more about traces check out our detailed guide on [Trace Viewer](#). To learn more about running tests on CI check out our detailed guide on [Continuous Integration](#). What's Next [Learn how to use Locators](#) [Learn how to perform Actions](#) [Learn how to write Assertions](#)

Getting started - VS Code Playwright Test was created specifically to accommodate the needs of end-to-end testing. Playwright supports all modern rendering engines including Chromium, WebKit, and Firefox. Test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation of Google Chrome for Android and Mobile Safari. Get started by installing Playwright and generating a test to see it in action. Alternatively you can also get started and run your tests using the CLI. **Installation** Install the VS Code extension from the marketplace or from the extensions tab in VS Code. Once installed, open the command panel and type: `Install Playwright`

Select Test: Install Playwright and Choose the browsers you would like to run your tests on. These can be later configured in the `playwright.config` file. You can also choose if you would like to have a GitHub Actions setup to run your tests on CI. **Running Tests** You can run a single test by clicking the green triangle next to your test block to run your test. Playwright will run through each line of the test and when it finishes you will see a green tick next to your test block as well as the time it took to run the test. **Run Tests and Show Browsers** You can also run your tests and show the browsers by selecting the option `Show Browsers` in the testing sidebar. Then when you click the green triangle to run your test the browser will open and you will visually see it run through your test. Leave this selected if you want browsers open for all your tests or uncheck it if you prefer your tests to run in headless mode with no browser open. Use the `Close all browsers` button to close all browsers. **View and Run All Tests** View all tests in the testing sidebar and extend the tests by clicking on each test. Tests that have not been run will not have the green check next to them. Run all tests by clicking on the white triangle as you hover over the tests in the testing sidebar. **Run Tests on Specific Browsers** The VS Code test runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown and choose another profile or modify the default profile by clicking `Select Default Profile` and select the browsers you wish to run your tests on. Choose various or all profiles to run tests on multiple profiles. These profiles are read from the `playwright.config` file. To add more profiles such as a mobile profile, first add it to your config file and it will then be available here. **Debugging Tests** With the VS Code extension you can debug your tests right in VS Code see error messages, create breakpoints and live debug your tests. **Error Messages** If your test fails VS Code will show you error messages right in the editor showing what was expected, what was received as well as a complete call log. **Live Debugging** You can debug your test live in VS Code. After running a test with the `Show Browser` option checked, click on any of the locators in VS Code and it will be highlighted in the Browser window. Playwright will highlight it if it exists and show you if there is more than one result. You can also edit the locators in VS Code and Playwright will show you the changes live in the browser window. **Run in Debug Mode** To set a breakpoint click next to the line number where you want the breakpoint to be until a red dot appears. Run the tests in debug mode by right clicking on the line next to the test you want to run. A browser window will open and the test will run and pause at where

the breakpoint is set. You can step through the tests, pause the test and rerun the tests from the menu in VS Code.

Debug in different Browsers

By default debugging is done using the Chromium profile. You can debug your tests on different browsers by right clicking on the debug icon in the testing sidebar and clicking on the 'Select Default Profile' option from the dropdown. Then choose the test profile you would like to use for debugging your tests. Each time you run your test in debug mode it will use the profile you selected. You can run tests in debug mode by right clicking the line number where your test is and selecting 'Debug Test' from the menu. To learn more about debugging, see [Debugging in Visual Studio Code](#).

Generating Tests

CodeGen will auto generate your tests for you as you perform actions in the browser and is a great way to quickly get started. The viewport for the browser window is set to a specific width and height. See the [configuration guide](#) to change the viewport or emulate different environments.

Record a New Test

To record a test click on the Record new button from the Testing sidebar. This will create a test-1.spec.ts file as well as open up a browser window. In the browser go to the URL you wish to test and start clicking around. Playwright will record your actions and generate a test for you. Once you are done recording click the cancel button or close the browser window. You can then inspect your test-1.spec.ts file and see your generated test. Your browser does not support the video tag.

Record at Cursor

To record from a specific point in your test file click the Record at cursor button from the Testing sidebar. This generates actions into the existing test at the current cursor position. You can run the test, position the cursor at the end of the test and continue generating the test.

Picking a Locator

Pick a locator and copy it into your test file by clicking the Pick locator button from the testing sidebar. Then in the browser click the element you require and it will now show up in the Pick locator box in VS Code. Press 'enter' on your keyboard to copy the locator into the clipboard and then paste anywhere in your code. Or press 'escape' if you want to cancel. Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If the generator finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.

What's next

Write tests using web first assertions, page fixtures and locators

See test reports

See a trace of your tests

Release notes

Version 1.36

Summer maintenance release

Browser Versions

Chromium 115.0.5790.75
Mozilla Firefox 115.0
WebKit 17.0

This version was also tested against the following stable channels: Google Chrome 114, Microsoft Edge 114

Version 1.35 Highlights

UI mode is now available in VSCode Playwright extension via a new "Show trace viewer" button:

- UI mode and trace viewer mark network requests handled with `page.route()` and `browserContext.route()` handlers, as well as those issued via the API
- testing: New option `maskColor` for methods `page.screenshot()`, `locator.screenshot()`, `expect(page).toHaveScreenshot()` and `expect(locator).toHaveScreenshot()` to change default masking color: `await page.goto('https://playwright.dev');`
`await expect(page).toHaveScreenshot({ mask: [page.locator('img')], maskColor: '#00FF00', // green});`
- New uninstall CLI command to uninstall browser binaries: `$ npx playwright uninstall` # remove browsers installed by this installation
`$ npx playwright uninstall --all` # remove all ever-install Playwright browsers
- Both UI mode and trace viewer now could be opened in a browser tab: `$ npx playwright test --ui-port 0` # open UI mode in a tab on a random port
`$ npx playwright show-trace --port 0` # open trace

viewer in tab on a random port & p Breaking changes playwright-core binary got renamed from playwright to playwright-core. So if you use playwright-core CLI, make sure to update the name: \$ npx playwright-core install # the new way to install browsers when using playwright-core This change does not affect @playwright/test and playwright package users.

Browser Versions Chromium 115.0.5790.13 Mozilla Firefox 113.0 WebKit 16.4 This version was also tested against the following stable channels: Google Chrome 114 Microsoft Edge 114 Version 1.34 Highlights UI Mode now shows steps, fixtures and attachments: New property testProject.teardown to specify a project that needs to run after this and all dependent projects have finished. Teardown is useful to cleanup any resources acquired by this project. A common pattern would be a setup dependency with a corresponding teardown:

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  projects: [
    { name: 'setup', testMatch: /global.setup\.ts/, teardown: 'teardown', },
    { name: 'teardown', testMatch: /global.teardown\.ts/, },
    { name: 'chromium', use: devices['Desktop Chrome'], dependencies: ['setup'], },
    { name: 'firefox', use: devices['Desktop Firefox'], dependencies: ['setup'], },
    { name: 'webkit', use: devices['Desktop Safari'], dependencies: ['setup'], },
  ],
});

```

New method expect.configure to create pre-configured expect instance with its own defaults such as timeout and soft.

```

const slowExpect = expect.configure({ timeout: 10000 });
await slowExpect(locator).toHaveText('Submit');
// Always do soft assertions
const softExpect = expect.configure({ soft: true });

```

New options stderr and stdout in testConfig.webServer to configure output handling:

```

playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Run your local dev server before starting the tests
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
    stdout: 'pipe',
    stderr: 'pipe',
  },
});

```

New locator.and() to create a locator that matches both locators.

```

const button = page.getByRole('button').and(page.getByTitle('Subscribe'));

```

New events browserContext.on('console') and browserContext.on('dialog') to subscribe to any dialogs and console messages from any page from the given browser context. Use the new methods consoleMessage.page() and dialog.page() to pin-point event source.

& p Breaking changes npx playwright test no longer works if you install both playwright and @playwright/test. There's no need to install both, since you can always import browser automation APIs from @playwright/test directly:

```

automation.ts
import { chromium, firefox, webkit } from '@playwright/test';
// ...

```

Node.js 14 is no longer supported since it reached its end-of-life on April 30, 2023.

Browser Versions Chromium 114.0.5735.26 Mozilla Firefox 113.0 WebKit 16.4 This version was also tested against the following stable channels: Google Chrome 113 Microsoft Edge 113 Version 1.33

Locators Update Use locator.or() to create a locator that matches either of the two locators. Consider a scenario where you'd like to click on a "New email" button, but sometimes a security settings dialog shows up instead. In this case, you can wait for either a "New email" button, or a dialog and act accordingly:

```

const newEmail = page.getByRole('button', { name: 'New email' });
const dialog = page.getByText('Confirm security settings');
await expect(newEmail.or(dialog)).toBeVisible();
if (await dialog.isVisible()) {
  await page.getByRole('button', { name: 'Dismiss' }).click();
}
await newEmail.click();

```

Use new options hasNot and hasNotText in locator.filter() to find elements that do not match certain conditions.

```

const rowLocator =

```

`page.locator('tr');await rowLocator .filter({ hasNotText: 'text in column 1' }) .filter({ hasNot: page.getByRole('button', { name: 'column 2 button' }) }) .screenshot();`Use new web-first assertion `expect(locator).toBeAttached()` to ensure that the element is present in the page's DOM. Do not confuse with the `expect(locator).toBeVisible()` that ensures that element is both attached & visible.New APIs `locator.or()`New option `hasNot` in `locator.filter()`New option `hasNotText` in `locator.filter()``expect(locator).toBeAttached()`New option `timeout` in `route.fetch()``reporter.onExit()`& p Breaking change The `mcr.microsoft.com/playwright:v1.33.0` now serves a Playwright image based on Ubuntu Jammy. To use the focal-based image, please use `mcr.microsoft.com/playwright:v1.33.0-focal` instead.Browser Versions Chromium 113.0.5672.53Mozilla Firefox 112.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 112Microsoft Edge 112Version 1.32 Introducing UI Mode (preview) New UI Mode lets you explore, run and debug tests. Comes with a built-in watch mode.Engage with a new flag `--ui:npx playwright test --ui`New APIs New options `updateMode` and `updateContent` in `page.routeFromHAR()` and `browserContext.routeFromHAR()`.Chaining existing locator objects, see locator docs for details.New property `testInfo.testId`.New option `name` in `method tracing.startChunk()`.& p Breaking change in component tests Note: component tests only, does not affect end-to-end tests.`@playwright/experimental-ct-react` now supports React 18 only.If you're running component tests with React 16 or 17, please replace `@playwright/experimental-ct-react` with `@playwright/experimental-ct-react17`.Browser Versions Chromium 112.0.5615.29Mozilla Firefox 111.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 111Microsoft Edge 111Version 1.31 New APIs New property `testProject.dependencies` to configure dependencies between projects.Using dependencies allows global setup to produce traces and other artifacts, see the setup steps in the test report and `more.playwright.config.ts``import { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: /global.setup\.ts/, }, { name: 'chromium', use: devices['Desktop Chrome'], dependencies: ['setup'], }, { name: 'firefox', use: devices['Desktop Firefox'], dependencies: ['setup'], }, { name: 'webkit', use: devices['Desktop Safari'], dependencies: ['setup'], },],});`New assertion `expect(locator).toBeInViewport()` ensures that locator points to an element that intersects viewport, according to the intersection observer API.`const button = page.getByRole('button');``// Make sure at least some part of element intersects viewport.await expect(button).toBeInViewport();``// Make sure element is fully outside of viewport.await expect(button).not.toBeInViewport();``// Make sure that at least half of the element intersects viewport.await expect(button).toBeInViewport({ ratio: 0.5 });`Miscellaneous DOM snapshots in trace viewer can be now opened in a separate window.New method `defineConfig` to be used in `playwright.config`.New option `Route.fetch.maxRedirects` for method `route.fetch()`.Playwright now supports Debian 11 arm64.Official docker images now include Node 18 instead of Node 16.& p Breaking change in component tests Note: component tests only, does not affect end-to-end tests.`playwright-ct.config` configuration file for component testing now requires calling `defineConfig`.`// Beforeimport { type PlaywrightTestConfig, devices } from '@playwright/experimental-ct-react';const config: PlaywrightTestConfig = { // ... config goes here ...};export default config;`Replace config variable definition with `defineConfig` call:`//`

Afterimport { defineConfig, devices } from '@playwright/experimental-ct-react';export default defineConfig({ // ... config goes here ...});Browser Versions Chromium 111.0.5563.19Mozilla Firefox 109.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 110Microsoft Edge 110Version 1.30 Browser Versions Chromium 110.0.5481.38Mozilla Firefox 108.0.2WebKit 16.4This version was also tested against the following stable channels:Google Chrome 109Microsoft Edge 109Version 1.29 New APIs New method route.fetch() and new option json for route.fulfill():await page.route('**/api/settings', async route => { // Fetch original settings. const response = await route.fetch(); // Force settings theme to a predefined value. const json = await response.json(); json.theme = 'Solorized'; // Fulfill with modified data. await route.fulfill({ json });});New method locator.all() to iterate over all matching elements:// Check all checkboxes!const checkboxes = page.getByRole('checkbox');for (const checkbox of await checkboxes.all()) await checkbox.check();locator.selectOption() matches now by value or label:<select multiple> <option value="red">Red</div> <option value="green">Green</div> <option value="blue">Blue</div></select>await element.selectOption('Red');Retry blocks of code until all assertions pass:await expect(async () => { const response = await page.request.get('https://api.example.com'); await expect(response).toBeOK();}).toPass();Read more in our documentation.

Automatically capture full page screenshot on test failure:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { screenshot: { mode: 'only-on-failure', fullPage: true, } }});Miscellaneous Playwright Test now respects jsconfig.json.New options args and proxy for androidDevice.launchBrowser().Option postData in method route.continue() now supports Serializable values.Browser Versions Chromium 109.0.5414.46Mozilla Firefox 107.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 108Microsoft Edge 108Version 1.28 Playwright Tools Record at Cursor in VSCode. You can run the test, position the cursor at the end of the test and continue generating the test.Live Locators in VSCode. You can hover and edit locators in VSCode to get them highlighted in the opened browser.Live Locators in CodeGen. Generate a locator for any element on the page using "Explore" tool.Codegen and Trace Viewer Dark Theme. Automatically picked up from operating system settings.Test Runner Configure retries and test timeout for a file or a test with test.describe.configure().// Each test in the file will be retried twice and have a timeout of 20 seconds.test.describe.configure({ retries: 2, timeout: 20_000 });test('runs first', async ({ page }) => {});test('runs second', async ({ page }) => {});Use testProject.snapshotPathTemplate and testConfig.snapshotPathTemplate to configure a template controlling location of snapshots generated by expect(page).toHaveScreenshot() and expect(snapshot).toMatchSnapshot().playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ testDir: './tests', snapshotPathTemplate: '{testDir}/__screenshots__/{testFilePath}/{arg}{ext}',});New APIs locator.blur()locator.clear()android.launchServer() and android.connect()androidDevice.on('close')Browser Versions Chromium 108.0.5359.29Mozilla Firefox 106.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 107Microsoft Edge 107Version 1.27 Locators With these new APIs writing locators is a joy:page.getByText() to locate by text

`content.page.getByRole()` to locate by ARIA role, ARIA attributes and accessible name.
`page.getByLabel()` to locate a form control by associated label's text.
`page.getByTestId()` to locate an element based on its data-testid attribute (other attribute can be configured).
`page.getByPlaceholder()` to locate an input by placeholder.
`page.getByAltText()` to locate an element, usually image, by its text alternative.
`page.getByTitle()` to locate an element by its title.
`await page.getByLabel('User Name').fill('John');`
`await page.getByLabel('Password').fill('secret-password');`
`await page.getByRole('button', { name: 'Sign in' }).click();`
`await expect(page.getByText('Welcome, John!')).toBeVisible();`
All the same methods are also available on `Locator`, `FrameLocator` and `Frame` classes.

Other highlights `workers` option in the `playwright.config.ts` now accepts a percentage string to use some of the available CPUs. You can also pass it in the command line: `npx playwright test --workers=20%`
New options `host` and `port` for the `html` reporter.
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ reporter: [['html', { host: 'localhost', port: '9223' }]],});`
New field `FullConfig.configFile` is available to test reporters, specifying the path to the config file if any.

As announced in v1.25, Ubuntu 18 will not be supported as of Dec 2022. In addition to that, there will be no WebKit updates on Ubuntu 18 starting from the next Playwright release.

Behavior Changes `expect(locator).toHaveAttribute()` with an empty value does not match missing attribute anymore. For example, the following snippet will succeed when button does not have a `disabled` attribute.
`await expect(page.getByRole('button')).toHaveAttribute('disabled', '');`
Command line options `--grep` and `--grep-invert` previously incorrectly ignored `grep` and `grepInvert` options specified in the config. Now all of them are applied together.

Browser Versions
Chromium 107.0.5304.18
Mozilla Firefox 105.0.1
WebKit 16.0
This version was also tested against the following stable channels:

Google Chrome 106
Microsoft Edge 106
Version 1.26

Assertions `New option` `enabled` for `expect(locator).toBeEnabled()`.
`expect(locator).toHaveText()` now pierces open shadow roots.
New option `editable` for `expect(locator).toBeEditable()`.
New option `visible` for `expect(locator).toBeVisible()`.

Other highlights `New option` `maxRedirects` for `apiRequestContext.get()` and others to limit redirect count.
New command-line flag `--pass-with-no-tests` that allows the test suite to pass when no files are found.
New command-line flag `--ignore-snapshots` to skip snapshot expectations, such as `expect(value).toMatchSnapshot()` and `expect(page).toHaveScreenshot()`.

Behavior Change A bunch of Playwright APIs already support the `waitUntil: 'domcontentloaded'` option. For example: `await page.goto('https://playwright.dev', { waitUntil: 'domcontentloaded', });`
Prior to 1.26, this would wait for all iframes to fire the `DOMContentLoaded` event. To align with web specification, the `'domcontentloaded'` value only waits for the target frame to fire the `'DOMContentLoaded'` event. Use `waitUntil: 'load'` to wait for all iframes.

Browser Versions **Chromium** 106.0.5249.30
Mozilla Firefox 104.0
WebKit 16.0
This version was also tested against the following stable channels:

Google Chrome 105
Microsoft Edge 105
Version 1.25

VSCoDe Extension Watch your tests running live & keep devtools open.
Pick selector. Record new test from current page state.
Test Runner `test.step()` now returns the value of the step function: `test('should work', async ({ page }) => { const pageTitle = await test.step('get title', async () => { await page.goto('https://playwright.dev'); return await page.title(); }); console.log(pageTitle); });`
Added `test.describe.fixme()`.
New 'interrupted' test

status.Enable tracing via CLI flag: `npx playwright test --trace=on`. Announcements

- We now ship Ubuntu 22.04 Jammy Jellyfish docker image: [mcr.microsoft.com/playwright:v1.34.0-jammy](https://mcr.microsoft.com/playwright/v1.34.0-jammy)
- This is the last release with macOS 10.15 support (deprecated as of 1.21).
- This is the last release with Node.js 12 support, we recommend upgrading to Node.js LTS (16).
- Ubuntu 18 is now deprecated and will not be supported as of Dec 2022.

Browser Versions Chromium 105.0.5195.19 Mozilla Firefox 103.0 WebKit 16.0 This version was also tested against the following stable channels: Google Chrome 104 Microsoft Edge 104 Version 1.24

Multiple Web Servers in playwright.config.ts

Launch multiple web servers, databases, or other processes by passing an array of configurations:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  webServer: [
    {
      command: 'npm run start',
      url: 'http://127.0.0.1:3000',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
    {
      command: 'npm run backend',
      url: 'http://127.0.0.1:3333',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
  ],
  use: {
    baseURL: 'http://localhost:3000/',
  },
});
```

Debian 11 Bullseye Support

Playwright now supports Debian 11 Bullseye on x86_64 for Chromium, Firefox and WebKit. Let us know if you encounter any issues! Linux support looks like this: Ubuntu 20.04 Ubuntu 22.04 Debian 11 Chromium WebKit Firefox

Anonymous Describe

It is now possible to call `test.describe()` to create suites without a title. This is useful for giving a group of tests a common option with `test.use()`.

```
test.describe(() => {
  test.use({ colorScheme: 'dark' });
  test('one', async ({ page }) => {
    // ...
  });
  test('two', async ({ page }) => {
    // ...
  });
});
```

Component Tests Update

Playwright 1.24 Component Tests introduce `beforeMount` and `afterMount` hooks. Use these to configure your app for tests. For example, this could be used to setup App router in Vue.js:

```
src/component.spec.ts
import { test } from '@playwright/experimental-ct-vue';
import { Component } from './mycomponent';
test('should work', async ({ mount }) => {
  const component = await mount(Component, {
    hooksConfig: {
      /* anything to configure your app */
    },
  });
  playwright/index.ts
import { router } from './router';
import { beforeMount } from '@playwright/experimental-ct-vue/hooks';
beforeMount(async ({ app, hooksConfig }) => {
  app.use(router);
});
```

A similar configuration in Next.js would look like this:

```
src/component.spec.jsx
import { test } from '@playwright/experimental-ct-react';
import { Component } from './mycomponent';
test('should work', async ({ mount }) => {
  const component = await mount(
    <Component></Component>,
    {
      // Pass mock value from test into `beforeMount`.
      hooksConfig: {
        router: {
          query: { page: 1, per_page: 10 },
          asPath: '/posts'
        }
      },
    },
  );
  playwright/index.js
import router from 'next/router';
import { beforeMount } from '@playwright/experimental-ct-react/hooks';
beforeMount(async ({ hooksConfig }) => {
  // Before mount, redefine useRouter to return mock value from test.
  router.useRouter = () => hooksConfig.router;
});
```

Version 1.23 Network Replay

Now you can record network traffic into a HAR file and re-use this traffic in your tests. To record network into HAR file: `npx playwright open --save-har=github.har.zip` <https://github.com/microsoft>

Alternatively, you can record HAR programmatically:

```
const context = await browser.newContext({
  recordHar: {
    path: 'github.har.zip',
  },
});
// ... do stuff ...
await context.close();
```

Use the new methods `page.routeFromHAR()` or `browserContext.routeFromHAR()` to serve matching responses from the HAR file: `await context.routeFromHAR('github.har.zip');`

Read more in our documentation.

Advanced Routing

You can now use `route.fallback()` to defer

routing to other handlers. Consider the following example: // Remove a header from all requests.

```
test.beforeEach(async ({ page }) => {
  await page.route('**/*', async route => {
    const headers = await route.request().allHeaders();
    delete headers['if-none-match'];
    route.fallback({ headers });
  });
});
test('should work', async ({ page }) => {
  await page.route('**/*', route => {
    if (route.request().resourceType() === 'image')
      route.abort();
    else
      route.fallback();
  });
});
```

Note that the new methods `page.routeFromHAR()` and `browserContext.routeFromHAR()` also participate in routing and could be deferred to.

Web-First Assertions Update

New method `expect(locator).toHaveValues()` that asserts all selected values of `<select multiple>` element. Methods `expect(locator).toContainText()` and `expect(locator).toHaveText()` now accept `ignoreCase` option.

Component Tests Update

Support for Vue2 via the `@playwright/experimental-ct-vue2` package. Support for component tests for create-react-app with components in .js files. Read more about component testing with Playwright.

Miscellaneous

If there's a service worker that's in your way, you can now easily disable it with a new context option `serviceWorkers: playwright.config.ts`

```
export default {
  use: {
    serviceWorkers: 'block',
  }
};
```

Using .zip path for `recordHar` context option automatically zips the resulting HAR:

```
const context = await browser.newContext({
  recordHar: {
    path: 'github.har.zip',
  }
});
```

If you intend to edit HAR by hand, consider using the "minimal" HAR recording mode that only records information that is essential for replaying:

```
const context = await browser.newContext({
  recordHar: {
    path: 'github.har',
    mode: 'minimal',
  }
});
```

Playwright now runs on Ubuntu 22 amd64 and Ubuntu 22 arm64. We also publish new docker image `mcr.microsoft.com/playwright:v1.34.0-jammy`.

Breaking Changes

`WebServer` is now considered "ready" if request to the specified url has any of the following HTTP status codes: 200-299, 300-399 (new), 400, 401, 402, 403 (new).

Version 1.22 Highlights

Components Testing (preview)

Playwright Test can now test your React, Vue.js or Svelte components. You can use all the features of Playwright Test (such as parallelization, emulation & debugging) while running components in real browsers. Here is what a typical component test looks like:

```
App.spec.ts
import { test, expect } from '@playwright/experimental-ct-react';
import App from './App'; // Let's test component in a dark scheme
test.use({ colorScheme: 'dark' });
test('should render', async ({ mount }) => {
  const component = await mount(<App>); // As with any Playwright test, assert locator text.
  await expect(component).toContainText('React'); // Or do a screenshot
  await expect(component).toHaveScreenshot(); // Or use any Playwright method
  await component.click();
});
```

Read more in our documentation.

Role selectors

that allow selecting elements by their ARIA role, ARIA attributes and accessible name.

```
// Click a button with accessible name "log in"
await page.locator('role=button[name="log in"]').click();
```

Read more in our documentation.

New `locator.filter()` API

to filter an existing locator

```
const buttons = page.locator('role=button'); // ...
const submitButton = buttons.filter({ hasText: 'Submit' });
await submitButton.click();
```

New web-first assertions

`expect(page).toHaveScreenshot()` and `expect(locator).toHaveScreenshot()` that wait for screenshot stabilization and enhances test reliability. The new assertions has screenshot-specific defaults, such as:

- disables animations
- uses CSS scale option

```
await page.goto('https://playwright.dev');
await expect(page).toHaveScreenshot();
```

The new `expect(page).toHaveScreenshot()` saves screenshots at the same location as

`expect(snapshot).toMatchSnapshot().`Version 1.21 Highlights New role selectors that allow selecting elements by their ARIA role, ARIA attributes and accessible name.// Click a button with accessible name "log in"`await page.locator('role=button[name="log in"]').click();`Read more in our documentation.New scale option in `page.screenshot()` for smaller sized screenshots.New caret option in `page.screenshot()` to control text caret. Defaults to "hide".New method `expect.poll` to wait for an arbitrary condition:// Poll the method until it returns an expected result.`await expect.poll(async () => { const response = await page.request.get('https://api.example.com'); return response.status();}).toBe(200);``expect.poll` supports most synchronous matchers, like `.toBe()`, `.toContain()`, etc. Read more in our documentation.Behavior Changes ESM support when running TypeScript tests is now enabled by default. The `PLAYWRIGHT_EXPERIMENTAL_TS_ESM` env variable is no longer required.The `mcr.microsoft.com/playwright` docker image no longer contains Python. Please use `mcr.microsoft.com/playwright/python` as a Playwright-ready docker image with pre-installed Python.Playwright now supports large file uploads (100s of MBs) via `locator.setInputFiles()` API.Browser Versions Chromium 101.0.4951.26Mozilla Firefox 98.0.2WebKit 15.4This version was also tested against the following stable channels:Google Chrome 100Microsoft Edge 100Version 1.20 Highlights New options for methods `page.screenshot()`, `locator.screenshot()` and `elementHandle.screenshot()`:Option `animations`: "disabled" rewinds all CSS animations and transitions to a consistent stateOption `mask`: `Locator[]` masks given elements, overlaying them with pink #FF00FF boxes.`expect().toMatchSnapshot()` now supports anonymous snapshots: when snapshot name is missing, Playwright Test will generate one automatically:`expect('Web is Awesome <3').toMatchSnapshot();`New `maxDiffPixels` and `maxDiffPixelRatio` options for fine-grained screenshot comparison using `expect().toMatchSnapshot()`:`expect(await page.screenshot()).toMatchSnapshot({ maxDiffPixels: 27, // allow no more than 27 different pixels.});`It is most convenient to specify `maxDiffPixels` or `maxDiffPixelRatio` once in `testConfig.expect`.Playwright Test now adds `testConfig.fullyParallel` mode. By default, Playwright Test parallelizes between files. In fully parallel mode, tests inside a single file are also run in parallel. You can also use `--fully-parallel` command line flag.`playwright.config.ts``export default { fullyParallel: true,};``testProject.grep` and `testProject.grepInvert` are now configurable per project. For example, you can now configure smoke tests project using `grep`:`playwright.config.ts``export default { projects: [{ name: 'smoke tests', grep: /@smoke/, },],};`Trace Viewer now shows API testing requests.`locator.highlight()` visually reveals element(s) for easier debugging.Announcements We now ship a designated Python docker image `mcr.microsoft.com/playwright/python`. Please switch over to it if you use Python. This is the last release that includes Python inside our javascript `mcr.microsoft.com/playwright` docker image.v1.20 is the last release to receive WebKit update for macOS 10.15 Catalina. Please update MacOS to keep using latest & greatest WebKit!Browser Versions Chromium 101.0.4921.0Mozilla Firefox 97.0.1WebKit 15.4This version was also tested against the following stable channels:Google Chrome 99Microsoft Edge 99Version 1.19 Playwright Test Update Playwright Test v1.19 now 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.locator('#status')).toHaveText('Success');`
`await expect.soft(page.locator('#eta')).toHaveText('1 day');`
 // ... and continue the test to check more things.
`await page.locator('#next-page').click();`
`await expect.soft(page.locator('#title')).toHaveText('Make another order');`
 Read more in our documentation
 You can now specify a custom error message as a second argument to the `expect` and `expect.soft` functions, for example:
`await expect(page.locator('text=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',
    |    ^  5 | });
    |    6 |
```

 Read more in our documentation
 By default, tests in a single file are run in order. If you have many independent tests in a single file, you can now run them in parallel with `test.describe.configure()`.
 Other Updates
 Locator now supports a `has` option that makes sure it contains another locator inside:
`await page.locator('article', { has: page.locator('.highlight') }).click();`
 Read more in locator documentation
 New `locator.page()`, `page.screenshot()` and `locator.screenshot()` now automatically hide blinking caret
 Playwright Codegen now generates locators and frame locators
 New option `url` in `testConfig.webServer` to ensure your web server is ready before running the tests
 New `testInfo.errors` and `testResult.errors` that contain all failed assertions and soft assertions.
 Potentially breaking change in Playwright Test Global Setup
 It is unlikely that this change will affect you, no action is required if your tests keep running as they did.
 We've noticed that in rare cases, the set of tests to be executed was configured in the global setup by means of the environment variables. We also noticed some applications that were post processing the reporters' output in the global teardown. If you are doing one of the two, learn more
 Browser Versions
 Chromium 100.0.4863.0
 Mozilla Firefox 96.0.1
 WebKit 15.4
 This version was also tested against the following stable channels: Google Chrome 98, Microsoft Edge 98, Version 1.18
 Locator Improvements
`locator.dragTo()`
`expect(locator).toBeChecked({ checked })`
 Each locator can now be optionally filtered by the text it contains:
`await page.locator('li', { hasText: 'my item' }).locator('button').click();`
 Read more in locator documentation
 Testing API improvements
`expect(response).toBeOK()`
`testInfo.attach()`
`test.info()`
 Improved TypeScript Support
 Playwright Test now respects `tsconfig.json`'s `baseUrl` and `paths`, so you can use aliases
 There is a new environment variable `PW_EXPERIMENTAL_TS_ESM` that allows importing ESM modules in your TS code, without the need for the compile step. Don't forget the `.js` suffix when you are importing your esm modules. Run your tests as follows:
`npm i --save-dev @playwright/test@1.18.0-rc1`
`PW_EXPERIMENTAL_TS_ESM=1 npx playwright test`
 Create Playwright
 The `npm init playwright` command is now generally available for your use:
 # Run from your project's root directory
`npm init playwright@latest`
 # Or create a new project
`npm init playwright@latest new-project`
 This will create a Playwright Test configuration file, optionally add examples, a GitHub Action workflow and a first test example `spec.ts`.
 New APIs & changes
 new `testCase.repeatEachIndex`
 API `acceptDownloads` option now defaults to `true`
 Breaking change: custom config options
 Custom config options are a convenient way to parametrize projects with different values. Learn more in this guide.
 Previously, any fixture introduced through `test.extend()` could be overridden in the

testProject.use config section. For example, **// WRONG: THIS SNIPPET DOES NOT WORK SINCE v1.18.**

```
// fixtures.js
const test = base.extend({ myParameter: 'default' });
// playwright.config.js
module.exports = { use: { myParameter: 'value', }, };

```

The proper way to make a fixture parametrized in the config file is to specify `option: true` when defining the fixture. For example, **// CORRECT: THIS SNIPPET WORKS SINCE v1.18.**

```
// fixtures.js
const test = base.extend({ // Fixtures marked as "option: true" will get a value specified in the config, // or fallback to the default value.
  myParameter: ['default', { option: true }],
});
// playwright.config.js
module.exports = { use: { myParameter: 'value', }, };

```

Browser Versions Chromium 99.0.4812.0 Mozilla Firefox 95.0 WebKit 15.4 This version was also tested against the following stable channels: Google Chrome 97 Microsoft Edge 97 Version 1.17

Frame Locators Playwright 1.17 introduces frame locators - a locator to the iframe on the page. Frame locators capture the logic sufficient to retrieve the iframe and then locate elements in that iframe. Frame locators are strict by default, will wait for iframe to appear and can be used in Web-First assertions. Frame locators can be created with either `page.frameLocator()` or `locator.frameLocator()` method.

```
const locator = page.frameLocator('#my-iframe').locator('text=Submit');
await locator.click();

```

Read more at our documentation.

Trace Viewer Update Playwright Trace Viewer is now available online at <https://trace.playwright.dev>! Just drag-and-drop your trace.zip file to inspect its contents. **NOTE:** trace files are not uploaded anywhere; trace.playwright.dev is a progressive web application that processes traces locally.

Playwright Test traces now include sources by default (these could be turned off with tracing option). Trace Viewer now shows test name.

New trace metadata tab with browser details.

Snapshots now have URL bar.

HTML Report Update HTML report now supports dynamic filtering.

Report is now a single static HTML file that could be sent by e-mail or as a slack attachment.

Ubuntu ARM64 support + more Playwright now supports Ubuntu 20.04 ARM64. You can now run Playwright tests inside Docker on Apple M1 and on Raspberry Pi. You can now use Playwright to install stable version of Edge on Linux: `npx playwright install msedge`.

New APIs Tracing now supports a 'title' option.

Page navigations support a new 'commit' waiting option.

HTML reporter got new configuration option `testConfig.snapshotDir`.

option testInfo.parallelIndex **testInfo.titlePath** **testOptions.trace** has new option `expect.toMatchSnapshot` supports subdirectories.

reporter.printsToStdio() Version 1.16

Playwright Test API Testing Playwright 1.16 introduces new API Testing that lets you send requests to the server directly from Node.js! Now you can:

- test your server
- prepare server side state before visiting the web application in a test
- validate server side post-conditions after running some actions in the browser

To do a request on behalf of Playwright's Page, use new `page.request` API:

```
import { test, expect } from '@playwright/test';
test('context fetch', async ({ page }) => { // Do a GET request on behalf of page
  const response = await page.request.get('http://example.com/foo.json'); // ...
});

```

To do a stand-alone request from node.js to an API endpoint, use new `request` fixture:

```
import { test, expect } from '@playwright/test';
test('context fetch', async ({ request }) => { // Do a GET request on behalf of page
  const response = await request.get('http://example.com/foo.json'); // ...
});

```

Read more about it in our API testing guide.

Response Interception It is now possible to do response interception by combining API Testing with request interception. For example, we can blur all the images on the page:

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

```

```
'jimp'; // image processing library
test('response interception', async ({ page }) => {
  await page.route('**/*.jpeg', async route => {
    const response = await page._request.fetch(route.request());
    const image = await jimp.read(await response.body());
    await image.blur(5);
    route.fulfill({ response, body: await image.getBufferAsync('image/jpeg') });
  });
  const response = await page.goto('https://playwright.dev');
  expect(response.status()).toBe(200);
});
```

Read more about response interception. New HTML reporter Try it out new HTML reporter with either --reporter=html or a reporter entry in playwright.config.ts file: \$ npx playwright test --reporter=html

The HTML reporter has all the information about tests and their failures, including surfacing trace and image artifacts. Read more about our reporters.

Playwright Library

`locator.waitFor()` Wait for a locator to resolve to a single element with a given state. Defaults to the state: 'visible'. Comes especially handy when working with lists:

```
import { test, expect } from '@playwright/test';
test('context fetch', async ({ page }) => {
  const completeness = page.locator('text=Success');
  await completeness.waitFor();
  expect(await page.screenshot()).toMatchSnapshot('screen.png');
});
```

Read more about `locator.waitFor()`.

Docker support for Arm64

Playwright Docker image is now published for Arm64 so it can be used on Apple Silicon. Read more about Docker integration.

Playwright Trace Viewer

web-first assertions inside trace viewer

run trace viewer with `npx playwright show-trace` and drop trace files to the trace viewer

PWA API testing is integrated with trace viewer

better visual attribution of action targets

Read more about Trace Viewer.

Browser Versions

Chromium 97.0.4666.0 Mozilla Firefox 93.0 WebKit 15.4

This version of Playwright was also tested against the following stable channels:

Google Chrome 94 Microsoft Edge 94 Version 1.15

Playwright Library

`Page.mouse.wheel()` By using `Page.mouse.wheel()` you are now able to scroll vertically or horizontally.

New Headers API

Previously it was not possible to get multiple header values of a response. This is now possible and additional helper functions are available:

```
Request.allHeaders()
Request.headersArray()
Request.headerValue(name: string)
Response.allHeaders()
Response.headersArray()
Response.headerValue(name: string)
Response.headerValues(name: string)
```

Forced-Colors emulation

Its now possible to emulate the forced-colors CSS media feature by passing it in the context options or calling `Page.emulateMedia()`.

New APIs

`Page.route()` accepts new `times` option to specify how many times this route should be matched.

`Page.setChecked(selector: string, checked: boolean)` and `Locator.setChecked(selector: string, checked: boolean)` was introduced to set the checked state of a checkbox.

`Request.sizes()` Returns resource size information for given http request.

`BrowserContext.tracing.startChunk()` - Start a new trace chunk.

`BrowserContext.tracing.stopChunk()` - Stops a new trace chunk.

Playwright Test

`test.parallel()` run tests in the same file in parallel

```
test.describe.parallel('group', () => {
  test('runs in parallel 1', async ({ page }) => { });
  test('runs in parallel 2', async ({ page }) => { });
});
```

By default, tests in a single file are run in order. If you have many independent tests in a single file, you can now run them in parallel with `test.describe.parallel(title, callback)`.

Add --debug CLI flag

By using `npx playwright test --debug` it will enable the Playwright Inspector for you to debug your tests.

Browser Versions

Chromium 96.0.4641.0 Mozilla Firefox 92.0 WebKit 15.0 Version 1.14

Playwright Library

New "strict" mode Selector ambiguity is a common problem in automation testing. "strict" mode ensures that your selector points to a single element and throws otherwise. Pass `strict: true` into your action calls to opt in. // This will throw if

you have more than one button!await page.click('button', { strict: true });

New Locators API

Locator represents a view to the element(s) on the page. It captures the logic sufficient to retrieve the element at any given moment. The difference between the Locator and ElementHandle is that the latter points to a particular element, while Locator captures the logic of how to retrieve that element. Also, locators are "strict" by default!const locator = page.locator('button');await locator.click(); Learn more in the documentation.

Experimental React and Vue selector engines

React and Vue selectors allow selecting elements by its component name and/or property values. The syntax is very similar to attribute selectors and supports all attribute selector operators.await page.locator('_react=SubmitButton[enabled=true]').click();await page.locator('_vue=submit-button[enabled=true]').click(); Learn more in the react selectors documentation and the vue selectors documentation.

New nth and visible selector engines

nth selector engine is equivalent to the :nth-match pseudo class, but could be combined with other selector engines. visible selector engine is equivalent to the :visible pseudo class, but could be combined with other selector engines.// select the first button among all buttonsawait button.click('button >> nth=0');// or if you are using locators, you can use first(), nth() and last()await page.locator('button').first().click();// click a visible buttonawait button.click('button >> visible=true');

Playwright Test - Web-First Assertions

expect now supports lots of new web-first assertions. Consider the following example:await expect(page.locator('.status')).toHaveText('Submitted'); Playwright Test will be re-testing the node with the selector .status until fetched Node has the "Submitted" text. It will be re-fetching the node and checking 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 testProject.expect value in test config. By default, the timeout for assertions is not set, so it'll wait forever, until the whole test times out. List of all new assertions:expect(locator).toBeChecked()expect(locator).toBeDisabled()expect(locator).toBeEditable()expect(locator).toBeEmpty()expect(locator).toBeEnabled()expect(locator).toBeFocused()expect(locator).toBeHidden()expect(locator).toBeVisible()expect(locator).toContainText(text, options?)expect(locator).toHaveAttribute(name, value)expect(locator).toHaveClass(expected)expect(locator).toHaveCount(count)expect(locator).toHaveCSS(name, value)expect(locator).toHaveId(id)expect(locator).toHaveJSProperty(name, value)expect(locator).toHaveText(expected, options)expect(page).toHaveTitle(title)expect(page).toHaveURL(url)expect(locator).toHaveValue(value)

Serial mode with describe.serial

Declares a group of tests that should always be run serially. If one of the tests fails, all subsequent tests are skipped. All tests in a group are retried together.test.describe.serial('group', () => { test('runs first', async ({ page }) => { /* ... */ }); test('runs second', async ({ page }) => { /* ... */ });}); Learn more in the documentation.

Steps API with test.step

Split long tests into multiple steps using test.step() API:import { test, expect } from '@playwright/test';test('test', async ({ page }) => { await test.step('Log in', async () => { // ... }); await test.step('news feed', async () => { // ... });});

Step information is exposed in reporters API.

Launch web server before running tests

To launch a server during the tests, use the webServer option in the configuration file. The server will wait for a given url to be available before running the tests, and the url will be passed over to Playwright as a baseUrl when creating a context.playwright.config.tsimport { defineConfig } from '@playwright/test';export default

```

defineConfig({ webServer: { command: 'npm run start', // command to launch url:
'http://127.0.0.1:3000', // url to await for timeout: 120 * 1000, reuseExistingServer: !
process.env.CI, },});

```

Learn more in the documentation.

Browser Versions

Chromium	Firefox	WebKit
94.0.4595.0	91.0	15.0

Version 1.13

Playwright Test – Introducing Reporter API which is already used to create an Allure Playwright reporter.

– New baseURL fixture to support relative paths in tests.

Playwright – Programmatic drag-and-drop support via the page.dragAndDrop() API.

– Enhanced HAR with body sizes for requests and responses. Use via recordHar option in browser.newContext().

Tools

Playwright Trace Viewer now shows parameters, returned values and console.log() calls.

Playwright Inspector can generate Playwright Test tests.

New and Overhauled Guides

- Intro
- Authentication
- Chrome Extensions
- Playwright Test Annotations
- Playwright Test Configuration
- Playwright Test Fixtures

Browser Versions

Chromium	Firefox	WebKit
93.0.4576.0	90.0	14.2

New Playwright APIs

- new baseURL option in browser.newContext() and browser.newPage()
- response.securityDetails() and response.serverAddr()
- page.dragAndDrop() and frame.dragAndDrop()
- download.cancel()
- page.inputValue(), frame.inputValue() and elementHandle.inputValue()
- new force option in page.fill(), frame.fill(), and elementHandle.fill()
- new force option in page.selectOption(), frame.selectOption(), and elementHandle.selectOption()

Version 1.12 – Introducing Playwright Test

Playwright Test is a new test runner built from scratch by Playwright team specifically to accommodate end-to-end testing needs:

- Run tests across all browsers.
- Execute tests in parallel.
- Enjoy context isolation and sensible defaults out of the box.
- Capture videos, screenshots and other artifacts on failure.
- Integrate your POMs as extensible fixtures.

Installation:

```

npm i -D @playwright/test

```

```

test tests/foo.spec.ts:
import { test, expect } from '@playwright/test';
test('basic test', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  const name = await page.innerText('.navbar__title');
  expect(name).toBe('Playwright');
});

```

Running:

```

npx playwright test

```

– Read more in Playwright Test documentation.

– Introducing Playwright Trace Viewer

Playwright Trace Viewer is a new GUI tool that helps exploring recorded Playwright traces after the script ran. Playwright traces let you examine:

- page DOM before and after each Playwright action
- page rendering before and after each Playwright action
- browser network during script execution

Traces are recorded using the new browserContext.tracing API:

```

const browser = await chromium.launch();
const context = await browser.newContext();
// Start tracing before creating / navigating a page.
await context.tracing.start({ screenshots: true, snapshots: true });
const page = await context.newPage();
await page.goto('https://playwright.dev');
// Stop tracing and export it into a zip archive.
await context.tracing.stop({ path: 'trace.zip' });

```

Traces are examined later with the Playwright CLI:

```

npx playwright show-trace trace.zip

```

That will open the following GUI:

– Read more in trace viewer documentation.

Browser Versions

Chromium	Firefox	WebKit
93.0.4530.0	89.0	14.2

This version of Playwright was also tested against the following stable channels:

- Google Chrome 91
- Microsoft Edge 91

New APIs

- reducedMotion option in page.emulateMedia(), browserType.launchPersistentContext(), browser.newContext() and browser.newPage()
- browserContext.on('request') browserContext.on('requestfailed') browserContext.on('requestfinished') browserContext.on('response')
- tracesDir option in browserType.launch() and browserType.launchPersistentContext()
- new browserContext.tracing API
- namespace
- new download.page() method

Version 1.11 – Read more in

New video: Playwright: A New Test Automation Framework for the Modern Web (slides) We talked about Playwright Showed engineering work behind the scenes Did live demos with new features ' (Special thanks to applitools for hosting the event and inviting us! Browser Versions Chromium 92.0.4498.0 Mozilla Firefox 89.0b6 WebKit 14.2 New APIs support for async predicates across the API in methods such as `page.waitForRequest()` and others new emulation devices: Galaxy S8, Galaxy S9+, Galaxy Tab S4, Pixel 3, Pixel 4 new methods: `page.waitForURL()` to await navigations to URL `video.delete()` and `video.saveAs()` to manage screen recording new options: `screen` option in the `browser.newContext()` method to emulate window `screen dimensions` `position` option in `page.check()` and `page.uncheck()` method `strict` option to dry-run actions in `page.check()`, `page.uncheck()`, `page.click()`, `page.dblclick()`, `page.hover()` and `page.tap()` Version 1.10 Playwright for Java v1.10 is now stable! Run Playwright against Google Chrome and Microsoft Edge stable channels with the new channels API. Chromium screenshots are fast on Mac & Windows. Bundled Browser Versions Chromium 90.0.4430.0 Mozilla Firefox 87.0b10 WebKit 14.2 This version of Playwright was also tested against the following stable channels: Google Chrome 89 Microsoft Edge 89 New APIs `browserType.launch()` now accepts the new 'channel' option. Read more in our documentation. Version 1.9 Playwright Inspector is a new GUI tool to author and debug your tests. Line-by-line debugging of your Playwright scripts, with play, pause and step-through. Author new scripts by recording user actions. Generate element selectors for your script by hovering over elements. Set the `PWDEBUG=1` environment variable to launch the Inspector. Pause script execution with `page.pause()` in headed mode. Pausing the page launches Playwright Inspector for debugging. New has-text pseudo-class for CSS selectors. `:has-text("example")` matches any element containing "example" somewhere inside, possibly in a child or a descendant element. See more examples. Page dialogs are now auto-dismissed during execution, unless a listener for dialog event is configured. Learn more about this. Playwright for Python is now stable with an idiomatic snake case API and pre-built Docker image to run tests in CI/CD. Browser Versions Chromium 90.0.4421.0 Mozilla Firefox 86.0b10 WebKit 14.1 New APIs `page.pause()`. Version 1.8 Selecting elements based on layout with `:left-of()`, `:right-of()`, `:above()` and `:below()`. Playwright now includes command line interface, former `playwright-cli.npx` `playwright --help` `page.selectOption()` now waits for the options to be present. New methods to assert element state like `page.isEditable()`. New APIs `elementHandle.isChecked()`, `elementHandle.isDisabled()`, `elementHandle.isEditable()`, `elementHandle.isEnabled()`, `elementHandle.isHidden()`, `elementHandle.isVisible()`, `page.isChecked()`, `page.isDisabled()`, `page.isEditable()`, `page.isEnabled()`, `page.isHidden()`, `page.isVisible()`. New option 'editable' in `elementHandle.waitForElementState()`. Browser Versions Chromium 90.0.4392.0 Mozilla Firefox 85.0b5 WebKit 14.1 Version 1.7 New Java SDK: Playwright for Java is now on par with JavaScript, Python and .NET bindings. Browser storage API: New convenience APIs to save and load browser storage state (cookies, local storage) to simplify automation scenarios with authentication. New CSS selectors: We heard your feedback for more flexible selectors and have revamped the selectors implementation. Playwright 1.7 introduces new CSS extensions and there's more coming soon. New website: The docs website at playwright.dev has been updated and is now built with Docusaurus. Support for Apple Silicon: Playwright browser binaries for WebKit and Chromium are now built for Apple Silicon. New APIs `browserContext.storageState()` to

get current state for later reuse.storageState option in browser.newContext() and browser.newPage() to setup browser context state.Browser Versions Chromium 89.0.4344.0Mozilla Firefox 84.0b9WebKit 14.1

Canary releasesPlaywright for Node.js has a canary releases system.It permits you to test new unreleased features instead of waiting for a full release. They get released daily on the next NPM tag of Playwright.It is a good way to give feedback to maintainers, ensuring the newly implemented feature works as intended.note Using a canary release in production might seem risky, but in practice, it's not. A canary release passes all automated tests and is used to test e.g. the HTML report, Trace Viewer, or Playwright Inspector with end-to-end tests. npm install -D @playwright/test@nextNext npm Dist Tag For any code-related commit on main, the continuous integration will publish a daily canary release under the @next npm dist tag.You can see on npm the current dist tags:latest: stable releasesnext: next releases, published dailybeta: after a release-branch was cut, usually a week before a stable release each commit gets published under this tagUsing a Canary Release npm install -D @playwright/test@nextDocumentation The stable and the next documentation is published on playwright.dev. To see the next documentation, press Shift on the keyboard 5 times.

Test configurationPlaywright has many options to configure how your tests are run. You can specify these options in the configuration file. Note that test runner options are top-level, do not put them into the use section.Basic Configuration Here are some of the most common configuration options.import { defineConfig, devices } from '@playwright/test';export default defineConfig({ // Look for test files in the "tests" directory, relative to this configuration file. testDir: 'tests', // Run all tests in parallel. fullyParallel: true, // Fail the build on CI if you accidentally left test.only in the source code. forbidOnly: !!process.env.CI, // Retry on CI only. retries: process.env.CI ? 2 : 0, // Opt out of parallel tests on CI. workers: process.env.CI ? 1 : undefined, // Reporter to use reporter: 'html', use: { // Base URL to use in actions like `await page.goto('/')`. baseURL: 'http://127.0.0.1:3000', // Collect trace when retrying the failed test. trace: 'on-first-retry', }, // Configure projects for major browsers. projects: [{ name: 'chromium', use: { ...devices['Desktop Chrome'] }, },], // Run your local dev server before starting the tests. webServer: { command: 'npm run start', url: 'http://127.0.0.1:3000', reuseExistingServer: !process.env.CI, }, });OptionDescriptiontestConfig.forbidOnlyWhether to exit with an error if any tests are marked as test.only. Useful on CI.testConfig.fullyParallelhave all tests in all files to run in parallel. See /Parallelism and sharding for more details.testConfig.projectsRun tests in multiple configurations or on multiple browserstestConfig.reporterReporter to use. See Test Reporters to learn more about which reporters are available.testConfig.retriesThe maximum number of retry attempts per test. See Test Retries to learn more about retries.testConfig.testDirDirectory with the test files.testConfig.useOptions with use{}testConfig.webServerTo launch a server during the tests, use the webServer optiontestConfig.workersThe maximum number of concurrent worker processes to use for parallelizing tests. Can also be set as percentage of logical CPU cores, e.g. '50%'. See /Parallelism and sharding for more details.Filtering Tests Filter tests by glob patterns or regular expressions.playwright.config.tsimport { defineConfig } from '@playwright/test';export

default defineConfig({ // Glob patterns or regular expressions to ignore test files. testIgnore: 'test-assets', // Glob patterns or regular expressions that match test files. testMatch: 'todo-tests/*.spec.ts',});OptionDescriptiontestConfig.testIgnoreGlob patterns or regular expressions that should be ignored when looking for the test files. For example, 'test-assets'testConfig.testMatchGlob patterns or regular expressions that match test files. For example, 'todo-tests/*.spec.ts'. By default, Playwright runs .*(test|spec).(js|ts|mjs) files.Advanced Configuration playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Folder for test artifacts such as screenshots, videos, traces, etc. outputDir: 'test-results', // path to the global setup files. globalSetup: require.resolve('./global-setup'), // path to the global teardown files. globalTeardown: require.resolve('./global-teardown'), // Each test is given 30 seconds. timeout: 30000,});OptionDescriptiontestConfig.globalSetupPath to the global setup file. This file will be required and run before all the tests. It must export a single function.testConfig.globalTeardownPath to the global teardown file. This file will be required and run after all the tests. It must export a single function.testConfig.outputDirFolder for test artifacts such as screenshots, videos, traces, etc.testConfig.timeoutPlaywright enforces a timeout for each test, 30 seconds by default. Time spent by the test function, fixtures, beforeEach and afterEach hooks is included in the test timeout.Expect Options Configuration for the expect assertion library.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ expect: { // Maximum time expect() should wait for the condition to be met. timeout: 5000, toHaveScreenshot: { // An acceptable amount of pixels that could be different, unset by default. maxDiffPixels: 10, }, toMatchSnapshot: { // An acceptable ratio of pixels that are different to the total amount of pixels, between 0 and 1. maxDiffPixelRatio: 0.1, }, },});OptionDescriptiontestConfig.expectWeb first assertions like expect(locator).toHaveText() have a separate timeout of 5 seconds by default. This is the maximum time the expect() should wait for the condition to be met. Learn more about test and expect timeouts and how to set them for a single test.testConfig.expect(page).toHaveScreenshot()Configuration for the expect(locator).toHaveScreenshot() method.testConfig.expect(snapshot).toMatchSnapshot()Configuration for the expect(locator).toMatchSnapshot() method.Add custom matchers using expect.extend You can extend Playwright assertions by providing custom matchers. These matchers will be available on the expect object.In this example we add a custom toBeWithinRange function in the configuration file. Custom matcher should return a message callback and a pass flag indicating whether the assertion passed.TypeScriptJavaScriptplaywright.config.tsimport { expect, defineConfig } from '@playwright/test';expect.extend({ toBeWithinRange(received: number, floor: number, ceiling: number) { const pass = received >= floor && received <= ceiling; if (pass) { return { message: () => 'passed', pass: true, }; } else { return { message: () => 'failed', pass: false, }; } },});export default defineConfig({});playwright.config.tsconst { expect, defineConfig } = require('@playwright/test');expect.extend({ toBeWithinRange(received, floor, ceiling) { const pass = received >= floor && received <= ceiling; if (pass) { return { message: () => 'passed', pass: true, }; } else { return { message: () => 'failed', pass: false, }; } },});module.exports = defineConfig({});Now we

can use `toBeWithinRange` in the test.
`example.spec.ts`
`import { test, expect } from '@playwright/test';`
`test('numeric ranges', () => { expect(100).toBeWithinRange(90, 110); expect(101).not.toBeWithinRange(0, 100); });`
 Do not confuse Playwright's `expect` with the `expect` library. The latter is not fully integrated with Playwright test runner, so make sure to use Playwright's own `expect`.
 For TypeScript, also add the following to your `global.d.ts`. If it does not exist, you need to create it inside your repository. Make sure that your `global.d.ts` gets included inside your `tsconfig.json` via the `include` or `compilerOptions.typeRoots` option so that your IDE will pick it up. You don't need it for JavaScript.
`global.d.ts`
`export {};`
`declare global { namespace PlaywrightTest { interface Matchers<R, T> { toBeWithinRange(a: number, b: number): R; } }}`

Test use options
 In addition to configuring the test runner you can also configure Emulation, Network and Recording for the Browser or BrowserContext. These options are passed to the `use: {}` object in the Playwright config.
Basic Options
 Set the base URL and storage state for all tests:
`playwright.config.ts`
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ use: { // Base URL to use in actions like `await page.goto('/')`. baseUrl: 'http://127.0.0.1:3000', // Populates context with given storage state. storageState: 'state.json', } });`

OptionDescription
`testOptions.baseUrl` Base URL used for all pages in the context. Allows navigating by using just the path, for example `page.goto('/settings')`.
`testOptions.storageState` Populates context with given storage state. Useful for easy authentication, learn more.
Emulation Options
 With Playwright you can emulate a real device such as a mobile phone or tablet. See our guide on projects for more info on emulating devices. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme". See our Emulation guide to learn more.

`playwright.config.ts`
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ use: { // Emulates `prefers-colors-scheme` media feature. colorScheme: 'dark', // Context geolocation. geolocation: { longitude: 12.492507, latitude: 41.889938 }, // Emulates the user locale. locale: 'en-GB', // Grants specified permissions to the browser context. permissions: ['geolocation'], // Emulates the user timezone. timezoneId: 'Europe/Paris', // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, } });`

OptionDescription
`testOptions.colorScheme` Emulates 'prefers-colors-scheme' media feature, supported values are 'light', 'dark', 'no-preference'.
`testOptions.geolocation` Context geolocation.
`testOptions.locale` Emulates the user locale, for example en-GB, de-DE, etc.
`testOptions.permissions` A list of permissions to grant to all pages in the context.
`testOptions.timezoneId` Changes the timezone of the context.
`testOptions.viewport` Viewport used for all pages in the context.

Network Options
 Available options to configure networking:
`playwright.config.ts`
`import { defineConfig } from '@playwright/test';`
`export default defineConfig({ use: { // Whether to automatically download all the attachments. acceptDownloads: false, // An object containing additional HTTP headers to be sent with every request. extraHTTPHeaders: { 'X-My-Header': 'value', }, // Credentials for HTTP authentication. httpCredentials: { username: 'user', password: 'pass', }, // Whether to ignore HTTPS errors during`

navigation. ignoreHTTPSErrors: true, // Whether to emulate network being offline.
 offline: true, // Proxy settings used for all pages in the test. proxy: { server: 'http://
 myproxy.com:3128', bypass:
 'localhost', }, },});OptionDescriptiontestOptions.acceptDownloadsWhether to
 automatically download all the attachments, defaults to true. Learn more about working
 with downloads.testOptions.extraHTTPHeadersAn object containing additional HTTP
 headers to be sent with every request. All header values must be
 strings.testOptions.httpCredentialsCredentials for HTTP
 authentication.testOptions.ignoreHTTPSErrorsWhether to ignore HTTPS errors during
 navigation.testOptions.offlineWhether to emulate network being
 offline.testOptions.proxyProxy settings used for all pages in the test.noteYou don't have
 to configure anything to mock network requests. Just define a custom Route that mocks
 the network for a browser context. See our network mocking guide to learn
 more.Recording Options With Playwright you can capture screenshots, record videos as
 well as traces of your test. By default these are turned off but you can enable them by
 setting the screenshot, video and trace options in your playwright.config.js file. Trace
 files, screenshots and videos will appear in the test output directory, typically test-
 results.playwright.config.tsimport { defineConfig } from '@playwright/test';export default
 defineConfig({ use: { // Capture screenshot after each test failure. screenshot: 'only-
 on-failure', // Record trace only when retrying a test for the first time. trace: 'on-first-
 retry', // Record video only when retrying a test for the first time. video: 'on-first-
 retry' },});OptionDescriptiontestOptions.screenshotCapture screenshots of your test.
 Options include 'off', 'on' and 'only-on-failure'testOptions.tracePlaywright can produce
 test traces while running the tests. Later on, you can view the trace and get detailed
 information about Playwright execution by opening Trace Viewer. Options include: 'off',
 'on', 'retain-on-failure' and 'on-first-retry'testOptions.videoPlaywright can record videos
 for your tests. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'Other
 Options playwright.config.tsimport { defineConfig } from '@playwright/test';export default
 defineConfig({ use: { // Maximum time each action such as `click()` can take. Defaults
 to 0 (no limit). actionTimeout: 0, // Name of the browser that runs tests. For example
 `chromium`, `firefox`, `webkit`. browserName: 'chromium', // Toggles bypassing
 Content-Security-Policy. bypassCSP: true, // Channel to use, for example "chrome",
 "chrome-beta", "msedge", "msedge-beta". channel: 'chrome', // Run browser in
 headless mode. headless: false, // Change the default data-testid attribute.
 testIdAttribute: 'pw-test-id', },});OptionDescriptiontestOptions.actionTimeoutTimeout for
 each Playwright action in milliseconds. Defaults to 0 (no timeout). Learn more about
 timeouts and how to set them for a single test.testOptions.browserNameName of the
 browser that runs tests. Defaults to 'chromium'. Options include chromium, firefox, or
 webkit.testOptions.bypassCSPToggles bypassing Content-Security-Policy. Useful when
 CSP includes the production origin. Defaults to false.testOptions.channelBrowser
 channel to use. Learn more about different browsers and
 channels.testOptions.headlessWhether to run the browser in headless mode meaning
 no browser is shown when running tests. Defaults to
 true.testOptions.testIdAttributeChanges the default data-testid attribute used by
 Playwright locators.More browser and context options Any options accepted by
 browserType.launch() or browser.newContext() can be put into launchOptions or

contextOptions respectively in the use section.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    launchOptions: {
      slowMo: 50,
    },
  },
});
```

However, most common ones like headless or viewport are available directly in the use section - see basic options, emulation or network.

Explicit Context Creation and Option Inheritance

If using the built-in browser fixture, calling `browser.newContext()` will create a context with options inherited from the config:

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    userAgent: 'some custom ua',
    viewport: { width: 100, height: 100 },
  },
});
```

An example test illustrating the initial context options are set:

```
import { test, expect } from '@playwright/test';
test('should inherit use options on context when using built-in browser fixture', async ({ browser }) => {
  const context = await browser.newContext();
  const page = await context.newPage();
  expect(await page.evaluate(() => navigator.userAgent)).toBe('some custom ua');
  expect(await page.evaluate(() => window.innerWidth)).toBe(100);
  await context.close();
});
```

Configuration Scopes

You can configure Playwright globally, per project, or per test. For example, you can set the locale to be used globally by adding `locale` to the `use` option of the Playwright config, and then override it for a specific project using the `project` option in the config. You can also override it for a specific test by adding `test.use({})` in the test file and passing in the options.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    locale: 'en-GB',
  },
});
```

You can override options for a specific project using the `project` option in the Playwright config:

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'], locale: 'de-DE' },
    },
  ],
});
```

You can override options for a specific test file by using the `test.use()` method and passing in the options. For example to run tests with the French locale for a specific test:

```
import { test, expect } from '@playwright/test';
test.use({ locale: 'fr-FR' });
test('example', async ({ page }) => {
  // ...
});
```

The same works inside a describe block. For example to run tests in a describe block with the French locale:

```
import { test, expect } from '@playwright/test';
test.describe('french language block', () => {
  test.use({ locale: 'fr-FR' });
  test('example', async ({ page }) => {
    // ...
  });
});
```

Web server Playwright comes with a webserver option in the config file which gives you the ability to launch a local dev server before running your tests. This is ideal for when writing your tests during development and when you don't have a staging or production url to test against.

Configuring a web server

Use the webserver property in your Playwright config to launch a development web server during the tests.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Run your local dev server before starting the tests
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
    stdout: 'ignore',
    stderr: 'pipe',
  },
});
```

testConfig.webServer Launch a development web server (or multiple) during the tests.

command Shell command to start the local dev server of your app.

url URL of your http server that is expected to return a 2xx, 3xx, 400, 401, 402, or 403 status code when the server is ready to accept connections.

reuseExistingServer If true, it will re-use an existing server on the url when available. If no server is running on that url, it will run the command to start a new server. If false, it will throw if an existing process is listening on the url. To see the stdout, you can set the `DEBUG=pw:webserver` environment variable.

stdout If "pipe", it will pipe the stdout of the command to the process stdout. If "ignore", it will ignore the stdout of the command. Default to "ignore".

stderr Whether to pipe the stderr of the command to the process stderr or ignore it. Defaults to "pipe".

Adding a server timeout

Webservers can sometimes take longer to boot up. In this case, you can increase the timeout to wait for the server to start.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Rest of your config...
  // Run your local dev server before starting the tests
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
    timeout: 120 * 1000,
  },
});
```

Adding a base URL

It is also recommended to specify the base URL in the `use: {}` section of your config, so that tests can use relative urls and you don't have to specify the full URL over and over again. When using `page.goto()`, `page.route()`, `page.waitForURL()`, `page.waitForRequest()`, or `page.waitForResponse()` it takes the base URL in consideration by using the `URL()` constructor for building the corresponding URL. For Example, by setting the base URL to `http://127.0.0.1:3000` and navigating to `/login` in your tests, Playwright will run the test using `http://127.0.0.1:3000/login`.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  // Rest of your config...
  // Run your local dev server before starting the tests
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
    use: {
      baseURL: 'http://127.0.0.1:3000',
    },
  },
});
```

Now you can use a relative path when navigating the page:

```
test.spec.ts
import { test } from '@playwright/test';
test('test', async ({ page }) => {
  // This will navigate to http://127.0.0.1:3000/login
  await page.goto('/login');
});
```

Multiple web servers

Multiple web servers (or background processes) can be launched simultaneously by providing an array of webServer configurations. See `testConfig.webServer` for more info.

```
playwright.config.ts
import { defineConfig } from '@playwright/test';
export default defineConfig({
  webServer: [
    {
      command: 'npm run start',
      url: 'http://127.0.0.1:3000',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
    {
      command: 'npm run backend',
      url: 'http://127.0.0.1:3333',
      timeout: 120 * 1000,
      reuseExistingServer: !process.env.CI,
    },
  ],
});
```

```
process.env.CI, } ], use: { baseUrl: 'http://127.0.0.1:3000', });
```

Library Playwright Library provides unified APIs for launching and interacting with browsers, while Playwright Test provides all this plus a fully managed end-to-end Test Runner and experience. Under most circumstances, for end-to-end testing, you'll want to use `@playwright/test` (Playwright Test), and not `playwright` (Playwright Library) directly. To get started with Playwright Test, follow the Getting Started Guide.

Differences when using library

Library Example

The following is an example of using the Playwright Library directly to launch Chromium, go to a page, and check its title:

```
TypeScriptJavaScriptimport { chromium, devices } from 'playwright';import assert from 'node:assert';(async () => { // Setup  const browser = await chromium.launch();  const context = await browser.newContext(devices['iPhone 11']);  const page = await context.newPage(); // The actual interesting bit  await context.route('*.jpg', route => route.abort());  await page.goto('https://example.com/');  assert(await page.title() === 'Example Domain'); // Ø=ÜN not a Web First assertion // Teardown  await context.close();  await browser.close();})();const assert = require('node:assert');const { chromium, devices } = require('playwright');(async () => { // Setup  const browser = await chromium.launch();  const context = await browser.newContext(devices['iPhone 11']);  const page = await context.newPage(); // The actual interesting bit  await context.route('*.jpg', route => route.abort());  await page.goto('https://example.com/');  assert(await page.title() === 'Example Domain'); // Ø=ÜN not a Web First assertion // Teardown  await context.close();  await browser.close();})();
```

Run it with `node my-script.js`.

Test Example

A test to achieve similar behavior, would look like:

```
TypeScriptJavaScriptimport { expect, test, devices } from '@playwright/test';test.use(devices['iPhone 11']);test('should be titled', async ({ page, context }) => {  await context.route('*.jpg', route => route.abort());  await page.goto('https://example.com/');  await expect(page).toHaveTitle('Example');});const { expect, test, devices } = require('@playwright/test');test.use(devices['iPhone 11']);test('should be titled', async ({ page, context }) => {  await context.route('*.jpg', route => route.abort());  await page.goto('https://example.com/');  await expect(page).toHaveTitle('Example');});
```

Run it with `npx playwright test`.

Key Differences

The key differences to note are as follows:

Library Test Installation

```
npm install playwright
npm init playwright@latest
```

- note `install` vs. `init`

`install` browsers Chromium, Firefox, WebKit are installed by default

```
npx playwright install
```

or

```
npx playwright install chromium
```

for a single one

```
import { chromium } from 'playwright';
```

test Initialization

Explicitly need to:

- Pick a browser to use, e.g. `chromium`
- Launch browser with `browserType.launch()`
- Create a context with `browser.newContext()`, and pass any context options explicitly, e.g. `devices['iPhone 11']`
- Create a page with `browserContext.newPage()`

An isolated page and context are provided to each test out-of-the-box, along with other built-in fixtures. No explicit creation. If referenced by the test in its arguments, the Test Runner will create them for the test. (i.e. lazy-initialization)

Assertions

No built-in Web-First Assertions

Web-First assertions like:

```
expect(page).toHaveTitle()
expect(page).toHaveScreenshot()
```

which auto-wait and retry for the condition to be met.

Cleanup

Explicitly need to:

- Close context with `browserContext.close()`
- Close browser with `browser.close()`

No explicit close of built-in fixtures; the Test Runner will take care of it.

Running

When using the Library, you run the code as a node script, possibly with some compilation first. When using the Test

Runner, you use the `npm run playwright test` command. Along with your config, the Test Runner handles any compilation and choosing what to run and how to run it. In addition to the above, Playwright Test, as a full-featured Test Runner, includes: Configuration Matrix and Projects: In the above example, in the Playwright Library version, if we wanted to run with a different device or browser, we'd have to modify the script and plumb the information through. With Playwright Test, we can just specify the matrix of configurations in one place, and it will create run the one test under each of these configurations. Parallelization Web-First Assertions Reporting Retries Easily Enabled Tracing and more... Usage Use npm or Yarn to install Playwright library in your Node.js project. See system requirements. `npm install -D playwright` This single command downloads the Playwright NPM package and browser binaries for Chromium, Firefox and WebKit. To modify this behavior see managing browsers. Once installed, you can require Playwright in a Node.js script, and launch any of the 3 browsers (chromium, firefox and webkit). `const { chromium } = require('playwright'); (async () => { const browser = await chromium.launch(); // Create pages, interact with UI elements, assert values await browser.close(); })();` Playwright APIs are asynchronous and return Promise objects. Our code examples use the `async/await` pattern to ease readability. The code is wrapped in an unnamed `async` arrow function which is invoking itself. `(async () => { // Start of async arrow function // Function code // ... })();` // End of the function and `()` to invoke itself First script In our first script, we will navigate to `https://playwright.dev/` and take a screenshot in WebKit. `const { webkit } = require('playwright'); (async () => { const browser = await webkit.launch(); const page = await browser.newPage(); await page.goto('https://playwright.dev/'); await page.screenshot({ path: `example.png` }); await browser.close(); })();` By default, Playwright runs the browsers in headless mode. To see the browser UI, pass the `headless: false` flag while launching the browser. You can also use `slowMo` to slow down execution. Learn more in the debugging tools section. `firefox.launch({ headless: false, slowMo: 50 });` Record scripts Command line tools can be used to record user interactions and generate JavaScript code. `npm run playwright codegen wikipedia.org` TypeScript support Playwright includes built-in support for TypeScript. Type definitions will be imported automatically. It is recommended to use type-checking to improve the IDE experience. In JavaScript Add the following to the top of your JavaScript file to get type-checking in VS Code or WebStorm. `// @ts-check // ...` Alternatively, you can use JSDoc to set types for variables. `/** @type {import('playwright').Page} */ let page;` In TypeScript TypeScript support will work out-of-the-box. Types can also be imported explicitly. `let page: import('playwright').Page;` Accessibility testing Playwright can be used to test your application for many types of accessibility issues. A few examples of problems this can catch include: Text that would be hard to read for users with vision impairments due to poor color contrast with the background behind it UI controls and form elements without labels that a screen reader could identify Interactive elements with duplicate IDs which can confuse assistive technologies The following examples rely on the `@axe-core/playwright` package which adds support for running the axe accessibility testing engine as part of your Playwright tests. Disclaimer Automated accessibility tests can detect some common accessibility problems such as missing or invalid properties. But many accessibility problems can only be discovered through manual testing. We recommend using a combination of automated testing, manual accessibility assessments, and inclusive user testing. For

manual assessments, we recommend Accessibility Insights for Web, a free and open source dev tool that walks you through assessing a website for WCAG 2.1 AA coverage. Example accessibility tests Accessibility tests work just like any other Playwright test. You can either create separate test cases for them, or integrate accessibility scans and assertions into your existing test cases. The following examples demonstrate a few basic accessibility testing scenarios.

Scanning an entire page

This example demonstrates how to test an entire page for automatically detectable accessibility violations. The test:

- Imports the `@axe-core/playwright` package
- Uses normal Playwright Test syntax to define a test case
- Uses normal Playwright syntax to navigate to the page under test
- Awaits `AxeBuilder.analyze()` to run the accessibility scan against the page
- Uses normal Playwright Test assertions to verify that there are no violations in the returned scan results

```
TypeScriptJavaScriptimport { test, expect } from '@playwright/test';import AxeBuilder from '@axe-core/playwright'; // 1test.describe('homepage', () => { // 2  test('should not have any automatically detectable accessibility issues', async ({ page }) => {    await page.goto('https://your-site.com/'); // 3    const accessibilityScanResults = await new AxeBuilder({ page }).analyze(); // 4    expect(accessibilityScanResults.violations).toEqual([]); // 5  });});const { test, expect } = require('@playwright/test');const AxeBuilder = require('@axe-core/playwright').default; // 1test.describe('homepage', () => { // 2  test('should not have any automatically detectable accessibility issues', async ({ page }) => {    await page.goto('https://your-site.com/'); // 3    const accessibilityScanResults = await new AxeBuilder({ page }).analyze(); // 4    expect(accessibilityScanResults.violations).toEqual([]); // 5  });});
```

Configuring axe to scan a specific part of a page

`@axe-core/playwright` supports many configuration options for axe. You can specify these options by using a Builder pattern with the `AxeBuilder` class. For example, you can use `AxeBuilder.include()` to constrain an accessibility scan to only run against one specific part of a page. `AxeBuilder.analyze()` will scan the page in its current state when you call it. To scan parts of a page that are revealed based on UI interactions, use `Locators` to interact with the page before invoking `analyze()`:

```
test('navigation menu flyout should not have automatically detectable accessibility violations', async ({ page }) => {  await page.goto('https://your-site.com/');  await page.getByRole('button', { name: 'Navigation Menu' }).click(); // It is important to wait for() the page to be in the desired // state *before* running analyze(). Otherwise, axe might not // find all the elements your test expects it to scan.  await page.locator('#navigation-menu-flyout').waitFor();  const accessibilityScanResults = await new AxeBuilder({ page })    .include('#navigation-menu-flyout')    .analyze();  expect(accessibilityScanResults.violations).toEqual([]);});
```

Scanning for WCAG violations

By default, axe checks against a wide variety of accessibility rules. Some of these rules correspond to specific success criteria from the Web Content Accessibility Guidelines (WCAG), and others are "best practice" rules that are not specifically required by any WCAG criterion. You can constrain an accessibility scan to only run those rules which are "tagged" as corresponding to specific WCAG success criteria by using `AxeBuilder.withTags()`. For example, Accessibility Insights for Web's Automated Checks only include axe rules that test for violations of WCAG A and AA success criteria; to match that behavior, you would use the tags `wcag2a`, `wcag2aa`, `wcag21a`, and

wcag21aa. Note that automated testing cannot detect all types of WCAG violations.

```
test('should not have any automatically detectable WCAG A or AA violations',
  async ({ page }) => {
    await page.goto('https://your-site.com/');
    const accessibilityScanResults = await new AxeBuilder({ page })
      .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
      .analyze();
    expect(accessibilityScanResults.violations).toEqual([]);
  });
```

You can find a complete listing of the rule tags axe-core supports in the "Axe-core Tags" section of the axe API documentation.

Handling known issues

A common question when adding accessibility tests to an application is "how do I suppress known violations?" The following examples demonstrate a few techniques you can use.

Excluding individual elements from a scan

If your application contains a few specific elements with known issues, you can use `AxeBuilder.exclude()` to exclude them from being scanned until you're able to fix the issues. This is usually the simplest option, but it has some important downsides: `exclude()` will exclude the specified elements and all of their descendants. Avoid using it with components that contain many children. `exclude()` will prevent all rules from running against the specified elements, not just the rules corresponding to known issues. Here is an example of excluding one element from being scanned in one specific test:

```
test('should not have any accessibility violations outside of elements with known issues',
  async ({ page }) => {
    await page.goto('https://your-site.com/page-with-known-issues');
    const accessibilityScanResults = await new
    AxeBuilder({ page })
      .exclude('#element-with-known-issue')
      .analyze();
    expect(accessibilityScanResults.violations).toEqual([]);
  });
```

If the element in question is used repeatedly in many pages, consider using a test fixture to reuse the same `AxeBuilder` configuration across multiple tests.

Disabling individual scan rules

If your application contains many different pre-existing violations of a specific rule, you can use `AxeBuilder.disableRules()` to temporarily disable individual rules until you're able to fix the issues. You can find the rule IDs to pass to `disableRules()` in the `id` property of the violations you want to suppress. A complete list of axe's rules can be found in axe-core's documentation.

```
test('should not have any accessibility violations outside of rules with known issues',
  async ({ page }) => {
    await page.goto('https://your-site.com/page-with-known-issues');
    const accessibilityScanResults = await new
    AxeBuilder({ page })
      .disableRules(['duplicate-id'])
      .analyze();
    expect(accessibilityScanResults.violations).toEqual([]);
  });
```

Using snapshots to allow specific known issues

If you would like to allow for a more granular set of known issues, you can use Snapshots to verify that a set of pre-existing violations has not changed. This approach avoids the downsides of using `AxeBuilder.exclude()` at the cost of slightly more complexity and fragility. Do not use a snapshot of the entire `accessibilityScanResults.violations` array. It contains implementation details of the elements in question, such as a snippet of their rendered HTML; if you include these in your snapshots, it will make your tests prone to breaking every time one of the components in question changes for an unrelated reason. **// Don't do this! This is fragile.**

```
expect(accessibilityScanResults.violations).toMatchSnapshot();
```

Instead, create a fingerprint of the violation(s) in question that contains only enough information to uniquely identify the issue, and use a snapshot of the fingerprint. **// This is less fragile than snapshotting the entire violations array.**

```
expect(violationFingerprints(accessibilityScanResults)).toMatchSnapshot();
```

// my-

```

test-utils.jsfunction violationFingerprints(accessibilityScanResults) { const
violationFingerprints = accessibilityScanResults.violations.map(violation => ({ rule:
violation.id, // These are CSS selectors which uniquely identify each element with //
a violation of the rule in question. targets: violation.nodes.map(node =>
node.target), })); return JSON.stringify(violationFingerprints, null, 2);}Exporting scan
results as a test attachment Most accessibility tests are primarily concerned with the
violations property of the axe scan results. However, the scan results contain more than
just violations. For example, the results also contain information about rules which
passed and about elements which axe found to have inconclusive results for some
rules. This information can be useful for debugging tests that aren't detecting all the
violations you expect them to.To include all of the scan results as part of your test
results for debugging purposes, you can add the scan results as a test attachment with
testInfo.attach(). Reporters can then embed or link the full results as part of your test
output.The following example demonstrates attaching scan results to a
test:test('example with attachment', async ({ page }, testInfo) => { await
page.goto('https://your-site.com/'); const accessibilityScanResults = await new
AxeBuilder({ page }).analyze(); await testInfo.attach('accessibility-scan-results',
{ body: JSON.stringify(accessibilityScanResults, null, 2), contentType: 'application/
json' }); expect(accessibilityScanResults.violations).toEqual([]));Using a test fixture
for common axe configuration Test fixtures are a good way to share common AxeBuilder
configuration across many tests. Some scenarios where this might be useful
include:Using a common set of rules among all of your testsSuppressing a known
violation in a common element which appears in many different pagesAttaching
standalone accessibility reports consistently for many scansThe following example
demonstrates creating and using a test fixture that covers each of those
scenarios.Creating a fixture This example fixture creates an AxeBuilder object which is
pre-configured with shared withTags() and exclude()
configuration.TypeScriptJavaScriptaxe-test.tsimport { test as base } from '@playwright/
test';import AxeBuilder from '@axe-core/playwright';type AxeFixture =
{ makeAxeBuilder: () => AxeBuilder;};// Extend base test by providing
"makeAxeBuilder"//// This new "test" can be used in multiple test files, and each of them
will get// a consistently configured AxeBuilder instance.export const test =
base.extend<AxeFixture>({ makeAxeBuilder: async ({ page }, use, testInfo) =>
{ const makeAxeBuilder = () => new AxeBuilder({ page }) .withTags(['wcag2a',
'wcag2aa', 'wcag21a', 'wcag21aa']) .exclude('#commonly-reused-element-with-
known-issue'); await use(makeAxeBuilder); });export { expect } from '@playwright/
test';// axe-test.jsconst base = require('@playwright/test');const AxeBuilder =
require('@axe-core/playwright').default;// Extend base test by providing
"makeAxeBuilder"//// This new "test" can be used in multiple test files, and each of them
will get// a consistently configured AxeBuilder instance.exports.test =
base.test.extend({ makeAxeBuilder: async ({ page }, use, testInfo) => { const
makeAxeBuilder = () => new AxeBuilder({ page }) .withTags(['wcag2a', 'wcag2aa',
'wcag21a', 'wcag21aa']) .exclude('#commonly-reused-element-with-known-
issue'); await use(makeAxeBuilder); });exports.expect = base.expect;Using a fixture
To use the fixture, replace the earlier examples' new AxeBuilder({ page }) with the newly
defined makeAxeBuilder fixture:const { test, expect } = require('./axe-test');test('example

```

```

using custom fixture', async ({ page, makeAxeBuilder }) => { await page.goto('https://
your-site.com/'); const accessibilityScanResults = await makeAxeBuilder() //
Automatically uses the shared AxeBuilder configuration, // but supports additional
test-specific configuration too .include('#specific-element-under-test') .analyze();
expect(accessibilityScanResults.violations).toEqual([]);});
ActionsPlaywright can interact with HTML Input elements such as text inputs,
checkboxes, radio buttons, select options, mouse clicks, type characters, keys and
shortcuts as well as upload files and focus elements. Text input Using locator.fill() is the
easiest way to fill out the form fields. It focuses the element and triggers an input event
with the entered text. It works for <input>, <textarea> and [contenteditable] elements.//
Text inputawait page.getByRole('textbox').fill('Peter');// Date inputawait
page.getByLabel('Birth date').fill('2020-02-02');// Time inputawait
page.getByLabel('Appointment time').fill('13:15');// Local datetime inputawait
page.getByLabel('Local time').fill('2020-03-02T05:15');// Checkboxes and radio buttons
Using locator.setChecked() is the easiest way to check and uncheck a checkbox or a
radio button. This method can be used with input[type=checkbox], input[type=radio] and
[role=checkbox] elements.// Check the checkboxawait page.getByLabel('I agree to the
terms above').check();// Assert the checked stateexpect(await
page.getByLabel('Subscribe to newsletter').isChecked()).toBeTruthy();// Select the radio
buttonawait page.getByLabel('XL').check();// Select options Selects one or multiple
options in the <select> element with locator.selectOption(). You can specify option
value, or label to select. Multiple options can be selected.// Single selection matching
the valueawait page.getByLabel('Choose a color').selectOption('blue');// Single
selection matching the labelawait page.getByLabel('Choose a
color').selectOption({ label: 'Blue' });// Multiple selected itemsawait
page.getByLabel('Choose multiple colors').selectOption(['red', 'green', 'blue']);Mouse
click Performs a simple human click.// Generic clickawait
page.getByRole('button').click();// Double clickawait page.getByText('Item').dblclick();//
Right clickawait page.getByText('Item').click({ button: 'right' });// Shift + clickawait
page.getByText('Item').click({ modifiers: ['Shift'] });// Hover over elementawait
page.getByText('Item').hover();// Click the top left cornerawait
page.getByText('Item').click({ position: { x: 0, y: 0 } });Under the hood, this and other
pointer-related methods:wait for element with given selector to be in DOMwait for it to
become displayed, i.e. not empty, no display:none, no visibility:hiddenwait for it to stop
moving, for example, until css transition finishesscroll the element into viewwait for it to
receive pointer events at the action point, for example, waits until element becomes non-
obscured by other elementsretry if the element is detached during any of the above
checksForcing the click Sometimes, apps use non-trivial logic where hovering the
element overlays it with another element that intercepts the click. This behavior is
indistinguishable from a bug where element gets covered and the click is dispatched
elsewhere. If you know this is taking place, you can bypass the actionability checks and
force the click:await page.getByRole('button').click({ force: true });Programmatic click If
you are not interested in testing your app under the real conditions and want to
simulate the click by any means possible, you can trigger the HTMLElement.click()
behavior via simply dispatching a click event on the element with
locator.dispatchEvent():await page.getByRole('button').dispatchEvent('click');Type

```

characters Type into the field character by character, as if it was a user with a real keyboard with `locator.type()`.// Type character by character
`await page.locator('#area').type('Hello World!');`This method will emit all the necessary keyboard events, with all the `keydown`, `keyup`, `keypress` events in place. You can even specify the optional delay between the key presses to simulate real user behavior.
`note`Most of the time, `page.fill()` will just work. You only need to type characters if there is special keyboard handling on the page.
Keys and shortcuts // Hit Enter
`await page.getByText('Submit').press('Enter');`// Dispatch Control+Right
`await page.getByRole('textbox').press('Control+ArrowRight');`// Press \$ sign on keyboard
`await page.getByRole('textbox').press('$');`The `locator.press()` method focuses the selected element and produces a single keystroke. It accepts the logical key names that are emitted in the `keyboardEvent.key` property of the keyboard events: `Backquote`, `Minus`, `Equal`, `Backslash`, `Backspace`, `Tab`, `Delete`, `Escape`, `ArrowDown`, `End`, `Enter`, `Home`, `Insert`, `PageDown`, `PageUp`, `ArrowRight`, `ArrowUp`, `F1 - F12`, `Digit0 - Digit9`, `KeyA - KeyZ`, etc. You can alternatively specify a single character you'd like to produce such as `"a"` or `"#"`. Following modification shortcuts are also supported: `Shift`, `Control`, `Alt`, `Meta`. Simple version produces a single character. This character is case-sensitive, so `"a"` and `"A"` will produce different results.// `<input id=name>`
`await page.locator('#name').press('Shift+A');`// `<input id=name>`
`await page.locator('#name').press('Shift+ArrowLeft');`Shortcuts such as `"Control+o"` or `"Control+Shift+T"` are supported as well. When specified with the modifier, modifier is pressed and being held while the subsequent key is being pressed. Note that you still need to specify the capital A in `Shift-A` to produce the capital character. `Shift-a` produces a lower-case one as if you had the `CapsLock` toggled.
Upload files You can select input files for upload using the `locator.setInputFiles()` method. It expects first argument to point to an input element with the type `"file"`. Multiple files can be passed in the array. If some of the file paths are relative, they are resolved relative to the current working directory. Empty array clears the selected files.// Select one file
`await page.getByLabel('Upload file').setInputFiles('myfile.pdf');`// Select multiple files
`await page.getByLabel('Upload files').setInputFiles(['file1.txt', 'file2.txt']);`// Remove all the selected files
`await page.getByLabel('Upload file').setInputFiles([]);`// Upload buffer from memory
`await page.getByLabel('Upload file').setInputFiles({ name: 'file.txt', mimeType: 'text/plain', buffer: Buffer.from('this is test')});`If you don't have input element in hand (it is created dynamically), you can handle the `page.on('filechooser')` event or use a corresponding waiting method upon your action.// Start waiting for file chooser before clicking. Note no `await`.
`const fileChooserPromise = page.waitForEvent('filechooser');`
`await page.getByLabel('Upload file').click();`
`const fileChooser = await fileChooserPromise;`
`await fileChooser.setFiles('myfile.pdf');`
Focus element For the dynamic pages that handle focus events, you can focus the given element with `locator.focus()`.
`await page.getByLabel('Password').focus();`
Drag and Drop You can perform drag&drop operation with `locator.dragTo()`. This method will:
Hover the element that will be dragged.
Press left mouse button.
Move mouse to the element that will receive the drop.
Release left mouse button.
`await page.locator('#item-to-be-dragged').dragTo(page.locator('#item-to-drop-at'));`
Dragging manually If you want precise control over the drag operation, use lower-level methods like `locator.hover()`, `mouse.down()`, `mouse.move()` and `mouse.up()`.
`await page.locator('#item-to-be-`

dragged').hover();await page.mouse.down();await page.locator('#item-to-drop-at').hover();await page.mouse.up();noteIf your page relies on the dragover event being dispatched, you need at least two mouse moves to trigger it in all browsers. To reliably issue the second mouse move, repeat your mouse.move() or locator.hover() twice. The sequence of operations would be: hover the drag element, mouse down, hover the drop element, hover the drop element second time, mouse up.

AssertionsPlaywright includes test assertions in the form of expect function. To make an assertion, call expect(value) and choose a matcher that reflects the expectation. There are many generic matchers like toEqual, toContain, toBeTruthy that can be used to assert any conditions.expect(success).toBeTruthy();Playwright also includes web-specific async matchers that will wait until the expected condition is met. Consider the following example:await

expect(page.getByTestId('status')).toHaveText('Submitted');Playwright 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 value in the test config. By default, the timeout for assertions is set to 5 seconds. Learn more about various timeouts.List of assertions

AssertionDescriptionexpect(locator).toBeAttached()Element is attachedexpect(locator).toBeChecked()Checkbox is checkedexpect(locator).toBeDisabled()Element is disabledexpect(locator).toBeEditable()Element is editableexpect(locator).toBeEmpty()Container is emptyexpect(locator).toBeEnabled()Element is enabledexpect(locator).toBeFocused()Element is focusedexpect(locator).toBeHidden()Element is not visibleexpect(locator).toBeInViewport()Element intersects viewportexpect(locator).toBeVisible()Element is visibleexpect(locator).toContainText()Element contains textexpect(locator).toHaveAttribute()Element has a DOM attributeexpect(locator).toHaveClass()Element has a class propertyexpect(locator).toHaveCount()List has exact number of childrenexpect(locator).toHaveCSS()Element has CSS propertyexpect(locator).toHaveId()Element has an IDexpect(locator).toHaveJSProperty()Element has a JavaScript propertyexpect(locator).toHaveScreenshot()Element has a screenshotexpect(locator).toHaveText()Element matches textexpect(locator).toHaveValue()Input has a valueexpect(locator).toHaveValues()Select has options selectedexpect(page).toHaveScreenshot()Page has a screenshotexpect(page).toHaveTitle()Page has a titleexpect(page).toHaveURL()Page has a URLexpect(apiResponse).toBeOK()Response has an OK statusNegating 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 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 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);expect.configure

You can create your own pre-configured expect instance to have its own defaults such as timeout and soft.const slowExpect = expect.configure({ timeout: 10000 });await slowExpect(locator).toHaveText('Submit');// Always do soft assertions.const softExpect = expect.configure({ soft: true });await softExpect(locator).toHaveText('Submit');expect.poll

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);expect.toPass

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});

API testingPlaywright can be used to get access to the REST API of your application.Sometimes you may want to send requests to the server directly from Node.js without loading a page and running js code in it. A few examples where it may

come in handy: Test your server API. Prepare server side state before visiting the web application in a test. Validate server side post-conditions after running some actions in the browser. All of that could be achieved via `APIRequestContext` methods. Writing API Test Configuration Writing tests Setup and teardown Using request context Sending API requests from UI tests Establishing preconditions Validating postconditions Reusing authentication state Context request vs global request Writing API Test

`APIRequestContext` can send all kinds of HTTP(S) requests over network. The following example demonstrates how to use Playwright to test issues creation via GitHub API. The test suite will do the following: Create a new repository before running tests. Create a few issues and validate server state. Delete the repository after running tests. Configuration GitHub API requires authorization, so we'll configure the token once for all tests. While at it, we'll also set the `baseURL` to simplify the tests. You can either put them in the configuration file, or in the test file with

```
test.use().playwright.config.ts import { defineConfig } from '@playwright/test'; export default defineConfig({ use: { // All requests we send go to this API endpoint. baseURL: 'https://api.github.com', extraHTTPHeaders: { // We set this header per GitHub guidelines. 'Accept': 'application/vnd.github.v3+json', // Add authorization token to all requests. // Assuming personal access token available in the environment. 'Authorization': `token ${process.env.API_TOKEN}`, }, });
```

Writing tests Playwright Test comes with the built-in request fixture that respects configuration options like `baseURL` or `extraHTTPHeaders` we specified and is ready to send some requests. Now we can add a few tests that will create new issues in the repository.

```
const REPO = 'test-repo-1'; const USER = 'github-username'; test('should create a bug report', async ({ request }) => { const newIssue = await request.post(`/repos/${USER}/${REPO}/issues`, { data: { title: '[Bug] report 1', body: 'Bug description', } }); expect(newIssue.ok()).toBeTruthy(); const issues = await request.get(`/repos/${USER}/${REPO}/issues`); expect(issues.ok()).toBeTruthy(); expect(await issues.json()).toContainEqual(expect.objectContaining({ title: '[Bug] report 1', body: 'Bug description' }))); test('should create a feature request', async ({ request }) => { const newIssue = await request.post(`/repos/${USER}/${REPO}/issues`, { data: { title: '[Feature] request 1', body: 'Feature description', } }); expect(newIssue.ok()).toBeTruthy(); const issues = await request.get(`/repos/${USER}/${REPO}/issues`); expect(issues.ok()).toBeTruthy(); expect(await issues.json()).toContainEqual(expect.objectContaining({ title: '[Feature] request 1', body: 'Feature description' })));
```

Setup and teardown These tests assume that repository exists. You probably want to create a new one before running tests and delete it afterwards. Use `beforeAll` and `afterAll` hooks for that.

```
test.beforeAll(async ({ request }) => { // Create a new repository const response = await request.post('/user/repos', { data: { name: REPO } }); expect(response.ok()).toBeTruthy(); test.afterAll(async ({ request }) => { // Delete the repository const response = await request.delete(`/repos/${USER}/${REPO}`); expect(response.ok()).toBeTruthy();
```

Using request context Behind the scenes, request fixture will actually call `apiRequest.newContext()`. You can always do that manually if you'd like more control. Below is a standalone script that does the same as `beforeAll` and `afterAll` from above.

```
import { request } from '@playwright/test'; const REPO = 'test-repo-1'; const USER = 'github-username'; (async () => { // Create a context that will
```


issue http requests. `const context = await request.newContext({ baseUrl: 'https://api.github.com', });` // Create a repository. `await context.post('/user/repos', { headers: { 'Accept': 'application/vnd.github.v3+json', // Add GitHub personal access token. 'Authorization': `token ${process.env.API_TOKEN}`, }, data: { name: REPO } });` // Delete a repository. `await context.delete(`/repos/${USER}/${REPO}`, { headers: { 'Accept': 'application/vnd.github.v3+json', // Add GitHub personal access token. 'Authorization': `token ${process.env.API_TOKEN}`, } });`());

Sending API requests from UI tests While running tests inside browsers you may want to make calls to the HTTP API of your application. It may be helpful if you need to prepare server state before running a test or to check some postconditions on the server after performing some actions in the browser. All of that could be achieved via `APIRequestContext` methods.

Establishing preconditions

The following test creates a new issue via API and then navigates to the list of all issues in the project to check that it appears at the top of the list.

```
import { test, expect } from '@playwright/test';
const REPO = 'test-repo-1';
const USER = 'github-username';
// Request context is reused by all tests in the file.
let apiContext;
test.beforeAll(async ({ playwright }) => {
  apiContext = await playwright.request.newContext({
    // All requests we send go to this API endpoint.
    baseUrl: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`
    },
  });
});
test.afterAll(async ({ }) => {
  // Dispose all responses.
  await apiContext.dispose();
});
test('last created issue should be first in the list', async ({ page }) => {
  const newIssue = await apiContext.post(`/repos/${USER}/${REPO}/issues`, {
    data: { title: '[Feature] request 1', }
  });
  expect(newIssue.ok()).toBeTruthy();
  await page.goto(`https://github.com/${USER}/${REPO}/issues`);
  const firstIssue = page.locator('a[data-hovercard-type=issue]').first();
  await expect(firstIssue).toHaveText('[Feature] request 1');
});
```

Validating postconditions

The following test creates a new issue via user interface in the browser and then uses checks if it was created via API:

```
import { test, expect } from '@playwright/test';
const REPO = 'test-repo-1';
const USER = 'github-username';
// Request context is reused by all tests in the file.
let apiContext;
test.beforeAll(async ({ playwright }) => {
  apiContext = await playwright.request.newContext({
    // All requests we send go to this API endpoint.
    baseUrl: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  });
});
test.afterAll(async ({ }) => {
  // Dispose all responses.
  await apiContext.dispose();
});
test('last created issue should be on the server', async ({ page, request }) => {
  await page.goto(`https://github.com/${USER}/${REPO}/issues`);
  await page.getByText('New Issue').click();
  await page.getByRole('textbox', { name: 'Title' }).fill('Bug report 1');
  await page.getByRole('textbox', { name: 'Comment body' }).fill('Bug description');
  await page.getByText('Submit new issue').click();
  const issueId = page.url().substr(page.url().lastIndexOf('/'));
  const newIssue = await request.get(`https://api.github.com/repos/${USER}/${REPO}/issues/${issueId}`);
});
```

```

expect(newIssue.ok()).toBeTruthy();
expect(newIssue.json()).toEqual(expect.objectContaining({ title: 'Bug report
1' }));});Reusing authentication state Web apps use cookie-based or token-based
authentication, where authenticated state is stored as cookies. Playwright provides
apiRequestContext.storageState() method that can be used to retrieve storage state
from an authenticated context and then create new contexts with that state.Storage
state is interchangeable between BrowserContext and APIRequestContext. You can
use it to log in via API calls and then create a new context with cookies already there.
The following code snippet retrieves state from an authenticated APIRequestContext
and creates a new BrowserContext with that state.const requestContext = await
request.newContext({ httpCredentials: { username: 'user', password:
'passwd' }});await requestContext.get('https://api.example.com/login');// Save storage
state into the file.await requestContext.storageState({ path: 'state.json' });// Create a
new context with the saved storage state.const context = await
browser.newContext({ storageState: 'state.json' });Context request vs global request
There are two types of APIRequestContext:associated with a BrowserContextisolated
instance, created via apiRequest.newContext()The main difference is that
APIRequestContext accessible via browserContext.request and page.request will
populate request's Cookie header from the browser context and will automatically
update browser cookies if APIResponse has Set-Cookie header:test('context request
will share cookie storage with its browser context', async ({ page, context }) => { await
context.route('https://www.github.com/', async route => { // Send an API request that
shares cookie storage with the browser context. const response = await
context.request.fetch(route.request()); const responseHeaders =
response.headers(); // The response will have 'Set-Cookie' header. const
responseCookies = new Map(responseHeaders['set-cookie'].split('\n').map(c =>
c.split(';')[0].split('='))); // The response will have 3 cookies in 'Set-Cookie' header.
expect(responseCookies.size).toBe(3); const contextCookies = await
context.cookies(); // The browser context will already contain all the cookies from the
API response. expect(new Map(contextCookies.map(({ name, value }) => [name,
value])).toEqual(responseCookies); route.fulfill({ response, headers:
{ ...responseHeaders, foo: 'bar' }, }); }); await page.goto('https://
www.github.com/');});If you don't want APIRequestContext to use and update cookies
from the browser context, you can manually create a new instance of
APIRequestContext which will have its own isolated cookies:test('global context request
has isolated cookie storage', async ({ page, context, browser, playwright }) => { //
Create a new instance of APIRequestContext with isolated cookie storage. const
request = await playwright.request.newContext(); await context.route('https://
www.github.com/', async route => { const response = await
request.fetch(route.request()); const responseHeaders = response.headers(); const
responseCookies = new Map(responseHeaders['set-cookie'].split('\n').map(c =>
c.split(';')[0].split('='))); // The response will have 3 cookies in 'Set-Cookie' header.
expect(responseCookies.size).toBe(3); const contextCookies = await
context.cookies(); // The browser context will not have any cookies from the isolated
API request. expect(contextCookies.length).toBe(0); // Manually export cookie
storage. const storageState = await request.storageState(); // Create a new context

```

and initialize it with the cookies from the global request. `const browserContext2 = await browser.newContext({ storageState }); const contextCookies2 = await browserContext2.cookies(); // The new browser context will already contain all the cookies from the API response. expect(new Map(contextCookies2.map(({ name, value }) => [name, value]))).toEqual(responseCookies); route.fulfill({ response, headers: { ...responseHeaders, foo: 'bar' }, }); }); await page.goto('https://www.github.com/'); await request.dispose();});`

AuthenticationPlaywright executes tests in isolated environments called browser contexts. This isolation model improves reproducibility and prevents cascading test failures. Tests can load existing authenticated state. This eliminates the need to authenticate in every test and speeds up test execution. Core concepts Regardless of the authentication strategy you choose, you are likely to store authenticated browser state on the file system. We recommend to create `playwright/.auth` directory and add it to your `.gitignore`. Your authentication routine will produce authenticated browser state and save it to a file in this `playwright/.auth` directory. Later on, tests will reuse this state and start already authenticated. `BashPowerShellBatchmkdir -p playwright/.authecho "\nplaywright/.auth" >> .gitignoreNew-Item -ItemType Directory -Force -Path playwright\.authAdd-Content -path .gitignore "r`nplaywright/.auth"md playwright\.authecho. >> .gitignoreecho "playwright/.auth" >> .gitignoreBasic: shared account in all tests` This is the recommended approach for tests without server-side state. Authenticate once in the setup project, save the authentication state, and then reuse it to bootstrap each test already authenticated. When to useWhen you can imagine all your tests running at the same time with the same account, without affecting each other. When not to useYour tests modify server-side state. For example, one test checks the rendering of the settings page, while the other test is changing the setting, and you run tests in parallel. In this case, tests must use different accounts. Your authentication is browser-specific. DetailsCreate `tests/auth.setup.ts` that will prepare authenticated browser state for all other tests. `tests/auth.setup.tsimport { test as setup, expect } from '@playwright/test';const authFile = 'playwright/.auth/user.json';setup('authenticate', async ({ page }) => { // Perform authentication steps. Replace these actions with your own. await page.goto('https://github.com/login'); await page.getByLabel('Username or email address').fill('username'); await page.getByLabel('Password').fill('password'); await page.getByRole('button', { name: 'Sign in' }).click(); // Wait until the page receives the cookies. // Sometimes login flow sets cookies in the process of several redirects. // Wait for the final URL to ensure that the cookies are actually set. await page.waitForURL('https://github.com/'); // Alternatively, you can wait until the page reaches a state where all cookies are set. await expect(page.getByRole('button', { name: 'View profile and more' })).toBeVisible(); // End of authentication steps. await page.context().storageState({ path: authFile });});` Create a new setup project in the config and declare it as a dependency for all your testing projects. This project will always run and authenticate before all the tests. All testing projects should use the authenticated state as `storageState`. `playwright.config.tsimport { defineConfig, devices } from '@playwright/test';export default defineConfig({ projects: [// Setup project { name: 'setup', testMatch: /\.*\setup\.ts/ }, { name: 'chromium', use: { ...devices['Desktop Chrome'], // Use prepared auth state. storageState:`

```

'playwright/.auth/user.json', }, dependencies: ['setup'], }, { name:
'firefox', use: { ...devices['Desktop Firefox'], // Use prepared auth state.
storageState: 'playwright/.auth/user.json', }, dependencies: ['setup'], }, ]});Tests
start already authenticated because we specified storageState in the config.tests/
example.spec.tsimport { test } from '@playwright/test';test('test', async ({ page }) => { //
page is authenticated});Moderate: one account per parallel worker This is the
recommended approach for tests that modify server-side state. In Playwright, worker
processes run in parallel. In this approach, each parallel worker is authenticated once.
All tests ran by worker are reusing the same authentication state. We will need multiple
testing accounts, one per each parallel worker.When to useYour tests modify shared
server-side state. For example, one test checks the rendering of the settings page,
while the other test is changing the setting.When not to useYour tests do not modify any
shared server-side state. In this case, all tests can use a single shared
account.DetailsWe will authenticate once per worker process, each with a unique
account.Create playwright/fixtures.ts file that will override storageState fixture to
authenticate once per worker. Use testInfo.parallelIndex to differentiate between
workers.playwright/fixtures.tsimport { test as baseTest, expect } from '@playwright/
test';import fs from 'fs';import path from 'path';export * from '@playwright/test';export
const test = baseTest.extend<{ }, { workerStorageState: string }>({ // Use the same
storage state for all tests in this worker. storageState: ({ workerStorageState }, use) =>
use(workerStorageState), // Authenticate once per worker with a worker-scoped
fixture. workerStorageState: [async ({ browser }, use) => { // Use parallelIndex as a
unique identifier for each worker. const id = test.info().parallelIndex; const fileName
= path.resolve(test.info().project.outputDir, `.auth/${id}.json`); if
(fs.existsSync(fileName)) { // Reuse existing authentication state if any. await
use(fileName); return; } // Important: make sure we authenticate in a clean
environment by unsetting storage state. const page = await
browser.newPage({ storageState: undefined }); // Acquire a unique account, for
example create a new one. // Alternatively, you can have a list of precreated accounts
for testing. // Make sure that accounts are unique, so that multiple team members //
can run tests at the same time without interference. const account = await
acquireAccount(id); // Perform authentication steps. Replace these actions with your
own. await page.goto('https://github.com/login'); await page.getByLabel('Username
or email address').fill(account.username); await
page.getByLabel('Password').fill(account.password); await page.getByRole('button',
{ name: 'Sign in' }).click(); // Wait until the page receives the cookies. // //
Sometimes login flow sets cookies in the process of several redirects. // Wait for the
final URL to ensure that the cookies are actually set. await page.waitForURL('https://
github.com/'); // Alternatively, you can wait until the page reaches a state where all
cookies are set. await expect(page.getByRole('button', { name: 'View profile and
more' })).toBeVisible(); // End of authentication steps. await
page.context().storageState({ path: fileName }); await page.close(); await
use(fileName); }, { scope: 'worker' }]);Now, each test file should import test from our
fixtures file instead of @playwright/test. No changes are needed in the config.tests/
example.spec.ts// Important: import our fixtures.import { test, expect } from './playwright/
fixtures';test('test', async ({ page }) => { // page is authenticated});Advanced scenarios

```

Authenticate with API request When to use Your web application supports authenticating via API that is easier/faster than interacting with the app UI. Details We will send the API request with APIRequestContext and then save authenticated state as usual. In the setup project: tests/auth.setup.ts

```
import { test as setup } from '@playwright/test';
const authFile = 'playwright/.auth/user.json';
setup('authenticate', async ({ request }) => {
  // Send authentication request. Replace with your own.
  await request.post('https://github.com/login', {
    form: { 'user': 'user', 'password': 'password' }
  });
  await request.storageState({ path: authFile });
});
```

Alternatively, in a worker fixture: playwright/fixtures.ts

```
import { test as baseTest, request } from '@playwright/test';
import fs from 'fs';
import path from 'path';
export * from '@playwright/test';
export const test = baseTest.extend<{}>({
  workerStorageState: string
})(>({
  // Use the same storage state for all tests in this worker.
  storageState: ({ workerStorageState }, use) =>
    use(workerStorageState),
  // Authenticate once per worker with a worker-scoped fixture.
  workerStorageState: [async ({}, use) => {
    // Use parallelIndex as a unique identifier for each worker.
    const id = test.info().parallelIndex;
    const fileName = path.resolve(test.info().project.outputDir, `.auth/${id}.json`);
    if (fs.existsSync(fileName)) {
      // Reuse existing authentication state if any.
      await use(fileName);
      return;
    }
    // Important: make sure we authenticate in a clean environment by unsetting storage state.
    const context = await request.newContext({ storageState: undefined });
    // Acquire a unique account, for example create a new one.
    // Alternatively, you can have a list of precreated accounts for testing.
    // Make sure that accounts are unique, so that multiple team members
    // can run tests at the same time without interference.
    const account = await acquireAccount(id);
    // Send authentication request. Replace with your own.
    await context.post('https://github.com/login', {
      form: { 'user': 'user', 'password': 'password' }
    });
    await context.storageState({ path: fileName });
    await context.dispose();
    await use(fileName);
  }, { scope: 'worker' }
]);
```

Multiple signed in roles When to use You have more than one role in your end to end tests, but you can reuse accounts across all tests. Details We will authenticate multiple times in the setup project: tests/auth.setup.ts

```
import { test as setup, expect } from '@playwright/test';
const adminFile = 'playwright/.auth/admin.json';
setup('authenticate as admin', async ({ page }) => {
  // Perform authentication steps. Replace these actions with your own.
  await page.goto('https://github.com/login');
  await page.getByLabel('Username or email address').fill('admin');
  await page.getByLabel('Password').fill('password');
  await page.getByRole('button', { name: 'Sign in' }).click();
  // Wait until the page receives the cookies.
  // Sometimes login flow sets cookies in the process of several redirects.
  // Wait for the final URL to ensure that the cookies are actually set.
  await page.waitForURL('https://github.com/');
  // Alternatively, you can wait until the page reaches a state where all cookies are set.
  await expect(page.getByRole('button', { name: 'View profile and more' })).toBeVisible();
  // End of authentication steps.
  await page.context().storageState({ path: adminFile });
});
```

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

```
setup('authenticate as user', async ({ page }) => {
  // Perform authentication steps. Replace these actions with your own.
  await page.goto('https://github.com/login');
  await page.getByLabel('Username or email address').fill('user');
  await page.getByLabel('Password').fill('password');
  await page.getByRole('button', { name: 'Sign in' }).click();
  // Wait until the page receives the cookies.
  // Sometimes login flow
```

sets cookies in the process of several redirects. // Wait for the final URL to ensure that the cookies are actually set. `await page.waitForURL('https://github.com/');` // Alternatively, you can wait until the page reaches a state where all cookies are set. `await expect(page.getByRole('button', { name: 'View profile and more' })).toBeVisible();` // End of authentication steps. `await page.context().storageState({ path: userFile });` After that, specify `storageState` for each test file or test group, instead of setting it in the config. `tests/example.spec.ts` `import { test } from '@playwright/test';` `test.use({ storageState: 'playwright/.auth/admin.json' });` `test('admin test', async ({ page }) => { // page is authenticated as admin });` `test.describe(() => { test.use({ storageState: 'playwright/.auth/user.json' }); test('user test', async ({ page }) => { // page is authenticated as a user }); });` **Testing multiple roles together** When to use You need to test how multiple authenticated roles interact together, in a single test. **Details** Use multiple `BrowserContexts` and `Pages` with different storage states in the same test. `tests/example.spec.ts` `import { test } from '@playwright/test';` `test('admin and user', async ({ browser }) => { // adminContext and all pages inside, including adminPage, are signed in as "admin". const adminContext = await browser.newContext({ storageState: 'playwright/.auth/admin.json' }); const adminPage = await adminContext.newPage(); // userContext and all pages inside, including userPage, are signed in as "user". const userContext = await browser.newContext({ storageState: 'playwright/.auth/user.json' }); const userPage = await userContext.newPage(); // ... interact with both adminPage and userPage ... await adminContext.close(); await userContext.close(); });` **Testing multiple roles with POM fixtures** When to use You need to test how multiple authenticated roles interact together, in a single test. **Details** You can introduce fixtures that will provide a page authenticated as each role. Below is an example that creates fixtures for two `Page Object Models` - `admin POM` and `user POM`. It assumes `adminStorageState.json` and `userStorageState.json` files were created in the global setup. `playwright/fixtures.ts` `import { test as base, type Page, type Locator } from '@playwright/test';` // `Page Object Model` for the "admin" page. // Here you can add locators and helper methods specific to the admin page. `class AdminPage { // Page signed in as "admin". page: Page; // Example locator pointing to "Welcome, Admin" greeting. greeting: Locator; constructor(page: Page) { this.page = page; this.greeting = page.locator('#greeting'); }}` // `Page Object Model` for the "user" page. // Here you can add locators and helper methods specific to the user page. `class UserPage { // Page signed in as "user". page: Page; // Example locator pointing to "Welcome, User" greeting. greeting: Locator; constructor(page: Page) { this.page = page; this.greeting = page.locator('#greeting'); }}` // Declare the types of your fixtures. `type MyFixtures = { adminPage: AdminPage; userPage: UserPage; };` `export * from '@playwright/test';` `export const test = base.extend<MyFixtures>({ adminPage: async ({ browser }, use) => { const context = await browser.newContext({ storageState: 'playwright/.auth/admin.json' }); const adminPage = new AdminPage(await context.newPage()); await use(adminPage); await context.close(); }, userPage: async ({ browser }, use) => { const context = await browser.newContext({ storageState: 'playwright/.auth/user.json' }); const userPage = new UserPage(await context.newPage()); await use(userPage); await context.close(); }, });` `tests/example.spec.ts` // Import test with our new fixtures. `import { test, expect } from '../playwright/fixtures';` // Use `adminPage` and `userPage` fixtures in

```

the test.test('admin and user', async ({ adminPage, userPage }) => { // ... interact with
both adminPage and userPage ... await
expect(adminPage.greeting).toHaveText('Welcome, Admin'); await
expect(userPage.greeting).toHaveText('Welcome, User');});

```

Session storage Reusing authenticated state covers cookies and local storage based authentication. Rarely, session storage is used for storing information associated with the signed-in state. Session storage is specific to a particular domain and is not persisted across page loads. Playwright does not provide API to persist session storage, but the following snippet can be used to save/load session storage.

```

// Get session storage and store as env variable
const sessionStorage = await page.evaluate(() =>
JSON.stringify(sessionStorage));
fs.writeFileSync('playwright/.auth/session.json',
JSON.stringify(sessionStorage), 'utf-8');
// Set session storage in a new context
const sessionStorage = JSON.parse(fs.readFileSync('playwright/.auth/session.json',
'utf-8'));
await context.addInitScript(storage => {
if (window.location.hostname === 'example.com') {
for (const [key, value] of Object.entries(storage))
window.sessionStorage.setItem(key, value);
}}, sessionStorage);

```

Auto-waiting Playwright performs a range of actionability checks on the elements before making actions to ensure these actions behave as expected. It auto-waits for all the relevant checks to pass and only then performs the requested action. If the required checks do not pass within the given timeout, action fails with the `TimeoutError`. For example, for `page.click()`, Playwright will ensure that:

- element is Attached to the DOM
- element is Visible
- element is Stable, as in not animating or completed animation
- element Receives Events, as in not obscured by other elements
- element is Enabled

Here is the complete list of actionability checks performed for each action:

Action	Attached	Visible	Stable	Receives Events	Enabled
click	Yes	Yes	Yes	Yes	Yes
dblclick	Yes	Yes	Yes	Yes	Yes
setChecked	Yes	Yes	Yes	Yes	Yes
tap	Yes	Yes	Yes	Yes	Yes
uncheck	Yes	Yes	Yes	Yes	Yes
hover	Yes	Yes	Yes	Yes	Yes
scrollIntoViewIfNeeded	Yes	Yes	Yes	Yes	Yes
screenshot	Yes	Yes	Yes	Yes	Yes
fill	Yes	Yes	Yes	Yes	Yes
selectText	Yes	Yes	Yes	Yes	Yes
dispatchEvent	Yes	Yes	Yes	Yes	Yes
focus	Yes	Yes	Yes	Yes	Yes
getAttribute	Yes	Yes	Yes	Yes	Yes
innerText	Yes	Yes	Yes	Yes	Yes
innerHTML	Yes	Yes	Yes	Yes	Yes
press	Yes	Yes	Yes	Yes	Yes
setInputFiles	Yes	Yes	Yes	Yes	Yes
selectOption	Yes	Yes	Yes	Yes	Yes
textContent	Yes	Yes	Yes	Yes	Yes
type	Yes	Yes	Yes	Yes	Yes

Forcing actions Some actions like `page.click()` support force option that disables non-essential actionability checks, for example passing `force` to `page.click()` method will not check that the target element actually receives click events.

Assertions You can check the actionability state of the element using one of the following methods as well. This is typically not necessary, but it helps writing assertive tests that ensure that after certain actions, elements reach actionable state:

- `elementHandle.isChecked()`
- `elementHandle.isDisabled()`
- `elementHandle.isEditable()`
- `elementHandle.isEnabled()`
- `elementHandle.isHidden()`
- `elementHandle.isVisible()`
- `page.isChecked()`
- `page.isDisabled()`
- `page.isEditable()`
- `page.isEnabled()`
- `page.isHidden()`
- `page.isVisible()`
- `locator.isChecked()`
- `locator.isDisabled()`
- `locator.isEditable()`
- `locator.isEnabled()`
- `locator.isHidden()`
- `locator.isVisible()`

Attached Element is considered attached when it is connected to a Document or a ShadowRoot.

Visible Element is considered visible when it has non-empty bounding box and does not have `visibility: hidden` computed style. Note that elements of zero size or with `display: none` are not considered visible.

Stable Element is considered stable when it has maintained the same bounding box for at least two consecutive animation frames.

Enabled Element

is considered enabled unless it is a `<button>`, `<select>`, `<input>` or `<textarea>` with a disabled property. Editable Element is considered editable when it is enabled and does not have readonly property set. Receives Events Element is considered receiving pointer events when it is the hit target of the pointer event at the action point. For example, when clicking at the point (10;10), Playwright checks whether some other element (usually an overlay) will instead capture the click at (10;10). For example, consider a scenario where Playwright will click Sign Up button regardless of when the `page.click()` call was made: page is checking that user name is unique and Sign Up button is disabled; after checking with the server, the disabled Sign Up button is replaced with another one that is now enabled.

Best Practices This guide should help you to make sure you are following our best practices and writing tests that are more resilient.

Testing philosophy Test user-visible behavior Automated tests should verify that the application code works for the end users, and avoid relying on implementation details such as things which users will not typically use, see, or even know about such as the name of a function, whether something is an array, or the CSS class of some element. The end user will see or interact with what is rendered on the page, so your test should typically only see/interact with the same rendered output. Make tests as isolated as possible Each test should be completely isolated from another test and should run independently with its own local storage, session storage, data, cookies etc. Test isolation improves reproducibility, makes debugging easier and prevents cascading test failures. In order to avoid repetition for a particular part of your test you can use before and after hooks. Within your test file add a before hook to run a part of your test before each test such as going to a particular URL or logging in to a part of your app. This keeps your tests isolated as no test relies on another. However it is also ok to have a little duplication when tests are simple enough especially if it keeps your tests clearer and easier to read and maintain.

```
import { test } from '@playwright/test';
test.beforeEach(async ({ page }) => {
  // Runs before each test and signs in each page.
  await page.goto('https://github.com/login');
  await page.getByLabel('Username or email address').fill('username');
  await page.getByLabel('Password').fill('password');
  await page.getByRole('button', { name: 'Sign in' }).click();
});
test('first', async ({ page }) => {
  // page is signed in.
});
test('second', async ({ page }) => {
  // page is signed in.
});
```

You can also reuse the signed-in state in the tests with global setup. That way you can log in only once and then skip the log in step for all of the tests.

Avoid testing third-party dependencies Only test what you control. Don't try to test links to external sites or third party servers that you do not control. Not only is it time consuming and can slow down your tests but also you can not control the content of the page you are linking to, or if there are cookie banners or overlay pages or anything else that might cause your test to fail. Instead, use the Playwright Network API and guarantee the response needed.

```
await page.route('**/api/fetch_data_third_party_dependency', route => route.fulfill({ status: 200, body: testData }));
await page.goto('https://example.com');
```

Testing with a database If working with a database then make sure you control the data. Test against a staging environment and make sure it doesn't change. For visual regression tests make sure the operating system and browser versions are the same.

Best Practices Use locators In order to write end to end tests we need to first find elements on the webpage. We can do this by using Playwright's built in locators. Locators come with auto waiting and retry-

ability. Auto waiting means that Playwright performs a range of actionability checks on the elements, such as ensuring the element is visible and enabled before it performs the click. To make tests resilient, we recommend prioritizing user-facing attributes and explicit contracts.

```
// Ø=ÜMpage.getByRole('button', { name: 'submit' });
```

Use chaining and filtering

Locators can be chained to narrow down the search to a particular part of the page.

```
const product = page.getByRole('listitem').filter({ hasText: 'Product 2' });
```

You can also filter locators by text or by another locator.

```
await page.getByRole('listitem').filter({ hasText: 'Product 2' }).getByRole('button', { name: 'Add to cart' }).click();
```

Prefer user-facing attributes to XPath or CSS selectors

Your DOM can easily change so having your tests depend on your DOM structure can lead to failing tests. For example consider selecting this button by its CSS classes. Should the designer change something then the class might change breaking your test.

```
// Ø=ÜNpage.locator('button.buttonIcon.episode-actions-later');
```

Use locators that are resilient to changes in the DOM.

```
// Ø=ÜMpage.getByRole('button', { name: 'submit' });
```

Generate locators

Playwright has a test generator that can generate tests and pick locators for you. It will look at your page and figure out the best locator, prioritizing role, text and test id locators. If the generator finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.

Use codegen to generate locators

To pick a locator run the codegen command followed by the URL that you would like to pick a locator from.

```
npx playwright codegen playwright.dev
```

This will open a new browser window as well as the Playwright inspector. To pick a locator first click on the 'Record' button to stop the recording. By default when you run the codegen command it will start a new recording. Once you stop the recording the 'Pick Locator' button will be available to click. You can then hover over any element on your page in the browser window and see the locator highlighted below your cursor. Clicking on an element will add the locator into the Playwright inspector. You can either copy the locator and paste into your test file or continue to explore the locator by editing it in the Playwright Inspector, for example by modifying the text, and seeing the results in the browser window.

Use the VS Code extension to generate locators

You can also use the VS Code Extension to generate locators as well as record a test. The VS Code extension also gives you a great developer experience when writing, running, and debugging tests.

Use web first assertions

Assertions are a way to verify that the expected result and the actual result matched or not. By using web first assertions Playwright will wait until the expected condition is met. For example, when testing an alert message, a test would click a button that makes a message appear and check that the alert message is there. If the alert message takes half a second to appear, assertions such as `toBeVisible()` will wait and retry if needed.

```
// Ø=ÜMawait expect(page.getByText('welcome')).toBeVisible();
```

```
// Ø=ÜNexpect(await page.getByText('welcome').isVisible()).toBe(true);
```

Don't use manual assertions

Don't use manual assertions that are not awaiting the expect. In the code below the `await` is inside the `expect` rather than before it. When using assertions such as `isVisible()` the test won't wait a single second, it will just check the locator is there and return immediately.

```
// Ø=ÜNexpect(await page.getByText('welcome').isVisible()).toBe(true);
```

Use web first assertions such as `toBeVisible()` instead.

```
// Ø=ÜMawait expect(page.getByText('welcome')).toBeVisible();
```

Configure debugging

Local debugging

For local debugging we recommend you debug your tests live in VSCode. by installing the VS Code extension. You can run tests in debug mode by right clicking on the line next to the test you want to run which will open a browser window and pause at where the breakpoint is set. You can live debug your test by clicking or editing the locators in your test in VS Code which will highlight this locator in the browser window as well as show you any other matching locators found on the page. You can also debug your tests with the Playwright inspector by running your tests with the `--debug` flag. `npx playwright test --debug` You can then step through your test, view actionability logs and edit the locator live and see it highlighted in the browser window. This will show you which locators match, how many of them there are. To debug a specific test add the name of the test file and the line number of the test followed by the `--debug` flag. `npx playwright test example.spec.ts:9 --debug`

Debugging on CI For CI failures, use the Playwright trace viewer instead of videos and screenshots. The trace viewer gives you a full trace of your tests as a local Progressive Web App (PWA) that can easily be shared. With the trace viewer you can view the timeline, inspect DOM snapshots for each action using dev tools, view network requests and more. Traces are configured in the Playwright config file and are set to run on CI on the first retry of a failed test. We don't recommend setting this to on so that traces are run on every test as it's very performance heavy. However you can run a trace locally when developing with the `--trace` flag. `npx playwright test --trace on` Once you run this command your traces will be recorded for each test and can be viewed directly from the HTML report. `npx playwright show-report` Traces can be opened by clicking on the icon next to the test or by opening each of the test reports and scrolling down to the traces section. Use Playwright's Tooling

Playwright comes with a range of tooling to help you write tests. The VS Code extension gives you a great developer experience when writing, running, and debugging tests. The test generator can generate tests and pick locators for you. The trace viewer gives you a full trace of your tests as a local PWA that can easily be shared. With the trace viewer you can view the timeline, inspect DOM snapshots for each action, view network requests and more. Typescript in Playwright works out of the box and gives you better IDE integrations. Your IDE will show you everything you can do and highlight when you do something wrong. No TypeScript experience is needed and it is not necessary for your code to be in TypeScript, all you need to do is create your tests with a `.ts` extension.

Test across all browsers Playwright makes it easy to test your site across all browsers no matter what platform you are on. Testing across all browsers ensures your app works for all users. In your config file you can set up projects adding the name and which browser or device to use. `playwright.config.ts`

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'] },
    },
  ],
});
```

Keep your Playwright dependency up to date By keeping your Playwright version up to date you will be able to test your app on the latest browser versions and catch failures before the latest browser version is released to the public. `npm install -D @playwright/test@latest` Check the release notes to see what the latest version is and what changes have been released. You can see what version of Playwright you have by running the following command. `npx playwright --version`

Run tests on CI Setup CI/CD and run your tests frequently. The more often you run your tests

the better. Ideally you should run your tests on each commit and pull request.

Playwright comes with a GitHub actions workflow so that tests will run on CI for you with no setup required. Playwright can also be setup on the CI environment of your choice. Use Linux when running your tests on CI as it is cheaper. Developers can use whatever environment when running locally but use linux on CI. Lint your tests. Linting the tests helps catching errors early. Use `@typescript-eslint/no-floating-promises` ESLint rule to make sure there are no missing awaits before the asynchronous calls to the Playwright API. Use parallelism and sharding. Playwright runs tests in parallel by default. Tests in a single file are run in order, in the same worker process. If you have many independent tests in a single file, you might want to run them in parallel.

```
import { test } from '@playwright/test';
test.describe.configure({ mode: 'parallel' });
test('runs in parallel 1', async ({ page }) => { /* ... */ });
test('runs in parallel 2', async ({ page }) => { /* ... */ });
```

Playwright can shard a test suite, so that it can be executed on multiple machines.

```
npx playwright test --shard=1/3
```

Productivity tips

Use Soft assertions If your test fails, Playwright will give you an error message showing what part of the test failed which you can see either in VS Code, the terminal, the HTML report, or the trace viewer. However, you can also use soft assertions these do not immediately terminate the test execution, but rather compile and display a list of failed assertions once the test ended.

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

Browsers Each version of Playwright needs specific versions of browser binaries to operate. You will need to use the Playwright CLI to install these browsers. With every release, Playwright updates the versions of the browsers it supports, so that the latest Playwright would support the latest browsers at any moment. It means that every time you update Playwright, you might need to re-run the install CLI command.

Install browsers Playwright can install supported browsers. Running the command without arguments will install the default browsers.

```
npx playwright install
```

You can also install specific browsers by providing an argument:

```
npx playwright install webkit
```

See all supported browsers: `npx playwright install --help`

Install system dependencies System dependencies can get installed automatically. This is useful for CI environments.

```
npx playwright install-deps
```

You can also install the dependencies for a single browser by passing it as an argument:

```
npx playwright install-deps chromium
```

It's also possible to combine `install-deps` with `install` so that the browsers and OS dependencies are installed with a single command.

```
npx playwright install --with-deps chromium
```

See system requirements for officially supported operating systems.

Update Playwright regularly By keeping your Playwright version up to date you will be able to use new features and test your app on the latest browser versions and catch failures before the latest browser version is released to the public.

Update playwright

```
npm install -D @playwright/test@latest
```

Install new browsers

```
npx playwright install
```

Check the release notes to see what the latest version is and what changes have been released.

See what version of Playwright you have by running the following command

```
npx playwright --version
```

Configure Browsers Playwright can run tests on Chromium, WebKit and Firefox browsers as well as branded browsers such as Google Chrome and Microsoft Edge. It can also run on emulated tablet and mobile devices. See the registry of device parameters for a complete list of selected desktop, tablet and mobile devices.

Run tests

on different browsers Playwright can run your tests in multiple browsers and configurations by setting up projects in the config. You can also add different options for each project.

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    /* Test against desktop browsers */
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } },
    /* Test against mobile viewports. */
    { name: 'Mobile Chrome', use: { ...devices['Pixel 5'] } },
    { name: 'Mobile Safari', use: { ...devices['iPhone 12'] } },
    /* Test against branded browsers. */
    { name: 'Google Chrome', use: { ...devices['Desktop Chrome'], channel: 'chrome' }, // or 'chrome-beta' },
    { name: 'Microsoft Edge', use: { ...devices['Desktop Edge'], channel: 'msedge' }, // or 'msedge-dev' },
  ],
});
```

Playwright will run all projects by default.

```
npx playwright test
```

Running 7 tests using 5 workers

```
[chromium] › example.spec.ts:3:1 › basic test (2s)
[firefox] › example.spec.ts:3:1 › basic test (2s)
[webkit] › example.spec.ts:3:1 › basic test (2s)
[Mobile Chrome] › example.spec.ts:3:1 › basic test (2s)
[Mobile Safari] › example.spec.ts:3:1 › basic test (2s)
[Google Chrome] › example.spec.ts:3:1 › basic test (2s)
[Microsoft Edge] › example.spec.ts:3:1 › basic test (2s)
```

Use the `--project` command line option to run a single project.

```
npx playwright test --project=firefox
```

Running 1 test using 1 worker

```
[firefox] › example.spec.ts:3:1 › basic test (2s)
```

The VS Code test runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown from the testing sidebar and choose another profile or modify the default profile by clicking **Select Default Profile** and select the browsers you wish to run your tests on. Choose a specific profile, various profiles or all profiles to run tests on.

Chromium For Google Chrome, Microsoft Edge and other Chromium-based browsers, by default, Playwright uses open source Chromium builds. Since the Chromium project is ahead of the branded browsers, when the world is on Google Chrome N, Playwright already supports Chromium N+1 that will be released in Google Chrome and Microsoft Edge a few weeks later.

Google Chrome & Microsoft Edge While Playwright can download and use the recent Chromium build, it can operate against the branded Google Chrome and Microsoft Edge browsers available on the machine (note that Playwright doesn't install them by default). In particular, the current Playwright version will support Stable and Beta channels of these browsers. Available channels are `chrome`, `msedge`, `chrome-beta`, `msedge-beta` or `msedge-dev`.

Certain Enterprise Browser Policies may impact Playwright's ability to launch and control Google Chrome and Microsoft Edge. Running in an environment with browser policies is outside of the Playwright project's scope.

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    /* Test against branded browsers. */
    { name: 'Google Chrome', use: { ...devices['Desktop Chrome'], channel: 'chrome' }, // or 'chrome-beta' },
    { name: 'Microsoft Edge', use: { ...devices['Desktop Edge'], channel: 'msedge' }, // or 'msedge-beta' or 'msedge-dev' },
  ],
});
```

Installing Google Chrome & Microsoft Edge If Google Chrome or Microsoft Edge is not available on your machine, you can install them using the Playwright command line tool:

```
npx playwright install
```

msedge Google Chrome or Microsoft Edge installations will be installed at the default global location of your operating system overriding your current browser installation. Run with the `--help` option to see a full a list of browsers that can be

installed. When to use Google Chrome & Microsoft Edge and when not to? Defaults

Using the default Playwright configuration with the latest Chromium is a good idea most of the time. Since Playwright is ahead of Stable channels for the browsers, it gives peace of mind that the upcoming Google Chrome or Microsoft Edge releases won't break your site. You catch breakage early and have a lot of time to fix it before the official Chrome update.

Regression testing Having said that, testing policies often require regression testing to be performed against the current publicly available browsers. In this case, you can opt into one of the stable channels, "chrome" or "msedge".

Media codecs Another reason for testing using official binaries is to test functionality related to media codecs. Chromium does not have all the codecs that Google Chrome or Microsoft Edge are bundling due to various licensing considerations and agreements. If your site relies on this kind of codecs (which is rarely the case), you will also want to use the official channel.

Enterprise policy Google Chrome and Microsoft Edge respect enterprise policies, which include limitations to the capabilities, network proxy, mandatory extensions that stand in the way of testing. So if you are part of the organization that uses such policies, it is easiest to use bundled Chromium for your local testing, you can still opt into stable channels on the bots that are typically free of such restrictions.

Firefox Playwright's Firefox version matches the recent Firefox Stable build. Playwright doesn't work with the branded version of Firefox since it relies on patches. Instead you can test against the recent Firefox Stable build.

WebKit Playwright's WebKit version matches the recent WebKit trunk build, before it is used in Apple Safari and other WebKit-based browsers. This gives a lot of lead time to react on the potential browser update issues. Playwright doesn't work with the branded version of Safari since it relies on patches. Instead you can test against the recent WebKit build.

Install behind a firewall or a proxy By default, Playwright downloads browsers from Microsoft's CDN. Sometimes companies maintain an internal proxy that blocks direct access to the public resources. In this case, Playwright can be configured to download browsers via a proxy server.

BashPowerShellBatch# For Playwright

```
Test HTTPS_PROXY=https://192.0.2.1 npx playwright install# For Playwright
Library HTTPS_PROXY=https://192.0.2.1 npm install playwright# For Playwright
Test $Env:HTTPS_PROXY="https://192.0.2.1" npx playwright install# For Playwright
Library $Env:HTTPS_PROXY="https://192.0.2.1" npm install playwright# For Playwright
Test set HTTPS_PROXY=https://192.0.2.1 npx playwright install# For Playwright
Library set HTTPS_PROXY=https://192.0.2.1 npm install playwright
```

If the requests of the proxy get intercepted with a custom untrusted certificate authority (CA) and it yields to Error: self signed certificate in certificate chain while downloading the browsers, you must set your custom root certificates via the `NODE_EXTRA_CA_CERTS` environment variable before installing the browsers:

BashPowerShellBatch

```
export
NODE_EXTRA_CA_CERTS="/path/to/cert.pem" $Env:NODE_EXTRA_CA_CERTS="C:\certs\root.crt"
set NODE_EXTRA_CA_CERTS="C:\certs\root.crt"
```

If your network is slow to connect to Playwright browser archive, you can increase the connection timeout in milliseconds with `PLAYWRIGHT_DOWNLOAD_CONNECTION_TIMEOUT` environment variable:

BashPowerShellBatch

```
PLAYWRIGHT_DOWNLOAD_CONNECTION_TIMEOUT=120000 npx playwright
install $Env:PLAYWRIGHT_DOWNLOAD_CONNECTION_TIMEOUT="120000" npx
playwright install
```

PLAYWRIGHT_DOWNLOAD_CONNECTION_TIMEOUT=120000 npx playwright installDownload from artifact repository By default, Playwright downloads browsers from Microsoft's CDN. Sometimes companies maintain an internal artifact repository to host browser binaries. In this case, Playwright can be configured to download from a custom location using the PLAYWRIGHT_DOWNLOAD_HOST env variable.

BashPowerShellBatch# For Playwright
TestPLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npx playwright install# For Playwright LibraryPLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npm install playwright# For Playwright
Test\$Env:PLAYWRIGHT_DOWNLOAD_HOST="192.0.2.1" npx playwright install# For Playwright Library\$Env:PLAYWRIGHT_DOWNLOAD_HOST="192.0.2.1" npm install playwright# For Playwright Testset PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npx playwright install# For Playwright Libraryset PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npm install playwrightIt is also possible to use a per-browser download hosts using PLAYWRIGHT_CHROMIUM_DOWNLOAD_HOST, PLAYWRIGHT_FIREFOX_DOWNLOAD_HOST and PLAYWRIGHT_WEBKIT_DOWNLOAD_HOST env variables that take precedence over PLAYWRIGHT_DOWNLOAD_HOST.

BashPowerShellBatch# For Playwright
TestPLAYWRIGHT_FIREFOX_DOWNLOAD_HOST=203.0.113.3
PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npx playwright install# For Playwright LibraryPLAYWRIGHT_FIREFOX_DOWNLOAD_HOST=203.0.113.3
PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npm install playwright# For Playwright Test\$Env:PLAYWRIGHT_FIREFOX_DOWNLOAD_HOST="203.0.113.3"\$Env:PLAYWRIGHT_DOWNLOAD_HOST="192.0.2.1" npx playwright install# For Playwright Library\$Env:PLAYWRIGHT_FIREFOX_DOWNLOAD_HOST="203.0.113.3"\$Env:PLAYWRIGHT_DOWNLOAD_HOST="192.0.2.1" npm install playwright# For Playwright Testset PLAYWRIGHT_FIREFOX_DOWNLOAD_HOST=203.0.113.3set PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npx playwright install# For Playwright Libraryset PLAYWRIGHT_FIREFOX_DOWNLOAD_HOST=203.0.113.3set PLAYWRIGHT_DOWNLOAD_HOST=192.0.2.1 npm install playwright

Managing browser binaries Playwright downloads Chromium, WebKit and Firefox browsers into the OS-specific cache folders: %USERPROFILE%\AppData\Local\ms-playwright on Windows ~\Library\Caches/ms-playwright on MacOS ~/.cache/ms-playwright on Linux. These browsers will take a few hundred megabytes of disk space when installed: du -hs ~/Library/Caches/ms-playwright/*281M chromium-XXXXXX187M firefox-XXXX180M webkit-XXXXX. You can override default behavior using environment variables. When installing Playwright, ask it to download browsers into a specific location:

BashPowerShellBatchPLAYWRIGHT_BROWSERS_PATH=\$HOME/pw-browsers npx playwright install\$Env:PLAYWRIGHT_BROWSERS_PATH="\$Env:USERPROFILE\pw-browsers" npx playwright installset PLAYWRIGHT_BROWSERS_PATH=%USERPROFILE%\pw-browsers npx playwright install

When running Playwright scripts, ask it to search for browsers in a shared location.

BashPowerShellBatchPLAYWRIGHT_BROWSERS_PATH=\$HOME/pw-browsers npx playwright

test\$Env:PLAYWRIGHT_BROWSERS_PATH="\$Env:USERPROFILE\pw-browsers"npx playwright testset PLAYWRIGHT_BROWSERS_PATH=%USERPROFILE%\pw-browsersnpx playwright testPlaywright keeps track of packages that need those browsers and will garbage collect them as you update Playwright to the newer versions.noteDevelopers can opt-in in this mode via exporting PLAYWRIGHT_BROWSERS_PATH=\$HOME/pw-browsers in their .bashrc.Hermetic install You can opt into the hermetic install and place binaries in the local folder:BashPowerShellBatch# Places binaries to node_modules/playwright-core/.local-browsersPLAYWRIGHT_BROWSERS_PATH=0 npx playwright install# Places binaries to node_modules\playwright-core\.local-browsers\$Env:PLAYWRIGHT_BROWSERS_PATH=0npx playwright install# Places binaries to node_modules\playwright-core\.local-browsersset PLAYWRIGHT_BROWSERS_PATH=0npx playwright installnotePLAYWRIGHT_BROWSERS_PATH does not change installation path for Google Chrome and Microsoft Edge.Skip browser downloads In certain cases, it is desired to avoid browser downloads altogether because browser binaries are managed separately.This can be done by setting PLAYWRIGHT_SKIP_BROWSER_DOWNLOAD variable before installation.BashPowerShellBatchPLAYWRIGHT_SKIP_BROWSER_DOWNLOAD=1 npx playwright install\$Env:PLAYWRIGHT_SKIP_BROWSER_DOWNLOAD=1npx playwright installset PLAYWRIGHT_SKIP_BROWSER_DOWNLOAD=1npx playwright installStale browser removal Playwright keeps track of the clients that use its browsers. When there are no more clients that require a particular version of the browser, that version is deleted from the system. That way you can safely use Playwright instances of different versions and at the same time, you don't waste disk space for the browsers that are no longer in use.To opt-out from the unused browser removal, you can set the PLAYWRIGHT_SKIP_BROWSER_GC=1 environment variable.Uninstall browsers This will remove the browsers (chromium, firefox, webkit) of the current Playwright installation:npx playwright uninstallTo remove browsers of other Playwright installations as well, pass --all flag:npx playwright uninstall --all

Chrome extensionsnoteExtensions only work in Chrome / Chromium launched with a persistent context.The following is code for getting a handle to the background page of a Manifest v2 extension whose source is located in ./my-extension:const { chromium } = require('playwright');(async () => { const pathToExtension = require('path').join(__dirname, 'my-extension'); const userDataDir = '/tmp/test-user-data-dir'; const browserContext = await chromium.launchPersistentContext(userDataDir, { headless: false, args: ['--disable-extensions-except=\${pathToExtension}', '--load-extension=\${pathToExtension}'] }); let [backgroundPage] = browserContext.backgroundPages(); if (!backgroundPage) backgroundPage = await browserContext.waitForEvent('backgroundpage'); // Test the background page as you would any other page. await browserContext.close();})();Testing To have the extension loaded when running tests you can use a test fixture to set the context. You can also dynamically retrieve the extension id and use it to load and test the popup page for example.First, add fixtures that will load the extension:fixtures.tsimport { test as base, expect, chromium, type BrowserContext } from '@playwright/test';import path from

```
'path';export const test = base.extend<{ context: BrowserContext; extensionId: string;}>({ context: async ({ }, use) => { const pathToExtension = path.join(__dirname, 'my-extension'); const context = await chromium.launchPersistentContext("", { headless: false, args: [ `--disable-extensions-except=${pathToExtension}`, `--load-extension=${pathToExtension}`, ], }); await use(context); await context.close(); }, extensionId: async ({ context }, use) => { /* // for manifest v2: let [background] = context.backgroundPages() if (!background) background = await context.waitForEvent('backgroundpage') */ // for manifest v3: let [background] = context.serviceWorkers(); if (!background) background = await context.waitForEvent('serviceworker'); const extensionId = background.url().split('/')[2]; await use(extensionId); },});export const expect = test.expect;Then use these fixtures in a test:import { test, expect } from './fixtures';test('example test', async ({ page }) => { await page.goto('https://example.com'); await expect(page.locator('body')).toHaveText('Changed by my-extension');});test('popup page', async ({ page, extensionId }) => { await page.goto(`chrome-extension://${extensionId}/popup.html`); await expect(page.locator('body')).toHaveText('my-extension popup');});
```

Headless mode By default, Chrome's headless mode in Playwright does not support Chrome extensions. To overcome this limitation, you can run Chrome's persistent context with a new headless mode by using the following code:

```
fixtures.ts// ...const pathToExtension = path.join(__dirname, 'my-extension');const context = await chromium.launchPersistentContext("", { headless: false, args: [ `--headless=new`, // the new headless arg for chrome v109+. Use '--headless=chrome' as arg for browsers v94-108. `--disable-extensions-except=${pathToExtension}`, `--load-extension=${pathToExtension}`, ],});// ...
```

Experimental: componentsPlaywright Test can now test your components.Example Here is what a typical component test looks like:

```
test('event should work', async ({ mount }) => { let clicked = false; // Mount a component. Returns locator pointing to the component. const component = await mount( <Button title="Submit" onClick={() => { clicked = true }}></Button> ); // As with any Playwright test, assert locator text. await expect(component).toContainText('Submit'); // Perform locator click. This will trigger the event. await component.click(); // Assert that respective events have been fired. expect(clicked).toBeTruthy();});
```

How to get started Adding Playwright Test to an existing project is easy. Below are the steps to enable Playwright Test for a React, Vue, Svelte or Solid project.

Step 1: Install Playwright Test for components for your respective framework

```
npm install playwright@latest -- --ctyarn create playwright -- ctpnpm dlx create-playwright --ct
```

This step creates several files in your workspace: playwright/index.html This file defines an html file that will be used to render components during testing. It must contain element with id="root", that's where components are mounted. It must also link the script called playwright/index.{js,ts,jsx,tsx}.

```
<html lang="en"> <body> <div id="root"></div> <script type="module" src="./index.ts"></script> </body></html>
```

playwright/index.ts You can include stylesheets, apply theme and inject code into the page where component is mounted using this script. It can be either a .js, .ts, .jsx or .tsx file.

// Apply theme here, add anything your component needs at runtime here.

Step 2. Create a test file src/App.spec.{ts,tsx}

```
ReactSolidSvelteVueimport { test, expect } from '@playwright/experimental-ct-react';import App from './App';test.use({ viewport: { width: 500, height:
```



```

500 } });test('should work', async ({ mount }) => { const component = await
mount(<App />); await expect(component).toContainText('Learn React');});import { test,
expect } from '@playwright/experimental-ct-vue';import App from './
App.vue';test.use({ viewport: { width: 500, height: 500 } });test('should work', async
({ mount }) => { const component = await mount(App); await
expect(component).toContainText('Vite + Vue');});If using TypeScript and Vue make
sure to add a vue.d.ts file to your project:declare module '*.vue';import { test, expect }
from '@playwright/experimental-ct-svelte';import App from './
App.svelte';test.use({ viewport: { width: 500, height: 500 } });test('should work', async
({ mount }) => { const component = await mount(App); await
expect(component).toContainText('Vite + Svelte');});import { test, expect } from
'@playwright/experimental-ct-solid';import App from './App';test.use({ viewport: { width:
500, height: 500 } });test('should work', async ({ mount }) => { const component = await
mount(<App />); await expect(component).toContainText('Learn Solid');});Step 3. Run
the tests You can run tests using the VS Code extension or the command line.npm run
test-ctFurther reading: configure reporting, browsers, tracing Refer to Playwright config
for configuring your project.Hooks You can use beforeMount and afterMount hooks to
configure your app. This lets you setup things like your app router, fake server etc.
giving you the flexibility you need. You can also pass custom configuration from the
mount call from a test, which is accessible from the hooksConfig fixture.playwright/
index.{js,ts,jsx,tsx} This includes any config that needs to be run before or after
mounting the component. An example of configuring a router is provided
below:ReactSolidVue3Vue2playwright/index.tsximport { beforeMount, afterMount } from
'@playwright/experimental-ct-react/hooks';import { BrowserRouter } from 'react-router-
dom';export type HooksConfig = { enableRouting?: boolean;}
beforeMount<HooksConfig>(async ({ App, hooksConfig }) => { if
(hooksConfig?.enableRouting) return <BrowserRouter><App /></
BrowserRouter>});In your test file: src/pages/ProductsPage.spec.tsximport { test,
expect } from '@playwright/experimental-ct-react';import type { HooksConfig } from
'@playwright/test';import { ProductsPage } from './pages/ProductsPage';test('configure
routing through hooks config', async ({ page, mount }) => { const component = await
mount<HooksConfig>(<ProductsPage />, { hooksConfig: { enableRouting: true }, });
await expect(component.getByRole('link')).toHaveAttribute('href', '/
products/42');});playwright/index.tsximport { beforeMount, afterMount } from
'@playwright/experimental-ct-solid/hooks';import { Router } from '@solidjs/router';export
type HooksConfig = { enableRouting?: boolean;}beforeMount<HooksConfig>(async
({ App, hooksConfig }) => { if (hooksConfig?.enableRouting) return <Router><App /
></Router>});In your test file: src/pages/ProductsPage.spec.tsximport { test, expect }
from '@playwright/experimental-ct-solid';import type { HooksConfig } from '@playwright/
test';import { ProductsPage } from './pages/ProductsPage';test('configure routing
through hooks config', async ({ page, mount }) => { const component = await
mount<HooksConfig>(<ProductsPage />, { hooksConfig: { enableRouting: true }, });
await expect(component.getByRole('link')).toHaveAttribute('href', '/
products/42');});playwright/index.tsimport { beforeMount, afterMount } from '@playwright/
experimental-ct-vue/hooks';import { router } from './src/router';export type HooksConfig
= { enableRouting?: boolean;}beforeMount<HooksConfig>(async ({ app,

```

```

hooksConfig }) => { if (hooksConfig?.enableRouting) app.use(router);});In your test
file: src/pages/ProductsPage.spec.tsimport { test, expect } from '@playwright/
experimental-ct-vue';import type { HooksConfig } from '@playwright/test';import
ProductsPage from './pages/ProductsPage.vue';test('configure routing through hooks
config', async ({ page, mount }) => { const component = await
mount<HooksConfig>(ProductsPage, { hooksConfig: { enableRouting: true }, });
await expect(component.getByRole('link')).toHaveAttribute('href', '/
products/42');});playwright/index.tsimport { beforeMount, afterMount } from '@playwright/
experimental-ct-vue2/hooks';import Router from 'vue-router';import { router } from '../src/
router';export type HooksConfig = { enableRouting?: boolean;}
beforeMount<HooksConfig>(async ({ app, hooksConfig }) => { if
(hooksConfig?.enableRouting) { Vue.use(Router); return { router } });});In your test
file: src/pages/ProductsPage.spec.tsimport { test, expect } from '@playwright/
experimental-ct-vue2';import type { HooksConfig } from '@playwright/test';import
ProductsPage from './pages/ProductsPage.vue';test('configure routing through hooks
config', async ({ page, mount }) => { const component = await
mount<HooksConfig>(ProductsPage, { hooksConfig: { enableRouting: true }, });
await expect(component.getByRole('link')).toHaveAttribute('href', '/
products/42');});Under the hood When Playwright Test is used to test web components,
tests run in Node.js, while components run in the real browser. This brings together the
best of both worlds: components run in the real browser environment, real clicks are
triggered, real layout is executed, visual regression is possible. At the same time, test
can use all the powers of Node.js as well as all the Playwright Test features. As a result,
the same parallel, parametrized tests with the same post-mortem Tracing story are
available during component testing. Here is how this is achieved: Once the tests are
executed, Playwright creates a list of components that the tests need. It then compiles a
bundle that includes these components and serves it using a local static web
server. Upon the mount call within the test, Playwright navigates to the facade page /
playwright/index.html of this bundle and tells it to render the component. Events are
marshalled back to the Node.js environment to allow verification. Playwright is using Vite
to create the components bundle and serve it. Known issues and limitations Q) I can't
import anything other than the components from TSX/JSX/Component files As per
above, you can only import your components from your test file. If you have utility
methods or constants in your TSX files, it is advised to extract them into the TS files
and import those utility methods and constants from your component files and from
your test files. That allows us to not load any of the component code in the Node-based
test runner and keep Playwright fast at executing your tests. Q) I have a project that
already uses Vite. Can I reuse the config? At this point, Playwright is bundler-agnostic,
so it is not reusing your existing Vite config. Your config might have a lot of things we
won't be able to reuse. So for now, you would copy your path mappings and other high
level settings into the ctViteConfig property of Playwright config.import { defineConfig }
from '@playwright/experimental-ct-react';export default defineConfig({ use:
{ ctViteConfig: { // ... }, },});Q) What's the difference between @playwright/test
and @playwright/experimental-ct-{react,svelte,vue,solid}? test('...', async ({ mount,
page, context }) => { // ...});@playwright/experimental-ct-{react,svelte,vue,solid} wrap
@playwright/test to provide an additional built-in component-testing specific fixture

```

called mount:ReactSolidSvelteVueimport { test, expect } from '@playwright/experimental-ct-react';import HelloWorld from './HelloWorld';test.use({ viewport: { width: 500, height: 500 } });test('should work', async ({ mount }) => { const component = await mount(<HelloWorld msg="greetings" />); await expect(component).toContainText('Greetings');});import { test, expect } from '@playwright/experimental-ct-vue';import HelloWorld from './HelloWorld.vue';test.use({ viewport: { width: 500, height: 500 } });test('should work', async ({ mount }) => { const component = await mount(HelloWorld, { props: { msg: 'Greetings', }, }); await expect(component).toContainText('Greetings');});import { test, expect } from '@playwright/experimental-ct-svelte';import HelloWorld from './HelloWorld.svelte';test.use({ viewport: { width: 500, height: 500 } });test('should work', async ({ mount }) => { const component = await mount(HelloWorld, { props: { msg: 'Greetings', }, }); await expect(component).toContainText('Greetings');});import { test, expect } from '@playwright/experimental-ct-solid';import HelloWorld from './HelloWorld';test.use({ viewport: { width: 500, height: 500 } });test('should work', async ({ mount }) => { const component = await mount(<HelloWorld msg="greetings" />); await expect(component).toContainText('Greetings');});

Additionally, it adds some config options you can use in your playwright-ct.config.{ts,js}. Finally, under the hood, each test re-uses the context and page fixture as a speed optimization for Component Testing. It resets them in between each test so it should be functionally equivalent to @playwright/test's guarantee that you get a new, isolated context and page fixture per-test.

Q) Can I use @playwright/test and @playwright/experimental-ct-{react,svelte,vue,solid}? Yes. Use a Playwright Config for each and follow their respective guides (E2E Playwright Test, Component Tests).

Q) Why can't I pass a variable to mount? This is a known issue. The following pattern does not work:const app = <App />;await mount(app);results inundefined: TypeError: Cannot read properties of undefined (reading 'map')while this works:await mount(<App />);

Q) How can I use Vite plugins? You can specify plugins via Vite config for testing settings. Note that once you start specifying plugins, you are responsible for specifying the framework plugin as well, vue() in this case:import { defineConfig, devices } from '@playwright/experimental-ct-vue';import { resolve } from 'path';import vue from '@vitejs/plugin-vue';import AutoImport from 'unplugin-auto-import/vite';import Components from 'unplugin-vue-components/vite';export default defineConfig({ testDir: './tests/component', use: { trace: 'on-first-retry', ctViteConfig: { plugins: [vue(), AutoImport({ imports: ['vue', 'vue-router', '@vueuse/head', 'pinia', { '@/store': ['useStore'], },], dts: 'src/auto-imports.d.ts', eslint: { enabled: true, }, }, Components({ dirs: ['src/components'], extensions: ['vue'], },), resolve: { alias: { '@': resolve(__dirname, './src'), }, }, }, });

Q) how can i test components that uses Pinia? Pinia needs to be initialized in playwright/index.{js,ts,jsx,tsx}. If you do this inside a beforeMount hook, the initialState can be overwritten on a per-test basis:playwright/index.tsimport { beforeMount, afterMount } from '@playwright/experimental-ct-vue/hooks';import { createTestingPinia } from '@pinia/testing';import type { StoreState } from 'pinia';import type { useStore } from '../src/store';export type HooksConfig = { store?: StoreState<ReturnType<typeof useStore>>};beforeMount<HooksConfig>(async ({ hooksConfig }) => { createTestingPinia({ initialState: hooksConfig?.store, /** *

Use http intercepting to mock api calls instead: `* https://playwright.dev/docs/mock#mock-api-requests */ stubActions: false, createSpy(args) { console.log('spy', args) return () => console.log('spy-returns') }, });});` In your test file: `src/pinia.spec.ts` `import { test, expect } from '@playwright/experimental-ct-vue'; import type { HooksConfig } from '@playwright/test'; import Store from './Store.vue'; test('override initialState', async ({ mount }) => { const component = await mount<HooksConfig>(Store, { hooksConfig: { store: { name: 'override initialState' } } }); await expect(component).toContainText('override initialState'); });`

Debugging Tests VS Code debugger We recommend using the VS Code Extension for debugging for a better developer experience. With the VS Code extension you can debug your tests right in VS Code, see error messages, set breakpoints and step through your tests.

Error Messages If your test fails VS Code will show you error messages right in the editor showing what was expected, what was received as well as a complete call log.

Live Debugging You can debug your test live in VS Code. After running a test with the Show Browser option checked, click on any of the locators in VS Code and it will be highlighted in the Browser window. Playwright will also show you if there are multiple matches. You can also edit the locators in VS Code and Playwright will show you the changes live in the browser window.

Picking a Locator Pick a locator and copy it into your test file by clicking the Pick locator button from the testing sidebar. Then in the browser click the element you require and it will now show up in the Pick locator box in VS Code. Press 'enter' on your keyboard to copy the locator into the clipboard and then paste anywhere in your code. Or press 'escape' if you want to cancel. Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If Playwright finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.

Run in Debug Mode To set a breakpoint click next to the line number where you want the breakpoint to be until a red dot appears. Run the tests in debug mode by right clicking on the line next to the test you want to run. A browser window will open and the test will run and pause at where the breakpoint is set. You can step through the tests, pause the test and rerun the tests from the menu in VS Code.

Debug in different Browsers By default debugging is done using the Chromium profile. You can debug your tests on different browsers by right clicking on the debug icon in the testing sidebar and clicking on the 'Select Default Profile' option from the dropdown. Then choose the test profile you would like to use for debugging your tests. Each time you run your test in debug mode it will use the profile you selected. You can run tests in debug mode by right clicking the line number where your test is and selecting 'Debug Test' from the menu. To learn more about debugging, see [Debugging in Visual Studio Code](#).

Playwright Inspector The Playwright Inspector is a GUI tool to help you debug your Playwright tests. It allows you to step through your tests, live edit locators, pick locators and see actionability logs.

Run in debug mode Run your tests with the `--debug` flag to open the inspector. This configures Playwright for debugging and opens the inspector. Additional useful defaults are configured when `--debug` is used: Browsers launch in headed mode Default timeout is set to 0 (= no timeout) Debug all tests on all browsers To debug all tests run the test command with the `--debug` flag. This will run tests one by one, and open the inspector and a browser window for each test.

`npx playwright test --debug` Debug one test on all browsers To

debug one test on a specific line run the test command followed by the name of the test file and the line number of the test you want to debug, followed by the --debug flag. This will run a single test in each browser configured in your playwright.config and open the inspector.`npx playwright test example.spec.ts:10 --debug`Debug on a specific browser In Playwright you can configure projects in your playwright.config. Once configured you can then debug your tests on a specific browser or mobile viewport using the --project flag followed by the name of the project configured in your playwright.config.`npx playwright test --project=chromium --debug``npx playwright test --project="Mobile Safari" --debug``npx playwright test --project="Microsoft Edge" --debug`Debug one test on a specific browser To run one test on a specific browser add the name of the test file and the line number of the test you want to debug as well as the --project flag followed by the name of the project.`npx playwright test example.spec.ts:10 --project=webkit --debug`Stepping through your tests You can play, pause or step through each action of your test using the toolbar at the top of the Inspector. You can see the current action highlighted in the test code, and matching elements highlighted in the browser window.Run a test from a specific breakpoint To speed up the debugging process you can add a `page.pause()` method to your test. This way you won't have to step through each action of your test to get to the point where you want to debug.`await page.pause();`Once you add a `page.pause()` call, run your tests in debug mode. Clicking the "Resume" button in the Inspector will run the test and only stop on the `page.pause()`.Live editing locators While running in debug mode you can live edit the locators. Next to the 'Pick Locator' button there is a field showing the locator that the test is paused on. You can edit this locator directly in the Pick Locator field, and matching elements will be highlighted in the browser window.Picking locators While debugging you might need to choose a more resilient locator. You can do this by clicking on the Pick Locator button and hovering over any element in the browser window. While hovering over an element you will see the code needed to locate this element highlighted below. Clicking an element in the browser will add the locator into the field where you can then either tweak it or copy it into your code.Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If Playwright finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.Actionability logs By the time Playwright has paused on a click action, it has already performed actionability checks that can be found in the log. This can help you understand what happened during your test and what Playwright did or tried to do. The log tells you if the element was visible, enabled and stable, if the locator resolved to an element, scrolled into view, and so much more. If actionability can't be reached, it will show the action as pending.Trace Viewer Playwright Trace Viewer is a GUI tool that lets you explore recorded Playwright traces of your tests. You can go back and forward through each action on the left side, and visually see what was happening during the action. In the middle of the screen, you can see a DOM snapshot for the action. On the right side you can see action details, such as time, parameters, return value and log. You can also explore console messages, network requests and the source code. Your browser does not support the video tag.To learn more about how to record traces and use the Trace Viewer, check out the Trace Viewer guide.Browser Developer Tools When running in Debug Mode with

PWDEBUG=console, a playwright object is available in the Developer tools console. Developer tools can help you to: Inspect the DOM tree and find element selectors See console logs during execution (or learn how to read logs via API) Check network activity and other developer tools features To debug your tests using the browser developer tools start by setting a breakpoint in your test to pause the execution using the `page.pause()` method. `await page.pause();` Once you have set a breakpoint in your test you can then run your test with

```
PWDEBUG=console.BashPowerShellBatchPWDEBUG=console npx playwright
test$env:PWDEBUG="console"npx playwright testset PWDEBUG=consolenpx
playwright testOnce Playwright launches the browser window you can open the
developer tools. The playwright object will be available in the console panel.playwright.
$(selector) Query the Playwright selector, using the actual Playwright query engine, for
example:playwright.$('.auth-form >> text=Log in');<button>Log in</button>playwright.$
$(selector) Same as playwright.$, but returns all matching elements.playwright.$$('li >>
text=John')[<li>, <li>, <li>, <li>]playwright.inspect(selector) Reveal element in the
Elements panel.playwright.inspect('text=Log in')playwright.locator(selector) Create a
locator and query matching elements, for example:playwright.locator('.auth-form',
{ hasText: 'Log in' });Locator () - element: button - elements:
```

```
[button]playwright.highlight(selector) Highlight the first occurrence of the
locator:playwright.highlight('.auth-form');playwright.clear() playwright.clear()Clear
existing highlights.playwright.selector(element) Generates selector for the given
element. For example, select an element in the Elements panel and pass
$0:playwright.selector($0)"div[id="glow-ingress-block"] >> text=/.*/"Verbose API
logs Playwright supports verbose logging with the DEBUG environment
variable.BashPowerShellBatchDEBUG=pw:api npx playwright
```

```
test$env:DEBUG="pw:api"npx playwright testset DEBUG=pw:apinpx playwright
testnoteFor WebKit: launching WebKit Inspector during the execution will prevent the
Playwright script from executing any further and will reset pre-configured user agent
and device emulation.Headless mode Playwright runs browsers in headless mode by
default. To change this behavior, use headless: false as a launch option. You can also
use the slowMo option to slow down execution (by N milliseconds per operation) and
follow along while debugging.// Chromium, Firefox, or WebKitawait
chromium.launch({ headless: false, slowMo: 100 });
```

Dialogs Playwright can interact with the web page dialogs such as `alert`, `confirm`, `prompt` as well as `beforeunload` confirmation. `alert()`, `confirm()`, `prompt()` dialogs By default, dialogs are auto-dismissed by Playwright, so you don't have to handle them. However, you can register a dialog handler before the action that triggers the dialog to either `dialog.accept()` or `dialog.dismiss()` it. `page.on('dialog', dialog => dialog.accept());await page.getByRole('button').click();` `notpage.on('dialog')` listener must handle the dialog. Otherwise your action will stall, be it `locator.click()` or something else. That's because dialogs in Web are modals and therefore block further page execution until they are handled. As a result, the following snippet will never resolve: **dangerWRONG!**

```
page.on('dialog', dialog => console.log(dialog.message()));await
page.getByRole('button').click(); // Will hang here
```

`notIf there is no listener for page.on('dialog'), all dialogs are automatically dismissed.beforeunload` dialog When `page.close()` is invoked with the truthy `runBeforeUnload` value, the page runs its unload

handlers. This is the only case when `page.close()` does not wait for the page to actually close, because it might be that the page stays open in the end of the operation. You can register a dialog handler to handle the `beforeunload` dialog yourself: `page.on('dialog', async dialog => { assert(dialog.type() === 'beforeunload'); await dialog.dismiss();});await page.close({ runBeforeUnload: true });`

Downloads For every attachment downloaded by the page, `page.on('download')` event is emitted. All these attachments are downloaded into a temporary folder. You can obtain the download url, file system path and payload stream using the `Download` object from the event. You can specify where to persist downloaded files using the `downloadsPath` option in `browserType.launch().noteDownloaded` files are deleted when the browser context that produced them is closed. Here is the simplest way to handle the file download: `// Start waiting for download before clicking. Note no await.const downloadPromise = page.waitForEvent('download');await page.getByText('Download file').click();const download = await downloadPromise;// Wait for the download process to completeconsole.log(await download.path());// Save downloaded file somewhereawait download.saveAs('/path/to/save/download/at.txt');` **Variations** If you have no idea what initiates the download, you can still handle the event: `page.on('download', download => download.path().then(console.log));` Note that handling the event forks the control flow and makes the script harder to follow. Your scenario might end while you are downloading a file since your main control flow is not awaiting for this operation to resolve. **noteFor** uploading files, see the uploading files section.

Evaluating JavaScript Playwright scripts run in your Playwright environment. Your page scripts run in the browser page environment. Those environments don't intersect, they are running in different virtual machines in different processes and even potentially on different computers. The `page.evaluate()` API can run a JavaScript function in the context of the web page and bring results back to the Playwright environment. Browser globals like `window` and `document` can be used in `evaluate`. `const href = await page.evaluate(() => document.location.href);` If the result is a `Promise` or if the function is asynchronous `evaluate` will automatically wait until it's resolved: `const status = await page.evaluate(async () => { const response = await fetch(location.href); return response.status;});` **Evaluation Argument** Playwright evaluation methods like `page.evaluate()` take a single optional argument. This argument can be a mix of `Serializable` values and `JSHandle` or `ElementHandle` instances. Handles are automatically converted to the value they represent. `// A primitive value.await page.evaluate(num => num, 42);` `// An array.await page.evaluate(array => array.length, [1, 2, 3]);` `// An object.await page.evaluate(object => object.foo, { foo: 'bar' });` `// A single handle.const button = await page.evaluate('window.button');` `await page.evaluate(button => button.textContent, button);` `// Alternative notation using elementHandle.evaluate.await button.evaluate((button, from) => button.textContent.substring(from), 5);` `// Object with multiple handles.const button1 = await page.evaluate('window.button1');` `const button2 = await page.evaluate('window.button2');` `await page.evaluate(o => o.button1.textContent + o.button2.textContent, { button1, button2 });` `// Object destructuring works. Note that property names must match// between the destructured object and the argument.// Also note the required parenthesis.await page.evaluate(({ button1, button2 }) => button1.textContent + button2.textContent, { button1, button2 });` `// Array works as well.`

Arbitrary names can be used for destructuring.// Note the required parenthesis.`await page.evaluate(([b1, b2]) => b1.textContent + b2.textContent, [button1, button2]);`// Any non-cyclic mix of serializables and handles works.`await page.evaluate(x => x.button1.textContent + x.list[0].textContent + String(x.foo), { button1, list: [button2], foo: null });`Right:`const data = { text: 'some data', value: 1 };`// Pass `|data|` as a parameter.`const result = await page.evaluate(data => { window.myApp.use(data); }, data);`Wrong:`const data = { text: 'some data', value: 1 };const result = await page.evaluate(() => { // There is no |data| in the web page. window.myApp.use(data); });`

EventsPlaywright allows listening to various types of events happening on the web page, such as network requests, creation of child pages, dedicated workers etc. There are several ways to subscribe to such events such as waiting for events or adding or removing event listeners.Waiting for event Most of the time, scripts will need to wait for a particular event to happen. Below are some of the typical event awaiting patterns.Wait for a request with the specified url using `page.waitForRequest()`// Start waiting for request before goto. Note no `await`.`const requestPromise = page.waitForRequest('**/*logo*.png');``await page.goto('https://wikipedia.org');``const request = await requestPromise;``console.log(request.url());`Wait for popup window:// Start waiting for popup before clicking. Note no `await`.`const popupPromise = page.waitForEvent('popup');``await page.getByText('open the popup').click();``const popup = await popupPromise;``await popup.goto('https://wikipedia.org');`Adding/removing event listener 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:`page.on('request', request => console.log(' Request sent: ${request.url()} '));``const listener = request => console.log(' Request finished: ${request.url()} ');``page.on('requestfinished', listener);``await page.goto('https://wikipedia.org');``page.off('requestfinished', listener);``await page.goto('https://www.openstreetmap.org/');`Adding one-off listeners If a certain event needs to be handled once, there is a convenience API for that:`page.once('dialog', dialog => dialog.accept('2021'));``await page.evaluate("prompt('Enter a number:');");`

ExtensibilityCustom selector enginesCustom selector engines Playwright supports custom selector engines, registered with `selectors.register()`.Selector engine should have the following properties:create function to create a relative selector from root (root is either a Document, ShadowRoot or Element) to a target element.query function to query first element matching selector relative to the root.queryAll function to query all elements matching selector relative to the root.By default the engine is run directly in the frame's JavaScript context and, for example, can call an application-defined function. To isolate the engine from any JavaScript in the frame, but leave access to the DOM, register the engine with `{contentScript: true}` option. Content script engine is safer because it is protected from any tampering with the global objects, for example altering `Node.prototype` methods. All built-in selector engines run as content scripts. Note that running as a content script is not guaranteed when the engine is used together with other custom engines.Selectors must be registered before creating the page.An example of registering selector engine that queries elements based on a tag name:`import { test, expect } from '@playwright/test';`// Must be a function that evaluates to a selector engine instance.`const createTagNameEngine = () => ({ // Returns the first`


```

element matching given selector in the root's subtree. query(root, selector) { return
root.querySelector(selector); }, // Returns all elements matching given selector in the
root's subtree. queryAll(root, selector) { return
Array.from(root.querySelectorAll(selector)); }}; // Register selectors before any page is
created.test.beforeAll(async ({ playwright }) => { // Register the engine. Selectors will
be prefixed with "tag=". await playwright.selectors.register('tag',
createTagNameEngine));});test('selector engine test', async ({ page }) => { // Now we
can use 'tag=' selectors. const button = page.locator('tag=button'); await
button.click(); // We can combine it with built-in locators. await
page.locator('tag=div').getByText('Click me').click(); // We can use it in any methods
supporting selectors. await expect(page.locator('tag=button')).toHaveCount(3));});
FramesA Page can have one or more Frame objects attached to it. Each page has a
main frame and page-level interactions (like click) are assumed to operate in the main
frame.A page can have additional frames attached with the iframe HTML tag. These
frames can be accessed for interactions inside the frame.// Locate element inside
frameconst username = await page.frameLocator('.frame-class').getByLabel('User
Name');await username.fill('John');Frame objects One can access frame objects using
the page.frame() API:// Get frame using the frame's name attributeconst frame =
page.frame('frame-login');// Get frame using frame's URLconst frame =
page.frame({ url: /.*/ });// Interact with the frameawait frame.fill('#username-
input', 'John');

```

HandlesPlaywright can create handles to the page DOM elements or any other objects inside the page. These handles live in the Playwright process, whereas the actual objects live in the browser. There are two types of handles:JSHandle to reference any JavaScript objects in the pageElementHandle to reference DOM elements in the page, it has extra methods that allow performing actions on the elements and asserting their properties.Since any DOM element in the page is also a JavaScript object, any ElementHandle is a JSHandle as well.Handles are used to perform operations on those actual objects in the page. You can evaluate on a handle, get handle properties, pass handle as an evaluation parameter, serialize page object into JSON etc. See the JSHandle class API for these and methods.API reference JSHandleElementHandleHere is the easiest way to obtain a JSHandle.const jsHandle = await page.evaluateHandle('window');// Use jsHandle for evaluations.Element Handles DiscouragedThe use of ElementHandle is discouraged, use Locator objects and web-first assertions instead.When ElementHandle is required, it is recommended to fetch it with the page.waitForSelector() or frame.waitForSelector() methods. These APIs wait for the element to be attached and visible.// Get the element handleconst elementHandle = page.waitForSelector('#box');// Assert bounding box for the elementconst boundingBox = await elementHandle.boundingBox();expect(boundingBox.width).toBe(100);// Assert attribute for the elementconst classNames = await elementHandle.getAttribute('class');expect(classNames.includes('highlighted')).toBeTruthy());Handles as parameters Handles can be passed into the page.evaluate() and similar methods. The following snippet creates a new array in the page, initializes it with data and returns a handle to this array into Playwright. It then uses the handle in subsequent evaluations:// Create new array in page.const myArrayHandle = await page.evaluateHandle(() => { window.myArray = [1];

```

return myArray;});// Get the length of the array.const length = await page.evaluate(a =>
a.length, myArrayHandle);// Add one more element to the array using the handleawait
page.evaluate(arg => arg.myArray.push(arg.newElement), { myArray: myArrayHandle,
newElement: 2});// Release the object when it's no longer needed.await
myArrayHandle.dispose();//Handle Lifecycle Handles can be acquired using the page
methods such as page.evaluateHandle(), page.$() or page.$$() or their frame
counterparts frame.evaluateHandle(), frame.$() or frame.$$. Once created, handles
will retain object from garbage collection unless page navigates or the handle is
manually disposed via the jsHandle.dispose() method.API reference JSHandleElementH
andleelementHandle.boundingBox()elementHandle.getAttribute()elementHandle.innerTe
xt()elementHandle.innerHTML()elementHandle.textContent()jsHandle.evaluate()page.ev
aluatHandle()page.$()page.$$()Locator vs ElementHandle cautionWe only recommend
using ElementHandle in the rare cases when you need to perform extensive DOM
traversal on a static page. For all user actions and assertions use locator instead.The
difference between the Locator and 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 stale DOM element. This can lead to unexpected
behaviors.const handle = await page.$('text=Submit');// ...await handle.hover();await
handle.click();//With the locator, every time the locator 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.getByText('Submit');// ...await
locator.hover();await locator.click();

```

IsolationTests written with Playwright execute in isolated clean-slate environments called browser contexts. This isolation model improves reproducibility and prevents cascading test failures. What is Test Isolation? Test Isolation is when each test is completely isolated from another test. Every test runs independently from any other test. This means that each test has it's own local storage, session storage, cookies etc. Playwright achieves this using BrowserContexts which are equivalent to incognito-like profiles. They are fast and cheap to create and are completely isolated, even when running in a single browser. Playwright creates a context for each test, and provides a default Page in that context. Why is Test Isolation Important? No failure carry-over. If one test fails it doesn't affect the other test. Easy to debug errors or flakiness, because you can run just a single test as many times as you'd like. Don't have to think about the order when running in parallel, sharding, etc. Two Ways of Test Isolation There are two different strategies when it comes to Test Isolation: start from scratch or cleanup in between. The problem with cleaning up in between tests is that it can be easy to forget to clean up and some things are impossible to clean up such as "visited links". State from one test can leak into the next test which could cause your test to fail and make debugging harder as the problem comes from another test. Starting from scratch means everything is new, so if the test fails you only have to look within that test to debug. How Playwright Achieves Test Isolation Playwright uses browser contexts to achieve Test Isolation. Each test has it's own Browser Context. Running the test creates a new browser context each time. When using Playwright as a Test Runner, browser contexts are created by default. Otherwise, you can create browser contexts

manually. TestLibraryimport { test } from '@playwright/test';test('example test', async ({ page, context }) => { // "context" is an isolated BrowserContext, created for this specific test. // "page" belongs to this context.});test('another test', async ({ page, context }) => { // "context" and "page" in this second test are completely // isolated from the first test.});const browser = await chromium.launch();const context = await browser.newContext();const page = await context.newPage();Browser contexts can also be used to emulate multi-page scenarios involving mobile devices, permissions, locale and color scheme. Check out our Emulation guide for more details. Multiple Contexts in a Single Test Playwright can create multiple browser contexts within a single scenario.

This is useful when you want to test for multi-user functionality, like a chat. TestLibraryimport { test } from '@playwright/test';test('admin and user', async ({ browser }) => { // Create two isolated browser contexts const adminContext = await browser.newContext(); const userContext = await browser.newContext(); // Create pages and interact with contexts independently const adminPage = await adminContext.newPage(); const userPage = await userContext.newPage();});const { chromium } = require('playwright');// Create a Chromium browser instanceconst browser = await chromium.launch();// Create two isolated browser contextsconst userContext = await browser.newContext();const adminContext = await browser.newContext();// Create pages and interact with contexts independentlyconst adminPage = await adminContext.newPage();const userPage = await userContext.newPage();

LocatorsLocators are the central piece of Playwright's auto-waiting and retry-ability. In a nutshell, locators represent a way to find element(s) on the page at any moment. Quick Guide These are the recommended built in locators.page.getByRole() to locate by explicit and implicit accessibility attributes.page.getByText() to locate by text content.page.getByLabel() to locate a form control by associated label's text.page.getByPlaceholder() to locate an input by placeholder.page.getByAltText() to locate an element, usually image, by its text alternative.page.getByTitle() to locate an element by its title attribute.page.getByTestId() to locate an element based on its data-testid attribute (other attributes can be configured).await page.getByLabel('User Name').fill('John');await page.getByLabel('Password').fill('secret-password');await page.getByRole('button', { name: 'Sign in' }).click();await expect(page.getByText('Welcome, John!')).toBeVisible();Locating elements Playwright comes with multiple built-in locators. To make tests resilient, we recommend prioritizing user-facing attributes and explicit contracts such as page.getByRole().For example, consider the following DOM structure.http://localhost:3000Sign in<button>Sign in</button>Locate the element by its role of button with name "Sign in".await page.getByRole('button', { name: 'Sign in' }).click();tipUse the code generator to generate a locator, and then edit it as you'd like. Every time a locator is used for an action, an up-to-date DOM element is located in the page. In the snippet below, the underlying DOM element will be located twice, once prior to every action. This means that if the DOM changes in between the calls due to re-render, the new element corresponding to the locator will be used.const locator = page.getByRole('button', { name: 'Sign in' });await locator.hover();await locator.click();Note that all methods that create a locator, such as page.getByLabel(), are also available on the Locator and FrameLocator classes, so you can chain them and iteratively narrow down your

locator.const locator = page.frameLocator('#my-frame').getByRole('button', { name: 'Sign in' });await locator.click();

Locate by role The page.getByRole() locator reflects how users and assistive technology perceive the page, for example whether some element is a button or a checkbox. When locating by role, you should usually pass the accessible name as well, so that the locator pinpoints the exact element. For example, consider the following DOM structure.

```
http://localhost:3000Sign up
SubscribeSubmit<h3>Sign up</h3><label> <input type="checkbox" /> Subscribe</label><br/><button>Submit</button>
```

You can locate each element by its implicit role:

```
await expect(page.getByRole('heading', { name: 'Sign up' })).toBeVisible();
await page.getByRole('checkbox', { name: 'Subscribe' }).check();
await page.getByRole('button', { name: 'Submit' }).click();
```

Role locators include buttons, checkboxes, headings, links, lists, tables, and many more and follow W3C specifications for ARIA role, ARIA attributes and accessible name. Note that many HTML elements like <button> have an implicitly defined role that is recognized by the role locator. Note that role locators do not replace accessibility audits and conformance tests, but rather give early feedback about the ARIA guidelines.

When to use role locators We recommend prioritizing role locators to locate elements, as it is the closest way to how users and assistive technology perceive the page.

Locate by label Most form controls usually have dedicated labels that could be conveniently used to interact with the form. In this case, you can locate the control by its associated label using page.getByLabel(). For example, consider the following DOM structure.

```
http://localhost:3000Password <label>Password <input type="password" /></label>
```

You can fill the input after locating it by the label text:

```
await page.getByLabel('Password').fill('secret');
```

When to use label locators Use this locator when locating form fields.

Locate by placeholder Inputs may have a placeholder attribute to hint to the user what value should be entered. You can locate such an input using page.getByPlaceholder(). For example, consider the following DOM structure.

```
http://localhost:3000<input type="email" placeholder="name@example.com" />
```

You can fill the input after locating it by the placeholder text:

```
await page.getByPlaceholder('name@example.com').fill('playwright@microsoft.com');
```

When to use placeholder locators Use this locator when locating form elements that do not have labels but do have placeholder texts.

Locate by text Find an element by the text it contains. You can match by a substring, exact string, or a regular expression when using page.getByText(). For example, consider the following DOM structure.

```
http://localhost:3000Welcome, John<span>Welcome, John</span>
```

You can locate the element by the text it contains:

```
await expect(page.getByText('Welcome, John')).toBeVisible();
```

Set an exact match:

```
await expect(page.getByText('Welcome, John', { exact: true })).toBeVisible();
```

Match with a regular expression:

```
await expect(page.getByText(/welcome, [A-Za-z]+$/i)).toBeVisible();
```

note Matching by text always normalizes whitespace, even with exact match. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing whitespace.

When to use text locators We recommend using text locators to find non-interactive elements like div, span, p, etc. For interactive elements like button, a, input, etc. use role locators. You can also filter by text which can be useful when trying to find a particular item in a list.

Locate by alt text All images should have an alt attribute that describes the image. You can locate an image based on the text alternative using

`page.getByAltText()`. For example, consider the following DOM structure: `http://localhost:3000` You can click on the image after locating it by the text alternative: `await page.getByAltText('playwright logo').click()`; When to use alt locators Use this locator when your element supports alt text such as `img` and `area` elements.

Locate by title Locate an element with a matching title attribute using `page.getByTitle()`. For example, consider the following DOM structure: `http://localhost:300025 issues25 issues` You can check the issues count after locating it by the title text: `await expect(page.getByTitle('Issues count')).toHaveText('25 issues')`; When to use title locators Use this locator when your element has the title attribute.

Locate by test id Testing by test ids is the most resilient way of testing as even if your text or role of the attribute changes the test will still pass. QA's and developers should define explicit test ids and query them with `page.getByTestId()`. However testing by test ids is not user facing. If the role or text value is important to you then consider using user facing locators such as role and text locators. For example, consider the following DOM structure: `http://localhost:3000Itinéraire<button data-testid="directions">Itinéraire</button>` You can locate the element by its test id: `await page.getByTestId('directions').click()`; When to use testid locators You can also use test ids when you choose to use the test id methodology or when you can't locate by role or text. Set a custom test id attribute By default, `page.getByTestId()` will locate elements based on the `data-testid` attribute, but you can configure it in your test config or by calling `selectors.setTestIdAttribute()`. Set the test id to use a custom data attribute for your tests: `playwright.config.ts import { defineConfig } from '@playwright/test'; export default defineConfig({ use: { testIdAttribute: 'data-pw' } });` In your html you can now use `data-pw` as your test id instead of the default `data-testid`. `http://localhost:3000Itinéraire<button data-pw="directions">Itinéraire</button>` And then locate the element as you would normally do: `await page.getByTestId('directions').click()`;

Locate by CSS or XPath If you absolutely must use CSS or XPath locators, you can use `page.locator()` to create a locator that takes a selector describing how to find an element in the page. Playwright supports CSS and XPath selectors, and auto-detects them if you omit `css=` or `xpath=` prefix. `await page.locator('css=button').click()`; `await page.locator('xpath=//button').click()`; `await page.locator('button').click()`; `await page.locator('//button').click()`; XPath and CSS selectors can be tied to the DOM structure or implementation. These selectors can break when the DOM structure changes. Long CSS or XPath chains below are an example of a bad practice that leads to unstable tests: `await page.locator('#tsf > div:nth-child(2) > div.A8SBwf > div.RNNXgb > div > div.a4blc > input').click()`; `await page.locator('//*[@id="tsf"]/div[2]/div[1]/div[1]/div/div[2]/input').click()`; When to use this CSS and XPath are not recommended as the DOM can often change leading to non resilient tests. Instead, try to come up with a locator that is close to how the user perceives the page such as role locators or define an explicit testing contract using test ids.

Locate in Shadow DOM All locators in Playwright by default work with elements in Shadow DOM. The exceptions are: Locating by XPath does not pierce shadow roots. Closed-mode shadow roots are not supported. Consider the following example with a custom web component: `<x-details role=button aria-expanded=true aria-controls=inner-details> <div>Title</div> #shadow-root <div id=inner-details>Details</div>`

`div></x-details>` You can locate in the same way as if the shadow root was not present at all. To click `<div>Details</div>`: `await page.getByText('Details').click();` `<x-details role=button aria-expanded=true aria-controls=inner-details>` `<div>Title</div>` `#shadow-root` `<div id=inner-details>Details</div></x-details>` To click `<x-details>`: `await page.locator('x-details', { hasText: 'Details' }).click();` `<x-details role=button aria-expanded=true aria-controls=inner-details>` `<div>Title</div>` `#shadow-root` `<div id=inner-details>Details</div></x-details>` To ensure that `<x-details>` contains the text "Details": `await expect(page.locator('x-details')).toContainText('Details');`

Filtering Locators

Consider the following DOM structure where we want to click on the buy button of the second product card. We have a few options in order to filter the locators to get the right one.

```

http://localhost:3000
Product 1Add to cartProduct 2Add to cart
<ul> <li>
<h3>Product 1</h3> <button>Add to cart</button> </li> <li> <h3>Product 2</h3>
<button>Add to cart</button> </li></ul>

```

Filter by text

Locators can be filtered by text with the `locator.filter()` method. It will search for a particular string somewhere inside the element, possibly in a descendant element, case-insensitively. You can also pass a regular expression.

```

await page .getByRole('listitem') .filter({ hasText: 'Product 2' }) .getByRole('button', { name: 'Add to cart' }) .click();
// Use a regular expression
await page .getByRole('listitem') .filter({ hasText: /Product 2/ }) .getByRole('button', { name: 'Add to cart' }) .click();

```

Filter by not having text

Alternatively, filter by not having text:

```

// 5 in-stock items
await expect(page.getByRole('listitem').filter({ hasNotText: 'Out of stock' })).toHaveLength(5);

```

Filter by child/descendant

Locators support an option to only select elements that have or have not a descendant matching another locator. You can therefore filter by any other locator such as `locator.getByRole()`, `locator.getByTestId()`, `locator.getByText()` etc.

```

http://localhost:3000
Product 1Add to cartProduct 2Add to cart
<ul> <li> <h3>Product 1</h3> <button>Add to cart</button> </li> <li>
<h3>Product 2</h3> <button>Add to cart</button> </li></ul>
await page .getByRole('listitem') .filter({ has: page.getByRole('heading', { name: 'Product 2' }) }) .getByRole('button', { name: 'Add to cart' }) .click();

```

We can also assert the product card to make sure there is only one

```

await expect(page .getByRole('listitem') .filter({ has: page.getByText('Product 2') })).toHaveLength(1);

```

Filter by not having child/descendant

We can also filter by not having a matching element inside.

```

await expect(page .getByRole('listitem') .filter({ hasNot: page.getByText('Product 2') })).toHaveLength(1);

```

Note that the inner locator is matched starting from the outer one, not from the document root.

Locator operators

Matching inside a locator

You can chain methods that create a locator, like `page.getByText()` or `locator.getByRole()`, to narrow down the search to a particular part of the page. In this example we first create a locator called `product` by locating its role of `listitem`. We then filter by text. We can use the `product` locator again to get by role of `button` and click it and then use an assertion to make sure there is only one product with the text "Product 2".

```

const product = page.getByRole('listitem').filter({ hasText: 'Product 2' });
await product.getByRole('button', { name: 'Add to cart' }).click();
await expect(product).toHaveLength(1);

```

You can also chain two locators together, for example to find a "Save" button inside a particular dialog:

```

const saveButton = page.getByRole('button', { name: 'Save' });
// ...
const dialog = page.getByTestId('settings-

```

dialog');await dialog.locator(saveButton).click();Matching two locators simultaneously
Method locator.and() narrows down an existing locator by matching an additional
locator. For example, you can combine page.getByRole() and page.getByTitle() to
match by both role and title.const button =
page.getByRole('button').and(page.getByTitle('Subscribe'));Matching one of the two
alternative locators If you'd like to target one of the two or more elements, and you don't
know which one it will be, use locator.or() to create a locator that matches any of the
alternatives.For example, consider a scenario where you'd like to click on a "New email"
button, but sometimes a security settings dialog shows up instead. In this case, you can
wait for either a "New email" button, or a dialog and act accordingly.const newEmail =
page.getByRole('button', { name: 'New' });const dialog = page.getByText('Confirm
security settings');await expect(newEmail.or(dialog)).toBeVisible();if (await
dialog.isVisible()) await page.getByRole('button', { name: 'Dismiss' }).click();await
newEmail.click();Matching only visible elements notIt's usually better to find a more
reliable way to uniquely identify the element instead of checking the visibility.Consider a
page with two buttons, first invisible and second visible.<button style='display:
none'>Invisible</button><button>Visible</button>This will find both buttons and throw a
strictness violation error:await page.locator('button').click();This will only find a second
button, because it is visible, and then click it.await
page.locator('button').locator('visible=true').click();Lists Count items in a list You can
assert locators in order to count the items in a list.For example, consider the following
DOM structure:http://localhost:3000applebananaorange apple
banana orangeUse the count assertion to ensure that the list has
3 items.await expect(page.getByRole('listitem')).toHaveCount(3);Assert all text in a list
You can assert locators in order to find all the text in a list.For example, consider the
following DOM structure:http://localhost:3000applebananaorange apple
banana orangeUse expect(locator).toHaveText() to ensure that
the list has the text "apple", "banana" and "orange".await
expect(page .getByRole('listitem')) .toHaveText(['apple', 'banana', 'orange']);Get a
specific item There are many ways to get a specific item in a list.Get by text Use the
page.getByText() method to locate an element in a list by it's text content and then click
on it.For example, consider the following DOM structure:http://
localhost:3000applebananaorange apple banana orange</
li>Locate an item by it's text content and click it.await
page.getByText('orange').click();http://localhost:3000applebananaorange
apple banana orangeFilter by text Use the locator.filter()
to locate a specific item in a list.For example, consider the following DOM
structure:http://localhost:3000applebananaorange apple banana
orangeLocate an item by the role of "listitem" and then filter by the text of
"orange" and then click it.await page .getByRole('listitem') .filter({ hasText:
'orange' }) .click();Get by test id Use the page.getByTestId() method to locate an
element in a list. You may need to modify the html and add a test id if you don't already
have a test id.For example, consider the following DOM structure:http://
localhost:3000applebananaorange <li data-testid='apple'>apple <li data-
testid='banana'>banana <li data-testid='orange'>orangeLocate an item
by it's test id of "orange" and then click it.await page.getByTestId('orange').click();Get by

nth item If you have a list of identical elements, and the only way to distinguish between them is the order, you can choose a specific element from a list with `locator.first()`, `locator.last()` or `locator.nth()`.

```
const banana = await
page.getByRole('listitem').nth(1);
```

However, use this method with caution. Often times, the page might change, and the locator will point to a completely different element from the one you expected. Instead, try to come up with a unique locator that will pass the strictness criteria.

Chaining filters

When you have elements with various similarities, you can use the `locator.filter()` method to select the right one. You can also chain multiple filters to narrow down the selection. For example, consider the following DOM structure:

```
http://localhost:3000JohnSay helloMarySay helloJohnSay goodbyeMarySay
goodbye<ul> <li> <div>John</div> <div><button>Say hello</button></div> </li>
<li> <div>Mary</div> <div><button>Say hello</button></div> </li> <li>
<div>John</div> <div><button>Say goodbye</button></div> </li> <li> <div>Mary</
div> <div><button>Say goodbye</button></div> </li></ul>
```

To take a screenshot of the row with "Mary" and "Say goodbye":

```
const rowLocator = page.getByRole('listitem');await
rowLocator .filter({ hasText: 'Mary' }) .filter({ has: page.getByRole('button', { name:
'Say goodbye' }) }) .screenshot({ path: 'screenshot.png' });
```

You should now have a "screenshot.png" file in your project's root directory.

Rare use cases

Do something with each element in the list

Iterate elements:

```
for (const row of await
page.getByRole('listitem').all()) console.log(await row.textContent());
```

Iterate using regular for loop:

```
const rows = page.getByRole('listitem');const count = await
rows.count();for (let i = 0; i < count; ++i) console.log(await
rows.nth(i).textContent());
```

Evaluate in the page

The code inside `locator.evaluateAll()` runs in the page, you can call any DOM apis there.

```
const rows =
page.getByRole('listitem');const texts = await rows.evaluateAll( list =>
list.map(element => element.textContent()));
```

Strictness

Locators are strict. This means that all operations on locators that imply some target DOM element will throw an exception if more than one element matches. For example, the following call throws if there are several buttons in the DOM:

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

On the other hand, Playwright understands when you perform a multiple-element operation, so the following call works perfectly fine when the locator resolves to multiple elements.

```
await
page.getByRole('button').count();
```

You can explicitly opt-out from strictness check by telling Playwright which element to use when multiple elements match, through `locator.first()`, `locator.last()`, and `locator.nth()`. These methods are not recommended because when your page changes, Playwright may click on an element you did not intend. Instead, follow best practices above to create a locator that uniquely identifies the target element.

More Locators

For less commonly used locators, look at the other locators guide.

Mock APIs

Introduction

Web APIs are usually implemented as HTTP endpoints. Playwright provides APIs to mock and modify network traffic, both HTTP and HTTPS. Any requests that a page does, including XHRs and fetch requests, can be tracked, modified and mocked. With Playwright you can also mock using HAR files that contain multiple network requests made by the page.

Mock API requests

The following code will intercept all the calls to `*/**/api/v1/fruits` and will return a custom response instead. No requests to the API will be made. The test goes to the URL that uses the mocked route

and asserts that mock data is present on the page.`test("mocks a fruit and doesn't call api", async ({ page }) => { // Mock the api call before navigating await page.route('*/**/api/v1/fruits', async route => { const json = [{ name: 'Strawberry', id: 21 }]; await route.fulfill({ json }); }); // Go to the page await page.goto('https://demo.playwright.dev/api-mocking'); // Assert that the Strawberry fruit is visible await expect(page.getByText('Strawberry')).toBeVisible();});` You can see from the trace of the example test that the API was never called, it was however fulfilled with the mock data. Read more about advanced networking. Modify API responses Sometimes, it is essential to make an API request, but the response needs to be patched to allow for reproducible testing. In that case, instead of mocking the request, one can perform the request and fulfill it with the modified response. In the example below we intercept the call to the fruit API and add a new fruit called 'playwright', to the data. We then go to the url and assert that this data is there:`test('gets the json from api and adds a new fruit', async ({ page }) => { // Get the response and add to it await page.route('*/**/api/v1/fruits', async route => { const response = await route.fetch(); const json = await response.json(); json.push({ name: 'Playwright', id: 100 }); // Fulfill using the original response, while patching the response body // with the given JSON object. await route.fulfill({ response, json }); }); // Go to the page await page.goto('https://demo.playwright.dev/api-mocking'); // Assert that the new fruit is visible await expect(page.getByText('Playwright', { exact: true })).toBeVisible();});` In the trace of our test we can see that the API was called and the response was modified. By inspecting the response we can see that our new fruit was added to the list. Read more about advanced networking. Mocking with HAR files A HAR file is an HTTP Archive file that contains a record of all the network requests that are made when a page is loaded. It contains information about the request and response headers, cookies, content, timings, and more. You can use HAR files to mock network requests in your tests. You'll need to: Record a HAR file. Commit the HAR file alongside the tests. Route requests using the saved HAR files in the tests. Recording a HAR file To record a HAR file we use `page.routeFromHAR()` or `browserContext.routeFromHAR()` method. This method takes in the path to the HAR file and an optional object of options. The options object can contain the URL so that only requests with the URL matching the specified glob pattern will be served from the HAR File. If not specified, all requests will be served from the HAR file. Setting update option to true will create or update the HAR file with the actual network information instead of serving the requests from the HAR file. Use it when creating a test to populate the HAR with real data.`test('records or updates the HAR file', async ({ page }) => { // Get the response from the HAR file await page.routeFromHAR('./hars/fruit.har', { url: '*/**/api/v1/fruits', update: true, }); // Go to the page await page.goto('https://demo.playwright.dev/api-mocking'); // Assert that the fruit is visible await expect(page.getByText('Strawberry')).toBeVisible();});` Modifying a HAR file Once you have recorded a HAR file you can modify it by opening the hashed .txt file inside your 'hars' folder and editing the JSON. This file should be committed to your source control. Anytime you run this test with `update: true` it will update your HAR file with the request from the API.`[{ "name": "Playwright", "id": 100 }, // ... other fruits]` Replaying from HAR Now that you have the HAR file recorded and modified the mock data, it can be used to serve matching responses in the test. For this, just turn off or simply remove the update option. This will run the test against the

HAR file instead of hitting the API. `test('gets the json from HAR and checks the new fruit has been added', async ({ page }) => { // Replay API requests from HAR. // Either use a matching response from the HAR, // or abort the request if nothing matches. await page.routeFromHAR('./hars/fruit.har', { url: '*/api/v1/fruits', update: false, }); // Go to the page await page.goto('https://demo.playwright.dev/api-mocking'); // Assert that the Playwright fruit is visible await expect(page.getByText('Playwright', { exact: true })).toBeVisible();});` In the trace of our test we can see that the route was fulfilled from the HAR file and the API was not called. If we inspect the response we can see our new fruit was added to the JSON, which was done by manually updating the hashed .txt file inside the hars folder. HAR replay matches URL and HTTP method strictly. For POST requests, it also matches POST payloads strictly. If multiple recordings match a request, the one with the most matching headers is picked. An entry resulting in a redirect will be followed automatically. Similar to when recording, if given HAR file name ends with .zip, it is considered an archive containing the HAR file along with network payloads stored as separate entries. You can also extract this archive, edit payloads or HAR log manually and point to the extracted har file. All the payloads will be resolved relative to the extracted har file on the file system. Recording HAR with CLI We recommend the update option to record HAR file for your test. However, you can also record the HAR with Playwright CLI. Open the browser with Playwright CLI and pass --save-har option to produce a HAR file. Optionally, use --save-har-glob to only save requests you are interested in, for example API endpoints. If the har file name ends with .zip, artifacts are written as separate files and are all compressed into a single zip. # Save API requests from example.com as "example.har" `archive.npx playwright open --save-har=example.har --save-har-glob="*/api/*" https://example.com` Read more about advanced networking.

Mock browser APIs Playwright provides native support for most of the browser features. However, there are some experimental APIs and APIs which are not (yet) fully supported by all browsers. Playwright usually doesn't provide dedicated automation APIs in such cases. You can use mocks to test the behavior of your application in such cases. This guide gives a few examples. Introduction Let's consider a web app that uses battery API to show your device's battery status. We'll mock the battery API and check that the page correctly displays the battery status. Creating mocks Since the page may be calling the API very early while loading it's important to setup all the mocks before the page started loading. The easiest way to achieve that is to call `page.addInitScript()`: `await page.addInitScript(() => { const mockBattery = { level: 0.75, charging: true, chargingTime: 1800, dischargingTime: Infinity, addEventListener: () => {} }; // Override the method to always return mock battery info. window.navigator.getBattery = async () => mockBattery;});` Once this is done you can navigate the page and check its UI state: // Configure mock API before each test. `test.beforeEach(async ({ page }) => { await page.addInitScript(() => { const mockBattery = { level: 0.90, charging: true, chargingTime: 1800, // seconds dischargingTime: Infinity, addEventListener: () => {} }; // Override the method to always return mock battery info. window.navigator.getBattery = async () => mockBattery; });});` `test('show battery status', async ({ page }) => { await page.goto('/'); await expect(page.locator('.battery-percentage')).toHaveText('90%'); await expect(page.locator('.battery-status')).toHaveText('Adapter'); await`

expect(page.locator('.battery-fully')).toHaveText('00:30');});

Verifying API calls Sometimes it is useful to check if the page made all expected APIs calls. You can record all API method invocations and then compare them with golden result. `page.exposeFunction()` may come in handy for passing message from the page back to the test code:

```
test('log battery calls', async ({ page }) => {
  const log = [];
  // Expose function for pushing messages to the Node.js script.
  await page.exposeFunction('logCall', msg => log.push(msg));
  await page.addInitScript(() => {
    const mockBattery = {
      level: 0.75,
      charging: true,
      chargingTime: 1800,
      dischargingTime: Infinity,
    };
    // Log addEventListener calls.
    addEventListener: (name, cb) => logCall(`addEventListener: ${name}`);
    // Override the method to always return mock battery info.
    window.navigator.getBattery = async () => {
      logCall('getBattery');
      return mockBattery;
    };
  });
  await page.goto('/');
  await expect(page.locator('.battery-percentage')).toHaveText('75%');
  // Compare actual calls with golden.
  expect(log).toEqual([
    'getBattery',
    'addEventListener:chargingchange',
    'addEventListener:levelchange'
  ]);
});
```

Updating mock To test that the app correctly reflects battery status updates it's important to make sure that the mock battery object fires same events that the browser implementation would. The following test demonstrates how to achieve that:

```
test('update battery status (no golden)', async ({ page }) => {
  await page.addInitScript(() => {
    // Mock class that will notify corresponding listeners when battery status changes.
    class BatteryMock {
      level = 0.10;
      charging = false;
      chargingTime = 1800;
      dischargingTime = Infinity;
      _chargingListeners = [];
      _levelListeners = [];
      addEventListener(eventName, listener) {
        if (eventName === 'chargingchange')
          this._chargingListeners.push(listener);
        if (eventName === 'levelchange')
          this._levelListeners.push(listener);
      }
      // Will be called by the test.
      _setLevel(value) {
        this.level = value;
        this._levelListeners.forEach(cb => cb());
      }
      _setCharging(value) {
        this.charging = value;
        this._chargingListeners.forEach(cb => cb());
      }
    }
    const mockBattery = new BatteryMock();
    // Override the method to always return mock battery info.
    window.navigator.getBattery = async () => mockBattery;
    // Save the mock object on window for easier access.
    window.mockBattery = mockBattery;
  });
  await page.goto('/');
  await expect(page.locator('.battery-percentage')).toHaveText('10%');
  // Update level to 27.5%
  await page.evaluate(() => window.mockBattery._setLevel(0.275));
  await expect(page.locator('.battery-percentage')).toHaveText('27.5%');
  await expect(page.locator('.battery-status')).toHaveText('Battery');
  // Emulate connected adapter
  await page.evaluate(() => window.mockBattery._setCharging(true));
  await expect(page.locator('.battery-status')).toHaveText('Adapter');
  await expect(page.locator('.battery-fully')).toHaveText('00:30');
});
```

Navigations Playwright can navigate to URLs and handle navigations caused by the page interactions. Basic navigation Simplest form of a navigation is opening a URL://

Navigate the page

```
await page.goto('https://example.com');
```

The code above loads the page and waits for the web page to fire the load event. The load event is fired when the whole page has loaded, including all dependent resources such as stylesheets, scripts, iframes, and images. `not` If the page does a client-side redirect before load, `page.goto()` will wait for the redirected page to fire the load event. When is the page loaded? Modern

pages perform numerous activities after the load event was fired. They fetch data lazily, populate UI, load expensive resources, scripts and styles after the load event was fired. There is no way to tell that the page is loaded, it depends on the page, framework, etc. So when can you start interacting with it? In Playwright you can interact with the page at any moment. It will automatically wait for the target elements to become actionable.// Navigate and click element// Click will auto-wait for the elementawait page.goto('https://example.com');await page.getByText('Example Domain').click();For the scenario above, Playwright will wait for the text to become visible, will wait for the rest of the actionability checks to pass for that element, and will click it. Playwright operates as a very fast user - the moment it sees the button, it clicks it. In the general case, you don't need to worry about whether all the resources loaded, etc.

Hydration At some point in time, you'll stumble upon a use case where Playwright performs an action, but nothing seemingly happens. Or you enter some text into the input field and will disappear. The most probable reason behind that is a poor page hydration. When page is hydrated, first, a static version of the page is sent to the browser. Then the dynamic part is sent and the page becomes "live". As a very fast user, Playwright will start interacting with the page the moment it sees it. And if the button on a page is enabled, but the listeners have not yet been added, Playwright will do its job, but the click won't have any effect. A simple way to verify if your page suffers from a poor hydration is to open Chrome DevTools, pick "Slow 3G" network emulation in the Network panel and reload the page. Once you see the element of interest, interact with it. You'll see that the button clicks will be ignored and the entered text will be reset by the subsequent page load code. The right fix for this issue is to make sure that all the interactive controls are disabled until after the hydration, when the page is fully functional.

Waiting for navigation Clicking an element could trigger multiple navigations. In these cases, it is recommended to explicitly page.waitForURL() to a specific url.await page.getByText('Click me').click();await page.waitForURL('**/login');Navigation events Playwright splits the process of showing a new document in a page into navigation and loading. Navigation starts by changing the page URL or by interacting with the page (e.g., clicking a link). The navigation intent may be canceled, for example, on hitting an unresolved DNS address or transformed into a file download. Navigation is committed when the response headers have been parsed and session history is updated. Only after the navigation succeeds (is committed), the page starts loading the document. Loading covers getting the remaining response body over the network, parsing, executing the scripts and firing load events:page.url() is set to the new urldocument content is loaded over network and parsedpage.on('domcontentloaded') event is firedpage executes some scripts and loads resources like stylesheets and imagespage.on('load') event is firedpage executes dynamically loaded scripts

NetworkIntroduction Playwright provides APIs to monitor and modify browser network traffic, both HTTP and HTTPS. Any requests that a page does, including XHRs and fetch requests, can be tracked, modified and handled.

Mock APIs Check out our API mocking guide to learn more on how to mock API requests and never hit the APIperform the API request and modify the responseuse HAR files to mock network requests.

Network mocking You don't have to configure anything to mock network requests. Just define a custom Route that mocks network for a browser context.example.spec.tsimport { test, expect } from '@playwright/

```

test';test.beforeEach(async ({ context }) => { // Block any css requests for each test in
this file. await context.route(/.css$/, route => route.abort());});test('loads page without
css', async ({ page }) => { await page.goto('https://playwright.dev'); // ... test goes
here});Alternatively, you can use page.route() to mock network in a single
page.example.spec.tsimport { test, expect } from '@playwright/test';test('loads page
without images', async ({ page }) => { // Block png and jpeg images. await page.route(/
(png|jpeg)$/, route => route.abort()); await page.goto('https://playwright.dev'); // ... test
goes here});HTTP Authentication Perform HTTP
Authentication.TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/
test';export default defineConfig({ use: { httpCredentials: { username: 'bill',
password: 'pa55w0rd', } }});const context = await
browser.newContext({ httpCredentials: { username: 'bill', password:
'pa55w0rd', },});const page = await context.newPage();await page.goto('https://
example.com');HTTP Proxy You can configure pages to load over the HTTP(S) proxy or
SOCKSv5. Proxy can be either set globally for the entire browser, or for each browser
context individually.You can optionally specify username and password for HTTP(S)
proxy, you can also specify hosts to bypass proxy for.Here is an example of a global
proxy:TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/
test';export default defineConfig({ use: { proxy: { server: 'http://
myproxy.com:3128', username: 'usr', password: 'pwd' } }});const browser =
await chromium.launch({ proxy: { server: 'http://myproxy.com:3128', username:
'usr', password: 'pwd' }});When specifying proxy for each context individually,
Chromium on Windows needs a hint that proxy will be set. This is done via passing a
non-empty proxy server to the browser itself. Here is an example of a context-specific
proxy:TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/
test';export default defineConfig({ use: { launchOptions: { // Browser proxy option
is required for Chromium on Windows. proxy: { server: 'per-context' } }, proxy:
{ server: 'http://myproxy.com:3128', } }});const browser = await
chromium.launch({ // Browser proxy option is required for Chromium on Windows.
proxy: { server: 'per-context' }});const context = await browser.newContext({ proxy:
{ server: 'http://myproxy.com:3128' }});Network events You can monitor all the Requests
and Responses:// Subscribe to 'request' and 'response' events.page.on('request',
request => console.log('>>', request.method(), request.url()));page.on('response',
response => console.log('<<', response.status(), response.url()));await
page.goto('https://example.com');Or wait for a network response after the button click
with page.waitForResponse();// Use a glob URL pattern. Note no await.const
responsePromise = page.waitForResponse('**/api/fetch_data');await
page.getByText('Update').click();const response = await responsePromise;Variations
Wait for Responses with page.waitForResponse();// Use a RegExp. Note no await.const
responsePromise = page.waitForResponse(/\.jpeg$/);await
page.getByText('Update').click();const response = await responsePromise;// Use a
predicate taking a Response object. Note no await.const responsePromise =
page.waitForResponse(response => response.url().includes(token));await
page.getByText('Update').click();const response = await responsePromise;Handle
requests await page.route('**/api/fetch_data', route => route.fulfill({ status: 200, body:
testData,}));await page.goto('https://example.com');You can mock API endpoints via

```

handling the network requests in your Playwright script. **Variations** Set up route on the entire browser context with `browserContext.route()` or page with `page.route()`. It will apply to popup windows and opened links.

```
await browserContext.route('**/api/login', route => route.fulfill({ status: 200, body: 'accept'}));
await page.goto('https://example.com');
// Modify requests
// Delete header
await page.route('**/*', route => { const headers = route.request().headers(); delete headers['X-Secret']; route.continue({ headers }); });
// Continue requests as POST
await page.route('**/*', route => route.continue({ method: 'POST' }));
```

You can continue requests with modifications. Example above removes an HTTP header from the outgoing requests.

Abort requests You can abort requests using `page.route()` and `route.abort()`.

```
await page.route('**/*.{png,jpg,jpeg}', route => route.abort());
// Abort based on the request type
await page.route('**/*', route => { return route.request().resourceType() === 'image' ? route.abort() : route.continue(); });
```

Modify responses To modify a response use `APIRequestContext` to get the original response and then pass the response to `route.fulfill()`. You can override individual fields on the response via options:

```
await page.route('**/title.html', async route => { // Fetch original response. const response = await route.fetch(); // Add a prefix to the title. let body = await response.text(); body = body.replace('<title>', '<title>My prefix:'); route.fulfill({ // Pass all fields from the response. response, // Override response body. body, // Force content type to be html. headers: { ...response.headers(), 'content-type': 'text/html' } }); });
```

WebSockets Playwright supports WebSockets inspection out of the box. Every time a WebSocket is created, the `page.on('websocket')` event is fired. This event contains the WebSocket instance for further web socket frames inspection:

```
page.on('websocket', ws => { console.log(`WebSocket opened: ${ws.url()}`); ws.on('framesent', event => console.log(event.payload)); ws.on('framereceived', event => console.log(event.payload)); ws.on('close', () => console.log('WebSocket closed')); });
```

Missing Network Events and Service Workers Playwright's built-in `browserContext.route()` and `page.route()` allow your tests to natively route requests and perform mocking and interception. If you're using Playwright's native `browserContext.route()` and `page.route()`, and it appears network events are missing, disable Service Workers by setting `Browser.newContext.serviceWorkers` to `'block'`. It might be that you are using a mock tool such as Mock Service Worker (MSW). While this tool works out of the box for mocking responses, it adds its own Service Worker that takes over the network requests, hence making them invisible to `browserContext.route()` and `page.route()`. If you are interested in both network testing and mocking, consider using built-in `browserContext.route()` and `page.route()` for response mocking. If you're interested in not solely using Service Workers for testing and network mocking, but in routing and listening for requests made by Service Workers themselves, please see this experimental feature.

Other locator `stipCheck` out the main locators guide for most common and recommended locators. In addition to recommended locators like `page.getByRole()` and `page.getByText()`, Playwright supports a variety of other locators described in this guide.

CSS locator note We recommend prioritizing user-visible locators like text or accessible role instead of using CSS that is tied to the implementation and could break when the page changes. Playwright can locate an element by CSS selector.

```
await page.locator('css=button').click();
```

Playwright augments standard CSS selectors in two

ways: CSS selectors pierce open shadow DOM. Playwright adds custom pseudo-classes like `:visible`, `:has-text()`, `:has()`, `:is()`, `:nth-match()` and more.

CSS: matching by text

Playwright includes a number of CSS pseudo-classes to match elements by their text content.

`article:has-text("Playwright")` - the `:has-text()` matches any element containing specified text somewhere inside, possibly in a child or a descendant element. Matching is case-insensitive, trims whitespace and searches for a substring. For example, `article:has-text("Playwright")` matches `<article><div>Playwright</div></article>`. Note that `:has-text()` should be used together with other CSS specifiers, otherwise it will match all the elements containing specified text, including the `<body>`.

`// Wrong, will match many elements including <body>`
`await page.locator(':has-text("Playwright")').click();`
`// Correct, only matches the <article> element`
`await page.locator('article:has-text("Playwright")').click();`

`#nav-bar :text("Home")` - the `:text()` pseudo-class matches the smallest element containing specified text. Matching is case-insensitive, trims whitespace and searches for a substring. For example, this will find an element with text "Home" somewhere inside the `#nav-bar` element.

`await page.locator('#nav-bar :text("Home")').click();`

`#nav-bar :text-is("Home")` - the `:text-is()` pseudo-class matches the smallest element with exact text. Exact matching is case-sensitive, trims whitespace and searches for the full string. For example, `:text-is("Log")` does not match `<button>Log in</button>` because `<button>` contains a single text node "Log in" that is not equal to "Log". However, `:text-is("Log")` matches `<button> Log in</button>`, because `<button>` contains a text node " Log". Similarly, `:text-is("Download")` will not match `<button>download</button>` because it is case-sensitive.

`#nav-bar :text-matches("reg?ex", "i")` - the `:text-matches()` pseudo-class matches the smallest element with text content matching the JavaScript-like regex. For example, `:text-matches("Log\\s*in", "i")` matches `<button>Login</button>` and `<button>log IN</button>`.

Text matching always normalizes whitespace. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing whitespace.

Input elements of the type button and submit are matched by their value instead of text content. For example, `:text("Log in")` matches `<input type=button value="Log in">`.

CSS: matching only visible elements

Playwright supports the `:visible` pseudo class in CSS selectors. For example, `css=button` matches all the buttons on the page, while `css=button:visible` only matches visible buttons. This is useful to distinguish elements that are very similar but differ in visibility. Consider a page with two buttons, first invisible and second visible.

`<button style='display:none'>Invisible</button><button>Visible</button>`

This will find both buttons and throw a strictness violation error.

`await page.locator('button').click();` This will only find a second button, because it is visible, and then click it.

`await page.locator('button:visible').click();`

CSS: elements that contain other elements

The `:has()` pseudo-class is an experimental CSS pseudo-class. It returns an element if any of the selectors passed as parameters relative to the `:scope` of the given element match at least one element.

Following snippet returns text content of an `<article>` element that has a `<div class=promo>` inside.

`await page.locator('article:has(div.promo)').textContent();`

CSS: elements matching one of the conditions

Comma-separated list of CSS selectors will match all elements that can be selected by one of the selectors in that list.

`// Clicks a <button> that has either a "Log in" or "Sign in" text.`
`await page.locator('button:has-text("Log in"), button:has-text("Sign`

in")).click();

The `:is()` pseudo-class is an experimental CSS pseudo-class that may be useful for specifying a list of extra conditions on an element.

CSS: matching elements based on layout

note Matching based on layout may produce unexpected results. For example, a different element could be matched when layout changes by one pixel. Sometimes, it is hard to come up with a good selector to the target element when it lacks distinctive features. In this case, using Playwright layout CSS pseudo-classes could help. These can be combined with regular CSS to pinpoint one of the multiple choices. For example, `input:right-of(:text("Password"))` matches an input field that is to the right of text "Password" - useful when the page has multiple inputs that are hard to distinguish between each other. Note that layout pseudo-classes are useful in addition to something else, like `input`. If you use a layout pseudo-class alone, like `:right-of(:text("Password"))`, most likely you'll get not the input you are looking for, but some empty element in between the text and the target input. Layout pseudo-classes use bounding client rect to compute distance and relative position of the elements.

- `:right-of(div > button)` - Matches elements that are to the right of any element matching the inner selector, at any vertical position.
- `:left-of(div > button)` - Matches elements that are to the left of any element matching the inner selector, at any vertical position.
- `:above(div > button)` - Matches elements that are above any of the elements matching the inner selector, at any horizontal position.
- `:below(div > button)` - Matches elements that are below any of the elements matching the inner selector, at any horizontal position.
- `:near(div > button)` - Matches elements that are near (within 50 CSS pixels) any of the elements matching the inner selector. Note that resulting matches are sorted by their distance to the anchor element, so you can use `locator.first()` to pick the closest one. This is only useful if you have something like a list of similar elements, where the closest is obviously the right one. However, using `locator.first()` in other cases most likely won't work as expected - it will not target the element you are searching for, but some other element that happens to be the closest like a random empty `<div>`, or an element that is scrolled out and is not currently visible.

// Fill an input to the right of "Username".await page.locator('input:right-of(:text("Username"))').fill('value');// Click a button near the promo card.await page.locator('button:near(.promo-card)').click();// Click the radio input in the list closest to the "Label 3".await page.locator('[type=radio]:left-of(:text("Label 3"))').first().click();

All layout pseudo-classes support optional maximum pixel distance as the last argument. For example `button:near(:text("Username"), 120)` matches a button that is at most 120 CSS pixels away from the element with the text "Username".

CSS: pick n-th match from the query result

note It is usually possible to distinguish elements by some attribute or text content, which is more resilient to page changes. Sometimes page contains a number of similar elements, and it is hard to select a particular one. For example:

```
<section> <button>Buy</button> </section>
<article><div> <button>Buy</button> </div></article>
<div><div> <button>Buy</button> </div></div>
```

In this case, `:nth-match(:text("Buy"), 3)` will select the third button from the snippet above. Note that index is one-based.

// Click the third "Buy" buttonawait page.locator(':nth-match(:text("Buy"), 3)').click();

`:nth-match()` is also useful to wait until a specified number of elements appear, using `locator.waitFor()`.

// Wait until all three buttons are visibleawait page.locator(':nth-match(:text("Buy"), 3)').waitFor();

note Unlike `:nth-child()`, elements do not have to be siblings, they could be anywhere on the page. In the snippet above, all three buttons match `:text("Buy")`

selector, and `:nth-match()` selects the third button. N-th element locator You can narrow down query to the n-th match using the `nth=` locator passing a zero-based index.// Click first button `await page.locator('button').locator('nth=0').click();`// Click last button `await page.locator('button').locator('nth=-1').click();`Parent element locator When you need to target a parent element of some other element, most of the time you should `locator.filter()` by the child locator. For example, consider the following DOM structure: `<label>Hello</label><label>World</label>` If you'd like to target the parent `` of a label with text "Hello", using `locator.filter()` works best: `const child = page.getByText('Hello');``const parent = page.getByRole('listitem').filter({ has: child });`Alternatively, if you cannot find a suitable locator for the parent element, use `xpath=...` Note that this method is not as reliable, because any changes to the DOM structure will break your tests. Prefer `locator.filter()` when possible.`const parent = page.getByText('Hello').locator('xpath=..');`React locator `note`React locator is experimental and prefixed with `_`. The functionality might change in future. React locator allows finding elements by their component name and property values. The syntax is very similar to CSS attribute selectors and supports all CSS attribute selector operators. In React locator, component names are transcribed with CamelCase. `await page.locator('_react=BookItem').click();`More examples: match by component: `_react=BookItem` match by component and exact property value, case-sensitive: `_react=BookItem[author = "Steven King"]` match by property value only, case-insensitive: `_react=[author = "steven king" i]` match by component and truthy property value: `_react=MyButton[enabled]` match by component and boolean value: `_react=MyButton[enabled = false]` match by property value substring: `_react=[author *= "King"]` match by component and multiple properties: `_react=BookItem[author *= "king" i][year = 1990]` match by nested property value: `_react=[some.nested.value = 12]` match by component and property value prefix: `_react=BookItem[author ^= "Steven"]` match by component and property value suffix: `_react=BookItem[author $= "Steven"]` match by component and key: `_react=BookItem[key = '2']` match by property value regex: `_react=[author = /Steven(\\s+King)?/i]` To find React element names in a tree use React DevTools. `note`React locator supports React 15 and above. `note`React locator, as well as React DevTools, only work against unminified application builds. Vue locator `note`Vue locator is experimental and prefixed with `_`. The functionality might change in future. Vue locator allows finding elements by their component name and property values. The syntax is very similar to CSS attribute selectors and supports all CSS attribute selector operators. In Vue locator, component names are transcribed with kebab-case. `await page.locator('_vue=book-item').click();`More examples: match by component: `_vue=book-item` match by component and exact property value, case-sensitive: `_vue=book-item[author = "Steven King"]` match by property value only, case-insensitive: `_vue=[author = "steven king" i]` match by component and truthy property value: `_vue=my-button[enabled]` match by component and boolean value: `_vue=my-button[enabled = false]` match by property value substring: `_vue=[author *= "King"]` match by component and multiple properties: `_vue=book-item[author *= "king" i][year = 1990]` match by nested property value: `_vue=[some.nested.value = 12]` match by component and property value prefix: `_vue=book-item[author ^= "Steven"]` match by component and property value suffix: `_vue=book-item[author $= "Steven"]` match by property value regex: `_vue=[author = /Steven(\\s+King)?/i]` To find Vue element names in

a tree use Vue DevTools.noteVue locator supports Vue2 and above.noteVue locator, as well as Vue DevTools, only work against unminified application builds.XPath locator dangerWe recommend prioritizing user-visible locators like text or accessible role instead of using XPath that is tied to the implementation and easily break when the page changes.XPath locators are equivalent to calling Document.evaluate.await page.locator('xpath=//button').click();noteAny selector string starting with // or .. are assumed to be an xpath selector. For example, Playwright converts '//html/body' to 'xpath=//html/body'.noteXPath does not pierce shadow roots.XPath union Pipe operator (|) can be used to specify multiple selectors in XPath. It will match all elements that can be selected by one of the selectors in that list.// Waits for either confirmation dialog or load spinner.await page.locator('//span[contains(@class, 'spinner__loading')]/div[@id='confirmation']').waitFor();Label to form control retargeting dangerWe recommend locating by label text instead of relying to label-to-control retargeting.Targeted input actions in Playwright automatically distinguish between labels and controls, so you can target the label to perform an action on the associated control.For example, consider the following DOM structure: <label for="password">Password:</label><input id="password" type="password">. You can target the label by it's "Password" text using page.getByText(). However, the following actions will be performed on the input instead of the label:locator.click() will click the label and automatically focus the input field;locator.fill() will fill the input field;locator.inputValue() will return the value of the input field;locator.selectText() will select text in the input field;locator.setInputFiles() will set files for the input field with type=file;locator.selectOption() will select an option from the select box.// Fill the input by targeting the label.await page.getByText('Password').fill('secret');However, other methods will target the label itself, for example expect(locator).toHaveText() will assert the text content of the label, not the input field.// Fill the input by targeting the label.await expect(page.locator('label')).toHaveText('Password');Legacy text locator dangerWe recommend the modern text locator instead.Legacy text locator matches elements that contain passed text.await page.locator('text=Log in').click();Legacy text locator has a few variations:text=Log in - default matching is case-insensitive, trims whitespace and searches for a substring. For example, text=Log matches <button>Log in</button>.await page.locator('text=Log in').click();text="Log in" - text body can be escaped with single or double quotes to search for a text node with exact content after trimming whitespace.For example, text="Log" does not match <button>Log in</button> because <button> contains a single text node "Log in" that is not equal to "Log". However, text="Log" matches <button> Log in</button>, because <button> contains a text node " Log ". This exact mode implies case-sensitive matching, so text="Download" will not match <button>download</button>.Quoted body follows the usual escaping rules, e.g. use \" to escape double quote in a double-quoted string: text="foo\"bar".await page.locator('text="Log in"').click();/Log\\s*in/i - body can be a JavaScript-like regex wrapped in / symbols. For example, text=/Log\\s*in/i matches <button>Login</button> and <button>log IN</button>.await page.locator('text=/Log\\s*in/i').click();noteString selectors starting and ending with a quote (either " or ') are assumed to be a legacy text locators. For example, "Log in" is converted to text="Log in" internally.noteMatching always normalizes whitespace. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing

whitespace. `note` Input elements of the type `button` and `submit` are matched by their value instead of text content. For example, `text=Log in` matches `<input type=button value="Log in">`. `id`, `data-testid`, `data-test-id`, `data-test` selectors `danger` We recommend locating by test id instead. Playwright supports shorthand for selecting elements using certain attributes. Currently, only the following attributes are supported: `id`, `data-testid`, `data-test-id`, `data-test`. // Fill an input with the id "username" `await page.locator('id=username').fill('value');` // Click an element with `data-test-id` "submit" `await page.locator('data-test-id=submit').click();` `note` Attribute selectors are not CSS selectors, so anything CSS-specific like `:enabled` is not supported. For more features, use a proper CSS selector, e.g. `css=[data-test="login"]:enabled`. `note` Attribute selectors pierce shadow DOM. To opt-out from this behavior, use `:light` suffix after attribute, for example `page.locator('data-test-id:light=submit').click()` Chaining selectors `danger` We recommend chaining locators instead. Selectors defined as `engine=body` or in short-form can be combined with the `>>` token, e.g. `selector1 >> selector2 >> selectors3`. When selectors are chained, the next one is queried relative to the previous one's result. For example, `css=article >> css=.bar > .baz >> css=span[attr=value]` is equivalent to `document.querySelector('article').querySelector('.bar > .baz').querySelector('span[attr=value]');` If a selector needs to include `>>` in the body, it should be escaped inside a string to not be confused with chaining separator, e.g. `text="some >> text"`. Intermediate matches `danger` We recommend filtering by another locator to locate elements that contain other elements. By default, chained selectors resolve to an element queried by the last selector. A selector can be prefixed with `*` to capture elements that are queried by an intermediate selector. For example, `css=article >> text=Hello` captures the element with the text `Hello`, and `*css=article >> text=Hello` (note the `*`) captures the `article` element that contains some element with the text `Hello`.

Page object models Large test suites can be structured to optimize ease of authoring and maintenance. Page object models are one such approach to structure your test suite. A page object represents a part of your web application. An e-commerce web application might have a home page, a listings page and a checkout page. Each of them can be represented by page object models. Page objects simplify authoring by creating a higher-level API which suits your application and simplify maintenance by capturing element selectors in one place and create reusable code to avoid repetition. Implementation We will create a `PlaywrightDevPage` helper class to encapsulate common operations on the `playwright.dev` page. Internally, it will use the page object. `TypeScript` `JavaScript` `Library` `playwright-dev-page.ts` `import { expect, type Locator, type Page } from '@playwright/test';` `export class PlaywrightDevPage` `{` `readonly page: Page;` `readonly getStartedLink: Locator;` `readonly` `gettingStartedHeader: Locator;` `readonly pomLink: Locator;` `readonly tocList: Locator;` `constructor(page: Page) {` `this.page = page;` `this.getStartedLink = page.locator('a', {` `hasText: 'Get started' });` `this.gettingStartedHeader = page.locator('h1', {` `hasText: 'Installation' });` `this.pomLink = page.locator('li', {` `hasText: 'Guides' }).locator('a', {` `hasText: 'Page Object Model' });` `this.tocList = page.locator('article div.markdown ul > li > a');` `}` `async goto() {` `await this.page.goto('https://playwright.dev');` `}` `async` `getStarted() {` `await this.getStartedLink.first().click();` `await` `expect(this.gettingStartedHeader).toBeVisible();` `}` `async pageObjectModel() {` `await`

```

this.getStarted(); await this.pomLink.click(); })playwright-dev-page.jsconst { expect } =
require('@playwright/test');exports.PlaywrightDevPage = class PlaywrightDevPage { /
** * @param {import('@playwright/test').Page} page */ constructor(page)
{ this.page = page; this.getStartedLink = page.locator('a', { hasText: 'Get
started' }); this.gettingStartedHeader = page.locator('h1', { hasText: 'Installation' });
this.pomLink = page.locator('li', { hasText: 'Guides' }).locator('a', { hasText: 'Page Object
Model' }); this.tocList = page.locator('article div.markdown ul > li > a'); } async goto()
{ await this.page.goto('https://playwright.dev'); } async getStarted() { await
this.getStartedLink.first().click(); await
expect(this.gettingStartedHeader).toBeVisible(); } async pageObjectModel() { await
this.getStarted(); await this.pomLink.click(); });models/PlaywrightDevPage.jsclass
PlaywrightDevPage { /** * @param {import('playwright').Page} page */
constructor(page) { this.page = page; this.getStartedLink = page.locator('a',
{ hasText: 'Get started' }); this.gettingStartedHeader = page.locator('h1', { hasText:
'Installation' }); this.pomLink = page.locator('li', { hasText: 'Playwright
Test' }).locator('a', { hasText: 'Page Object Model' }); this.tocList = page.locator('article
div.markdown ul > li > a'); } async getStarted() { await
this.getStartedLink.first().click(); await
expect(this.gettingStartedHeader).toBeVisible(); } async pageObjectModel() { await
this.getStarted(); await this.pomLink.click(); })module.exports =
{ PlaywrightDevPage };Now we can use the PlaywrightDevPage class in our
tests.TypeScriptJavaScriptLibraryexample.spec.tsimport { test, expect } from
'@playwright/test';import { PlaywrightDevPage } from './playwright-dev-
page';test('getting started should contain table of contents', async ({ page }) => { const
playwrightDev = new PlaywrightDevPage(page); await playwrightDev.goto(); await
playwrightDev.getStarted(); await expect(playwrightDev.tocList).toHaveText([ `How to
install Playwright`, `What's Installed`, `How to run the example test`, `How to open
the HTML test report`, `Write tests using web first assertions, page fixtures and
locators`, `Run single test, multiple tests, headed mode`, `Generate tests with
Codegen`, `See a trace of your tests` ]));test('should show Page Object Model
article', async ({ page }) => { const playwrightDev = new PlaywrightDevPage(page);
await playwrightDev.goto(); await playwrightDev.pageObjectModel(); await
expect(page.locator('article')).toContainText('Page Object Model is a common
pattern');});example.spec.jsconst { test, expect } = require('@playwright/test');const
{ PlaywrightDevPage } = require('./playwright-dev-page');test('getting started should
contain table of contents', async ({ page }) => { const playwrightDev = new
PlaywrightDevPage(page); await playwrightDev.goto(); await
playwrightDev.getStarted(); await expect(playwrightDev.tocList).toHaveText([ `How to
install Playwright`, `What's Installed`, `How to run the example test`, `How to open
the HTML test report`, `Write tests using web first assertions, page fixtures and
locators`, `Run single test, multiple tests, headed mode`, `Generate tests with
Codegen`, `See a trace of your tests` ]));test('should show Page Object Model
article', async ({ page }) => { const playwrightDev = new PlaywrightDevPage(page);
await playwrightDev.goto(); await playwrightDev.pageObjectModel(); await
expect(page.locator('article')).toContainText('Page Object Model is a common
pattern');});example.spec.jsconst { PlaywrightDevPage } = require('./playwright-dev-

```

```

page');// In the testconst page = await browser.newPage();await
playwrightDev.goto();await playwrightDev.getStarted();await
expect(playwrightDev.tocList).toHaveText([ `How to install Playwright`, `What's
Installed`, `How to run the example test`, `How to open the HTML test report`, `Write
tests using web first assertions, page fixtures and locators`, `Run single test, multiple
tests, headed mode`, `Generate tests with Codegen`, `See a trace of your tests`]);
PagesPagesMultiple pagesHandling new pagesHandling popupsPages Each
BrowserContext can have multiple pages. A Page refers to a single tab or a popup
window within a browser context. It should be used to navigate to URLs and interact
with the page content.// Create a page.const page = await context.newPage();//
Navigate explicitly, similar to entering a URL in the browser.await page.goto('http://
example.com');// Fill an input.await page.locator('#search').fill('query');// Navigate
implicitly by clicking a link.await page.locator('#submit').click();// Expect a new
url.console.log(page.url());Multiple pages Each browser context can host multiple pages
(tabs).Each page behaves like a focused, active page. Bringing the page to front is not
required.Pages inside a context respect context-level emulation, like viewport sizes,
custom network routes or browser locale.// Create two pagesconst pageOne = await
context.newPage();const pageTwo = await context.newPage();// Get pages of a browser
contextconst allPages = context.pages();Handling new pages The page event on
browser contexts can be used to get new pages that are created in the context. This
can be used to handle new pages opened by target="_blank" links.// Start waiting for
new page before clicking. Note no await.const pagePromise =
context.waitForEvent('page');await page.getByText('open new tab').click();const
newPage = await pagePromise;await newPage.waitForLoadState();console.log(await
newPage.title());If the action that triggers the new page is unknown, the following
pattern can be used.// Get all new pages (including popups) in the
contextcontext.on('page', async page => { await page.waitForLoadState();
console.log(await page.title());});Handling popups If the page opens a pop-up (e.g.
pages opened by target="_blank" links), you can get a reference to it by listening to the
popup event on the page.This event is emitted in addition to the
browserContext.on('page') event, but only for popups relevant to this page.// Start
waiting for popup before clicking. Note no await.const popupPromise =
page.waitForEvent('popup');await page.getByText('open the popup').click();const popup
= await popupPromise;// Wait for the popup to load.await
popup.waitForLoadState();console.log(await popup.title());If the action that triggers the
popup is unknown, the following pattern can be used.// Get all popups when they
openpage.on('popup', async popup => { await popup.waitForLoadState();
console.log(await popup.title());});
ScreenshotsHere is a quick way to capture a screenshot and save it into a file:await
page.screenshot({ path: 'screenshot.png' });Screenshots API accepts many parameters
for image format, clip area, quality, etc. Make sure to check them out.Full page
screenshotsCapture into bufferElement screenshotFull page screenshots Full page
screenshot is a screenshot of a full scrollable page, as if you had a very tall screen and
the page could fit it entirely.await page.screenshot({ path: 'screenshot.png', fullPage:
true });Capture into buffer Rather than writing into a file, you can get a buffer with the
image and post-process it or pass it to a third party pixel diff facility.const buffer = await

```

page.screenshot();console.log(buffer.toString('base64'));Element screenshot Sometimes it is useful to take a screenshot of a single element.await

```
page.locator('.header').screenshot({ path: 'screenshot.png' });
```

Visual comparisonsPlaywright Test includes the ability to produce and visually compare screenshots using await expect(page).toHaveScreenshot(). On first execution,

Playwright test will generate reference screenshots. Subsequent runs will compare

against the reference.example.spec.tsimport { test, expect } from '@playwright/

```
test';test('example test', async ({ page }) => { await page.goto('https://playwright.dev');
```

```
await expect(page).toHaveScreenshot();});When you run above for the first time, test
```

runner will say:Error: A snapshot doesn't exist at example.spec.ts-snapshots/example-test-1-chromium-darwin.png, writing actual.That's because there was no golden file yet.

This method took a bunch of screenshots until two consecutive screenshots matched, and saved the last screenshot to file system. It is now ready to be added to the

repository.The name of the folder with the golden expectations starts with the name of

your test file:drwxr-xr-x 5 user group 160 Jun 4 11:46 .drwxr-xr-x 6 user group 192

Jun 4 11:45 ..-rw-r--r-- 1 user group 231 Jun 4 11:16 example.spec.tsdrwxr-xr-x 3

user group 96 Jun 4 11:46 example.spec.ts-snapshotsThe snapshot name example-

test-1-chromium-darwin.png consists of a few parts:example-test-1.png - an auto-

generated name of the snapshot. Alternatively you can specify snapshot name as the

first argument of the toHaveScreenshot() method:await

```
expect(page).toHaveScreenshot('landing.png');chromium-darwin - the browser name
```

and the platform. Screenshots differ between browsers and platforms due to different

rendering, fonts and more, so you will need different snapshots for them. If you use

multiple projects in your configuration file, project name will be used instead of

chromium.If you are not on the same operating system as your CI system, you can use

Docker to generate/update the screenshots:docker run --rm --network host -v \$(pwd):/

work/ -w /work/ -it mcr.microsoft.com/playwright:v1.36.0-jammy /bin/bashnpm installnpx

playwright test --update-snapshotsSometimes you need to update the reference

screenshot, for example when the page has changed. Do this with the --update-

snapshots flag.npx playwright test --update-snapshotsNote that snapshotName also

accepts an array of path segments to the snapshot file such as

```
expect().toHaveScreenshot(['relative', 'path', 'to', 'snapshot.png']).
```

However, this path must stay within the snapshots directory for each test file (i.e.

a.spec.js-snapshots), otherwise it will throw.Playwright Test uses the pixelmatch library.

You can pass various options to modify its behavior:example.spec.tsimport { test,

```
expect } from '@playwright/test';test('example test', async ({ page }) => { await
```

```
page.goto('https://playwright.dev'); await
```

```
expect(page).toHaveScreenshot({ maxDiffPixels: 100 });});If you'd like to share the
```

default value among all the tests in the project, you can specify it in the playwright

config, either globally or per project:playwright.config.tsimport { defineConfig } from

```
'@playwright/test';export default defineConfig({ expect: { toHaveScreenshot:
```

```
{ maxDiffPixels: 100 }, },});Apart from screenshots, you can use
```

```
expect(value).toMatchSnapshot(snapshotName) to compare text or arbitrary binary
```

data. Playwright Test auto-detects the content type and uses the appropriate

comparison algorithm.Here we compare text content against the

```
reference.example.spec.tsimport { test, expect } from '@playwright/test';test('example
```

```
test', async ({ page }) => { await page.goto('https://playwright.dev'); expect(await page.textContent('.hero__title')).toMatchSnapshot('hero.txt');});
```

Snapshots are stored next to the test file, in a separate directory. For example, my.spec.ts file will produce and store snapshots in the my.spec.ts-snapshots directory. You should commit this directory to your version control (e.g. git), and review any changes to it.

Test generator Playwright comes with the ability to generate tests for you as you perform actions in the browser and is a great way to quickly get started with testing. Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If the generator finds multiple elements matching the locator, it will improve the locator to make it resilient that uniquely identify the target element.

Generate tests in VS Code

Install the VS Code extension and generate tests directly from VS Code. The extension is available on the VS Code Marketplace. Check out our guide on getting started with VS Code.

Record a New Test

To record a test click on the Record new button from the Testing sidebar. This will create a test-1.spec.ts file as well as open up a browser window. In the browser go to the URL you wish to test and start clicking around to record your user actions. Playwright will record your actions and generate the test code directly in VS Code. Once you are done recording click the cancel button or close the browser window. You can then inspect your test-1.spec.ts file and see your generated test and then manually improve the test by adding assertions.

Record at Cursor

To record from a specific point in your test move your cursor to where you want to record more actions and then click the Record at cursor button from the Testing sidebar. If your browser window is not already open then first run the test with 'Show browser' checked and then click the Record at cursor button. In the browser window start performing the actions you want to record. In the test file in VS Code you will see your new generated actions added to your test at the cursor position.

Generating locators

You can generate locators with the test generator. Click on the Pick locator button from the testing sidebar and then hover over elements in the browser window to see the locator highlighted underneath each element. Click the element you require and it will now show up in the Pick locator box in VS Code. Press Enter on your keyboard to copy the locator into the clipboard and then paste anywhere in your code. Or press 'escape' if you want to cancel.

Generate tests with the Playwright Inspector

When running the codegen command two windows will be opened, a browser window where you interact with the website you wish to test and the Playwright Inspector window where you can record your tests and then copy them into your editor.

Running Codegen

Use the codegen command to run the test generator followed by the URL of the website you want to generate tests for. The URL is optional and you can always run the command without it and then add the URL directly into the browser window instead.

```
npx playwright codegen demo.playwright.dev/todomvc
```

Recording a test

Run the codegen command and perform actions in the browser window. Playwright will generate the code for the user interactions which you can see in the Playwright Inspector window. Once you have finished recording your test stop the recording and press the copy button to copy your generated test into your editor.

Generating locators

You can generate locators with the test generator. Press the 'Record' button to stop the recording and the 'Pick Locator' button will appear. Click on the 'Pick Locator' button and then hover over elements in the browser window to see the locator highlighted underneath each element. To choose a locator click on the element you would like to locate and the code

for that locator will appear in the field next to the Pick Locator button. You can then edit the locator in this field to fine tune it or use the copy button to copy it and paste it into your code.

Emulation You can use the test generator to generate tests using emulation so as to generate a test for a specific viewport, device, color scheme, as well as emulate the geolocation, language or timezone. The test generator can also generate a test while preserving authenticated state.

Emulate viewport size Playwright opens a browser window with it's viewport set to a specific width and height and is not responsive as tests need to be run under the same conditions. Use the `--viewport` option to generate tests with a different viewport size.

```
npx playwright codegen --viewport-size=800,600 playwright.dev
```

Emulate devices Record scripts and tests while emulating a mobile device using the `--device` option which sets the viewport size and user agent among others.

```
npx playwright codegen --device="iPhone 13" playwright.dev
```

Emulate color scheme Record scripts and tests while emulating the color scheme with the `--color-scheme` option.

```
npx playwright codegen --color-scheme=dark playwright.dev
```

Emulate geolocation, language and timezone Record scripts and tests while emulating timezone, language & location using the `--timezone`, `--geolocation` and `--lang` options.

Once the page opens: Accept the cookies On the top right click on the locate me button to see geolocation in action.

```
npx playwright codegen --timezone="Europe/Rome" --geolocation="41.890221,12.492348" --lang="it-IT" bing.com/maps
```

Preserve authenticated state Run codegen with `--save-storage` to save cookies and localStorage at the end of the session. This is useful to separately record an authentication step and reuse it later when recording more tests.

```
npx playwright codegen github.com/microsoft/playwright --save-storage=auth.json
```

Login After performing authentication and closing the browser, `auth.json` will contain the storage state which you can then reuse in your tests. Make sure you only use the `auth.json` locally as it contains sensitive information. Add it to your `.gitignore` or delete it once you have finished generating your tests.

Load authenticated state Run with `--load-storage` to consume the previously loaded storage from the `auth.json`. This way, all cookies and localStorage will be restored, bringing most web apps to the authenticated state without the need to login again. This means you can continue generating tests from the logged in state.

```
npx playwright codegen --load-storage=auth.json github.com/microsoft/playwright
```

Record using custom setup If you would like to use codegen in some non-standard setup (for example, use `browserContext.route()`), it is possible to call `page.pause()` that will open a separate window with codegen controls.

```
const { chromium } = require('@playwright/test');
(async () => { // Make sure to run headed.
  const browser = await chromium.launch({ headless: false });
  // Setup context however you like.
  const context = await browser.newContext({ // * pass any options */ });
  await context.route('**/*', route => route.continue());
  // Pause the page, and start recording manually.
  const page = await context.newPage();
  await page.pause();
})();
```

Trace viewer Playwright Trace Viewer is a GUI tool that helps you explore recorded Playwright traces after the script has ran. You can open traces locally or in your browser on trace.playwright.dev.

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

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

Actions Once trace is opened, you will see the list of actions Playwright performed on the left hand side: Selecting each action reveals: action snapshots, action log, source code location, network log for this action. In the properties pane you will also

see rendered DOM snapshots associated with each action. Metadata See metadata such as the time the action was performed, what browser engine was used, what the viewport was and if it was mobile and how many pages, actions and events were recorded. Screenshots When tracing with the screenshots option turned on, each trace records a screencast and renders it as a film strip: You can hover over the film strip to see a magnified image of for each action and state which helps you easily find the action you want to inspect. Snapshots When tracing with the snapshots option turned on, Playwright captures a set of complete DOM snapshots for each action. Depending on the type of the action, it will capture: TypeDescriptionBeforeA snapshot at the time action is called. ActionA snapshot at the moment of the performed input. This type of snapshot is especially useful when exploring where exactly Playwright clicked. AfterA snapshot after the action. Here is what the typical Action snapshot looks like: Notice how it highlights both, the DOM Node as well as the exact click position. Call See what action was called, the time and duration as well as parameters, return value and log. Console See the console output for the action where you can see console logs or errors. Network See any network requests that were made during the action. Source See the source code for your entire test. Recording a trace locally To record a trace during development mode set the --trace flag to on when running your tests. `npx playwright test --trace on` You can then open the HTML report and click on the trace icon to open the trace. `npx playwright show-report` Recording a trace on CI Traces should be run on continuous integration on the first retry of a failed test by setting the trace: 'on-first-retry' option in the test configuration file. This will produce a trace.zip file for each test that was retried.

```
TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ retries: 1, use: { trace: 'on-first-retry', }, });const browser = await chromium.launch();const context = await browser.newContext();// Start tracing before creating / navigating a page.await context.tracing.start({ screenshots: true, snapshots: true });const page = await context.newPage();await page.goto('https://playwright.dev');// Stop tracing and export it into a zip archive.await context.tracing.stop({ path: 'trace.zip' });
```

Available options to record a trace: 'on-first-retry' - Record a trace only when retrying a test for the first time. 'on-all-retries' - Record traces for all test retries. 'off' - Do not record a trace. 'on' - Record a trace for each test. (not recommended as it's performance heavy) 'retain-on-failure' - Record a trace for each test, but remove it from successful test runs. You can also use trace: 'retain-on-failure' if you do not enable retries but still want traces for failed tests. If you are not using Playwright as a Test Runner, use the browserContext.tracing API instead. Viewing the trace You can open the saved trace using Playwright CLI or in your browser on `trace.playwright.dev`. `npx playwright show-trace trace.zip` Using `trace.playwright.dev` `trace.playwright.dev` is a statically hosted variant of the Trace Viewer. You can upload trace files using drag and drop. Viewing remote traces You can open remote traces using it's URL. They could be generated on a CI run which makes it easy to view the remote trace without having to manually download the file. `npx playwright show-trace https://example.com/trace.zip` You can also pass the URL of your uploaded trace (e.g. inside your CI) from some accessible storage as a parameter. CORS (Cross-Origin Resource Sharing) rules might apply. `https://trace.playwright.dev/?trace=https://demo.playwright.dev/reports/todomvc/data/cb0fa77ebd9487a5c899f3ae65a7ffdbac681182.zip`

VideosWith Playwright you can record videos for your tests. Record video Playwright Test can record videos for your tests, controlled by the video option in your Playwright config. By default videos are off. 'off' - Do not record video. 'on' - Record video for each test. 'retain-on-failure' - Record video for each test, but remove all videos from successful test runs. 'on-first-retry' - Record video only when retrying a test for the first time. Video files will appear in the test output directory, typically test-results. See testOptions.video for advanced video configuration. Videos are saved upon browser context closure at the end of a test. If you create a browser context manually, make sure to await browserContext.close().

```
TestLibraryplaywright.config.tsimport
{ defineConfig } from '@playwright/test';export default defineConfig({ use: { video: 'on-first-retry', },});const context = await browser.newContext({ recordVideo: { dir: 'videos/' } });// Make sure to await close, so that videos are saved.await
context.close();You can also specify video size. The video size defaults to the viewport size scaled down to fit 800x800. The video of the viewport is placed in the top-left corner of the output video, scaled down to fit if necessary. You may need to set the viewport size to match your desired video size.
```

```
TestLibraryplaywright.config.tsimport
{ defineConfig } from '@playwright/test';export default defineConfig({ use: { video: { mode: 'on-first-retry', size: { width: 640, height: 480 } } },});const context = await browser.newContext({ recordVideo: { dir: 'videos/', size: { width: 640, height: 480 }, } });For multi-page scenarios, you can access the video file associated with the page via the page.video().const path = await page.video().path();noteNote that the video is only available after the page or browser context is closed.
```

WebView2The following will explain how to use Playwright with Microsoft Edge WebView2. WebView2 is a WinForms control, which will use Microsoft Edge under the hood to render web content. It is a part of the Microsoft Edge browser and is available on Windows 10 and Windows 11. Playwright can be used to automate WebView2 applications and can be used to test web content in WebView2. For connecting to WebView2, Playwright uses browserType.connectOverCDP() which connects to it via the Chrome DevTools Protocol (CDP). Overview A WebView2 control can be instructed to listen to incoming CDP connections by setting either the WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS environment variable with --remote-debugging-port=9222 or calling EnsureCoreWebView2Async with the --remote-debugging-port=9222 argument. This will start the WebView2 process with the Chrome DevTools Protocol enabled which allows the automation by Playwright. 9222 is an example port in this case, but any other unused port can be used as well.

```
await
this.webView.EnsureCoreWebView2Async(await
CoreWebView2Environment.CreateAsync(null, null, new
CoreWebView2EnvironmentOptions(){ AdditionalBrowserArguments = "--remote-debugging-port=9222",})).ConfigureAwait(false);Once your application with the
WebView2 control is running, you can connect to it via Playwright:const browser = await
playwright.chromium.connectOverCDP('http://localhost:9222');const context =
browser.contexts()[0];const page = context.pages()[0];To ensure that the WebView2
control is ready, you can wait for the CoreWebView2InitializationCompleted
event:this.webView.CoreWebView2InitializationCompleted += (_, e) =>{ if
(e.IsSuccess) { Console.WriteLine("WebView2 initialized"); } };Writing and
running tests By default, the WebView2 control will use the same user data directory for
```

all instances. This means that if you run multiple tests in parallel, they will interfere with each other. To avoid this, you should set the `WEBVIEW2_USER_DATA_FOLDER` environment variable (or use `WebView2.EnsureCoreWebView2Async` Method) to a different folder for each test. This will make sure that each test runs in its own user data directory. Using the following, Playwright will run your WebView2 application as a subprocess, assign a unique user data directory to it and provide the Page instance to your test:

```
webView2Test.tsimport { test as base } from '@playwright/test';import fs from 'fs';import os from 'os';import path from 'path';import childProcess from 'child_process';const EXECUTABLE_PATH = path.join(__dirname, '../webview2-app/bin/Debug/net6.0-windows/webview2.exe');export const test = base.extend({ browser: async ({ playwright }, use, testInfo) => { const cdpPort = 10000 + testInfo.workerIndex; fs.accessSync(EXECUTABLE_PATH, fs.constants.X_OK); // Make sure that the executable exists and is executable const userDataDir = path.join(fs.realpathSync.native(os.tmpdir()), `playwright-webview2-tests/user-data-dir-${testInfo.workerIndex}`); const webView2Process = childProcess.spawn(EXECUTABLE_PATH, [], { shell: true, env: { ...process.env, WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS: `--remote-debugging-port=${cdpPort}`, WEBVIEW2_USER_DATA_FOLDER: userDataDir, } }); await new Promise<void>(resolve => webView2Process.stdout.on('data', data => { if (data.toString().includes('WebView2 initialized')) resolve(); })); const browser = await playwright.chromium.connectOverCDP(`http://127.0.0.1:${cdpPort}`); await use(browser); await browser.close(); childProcess.execSync(`taskkill /pid ${webView2Process.pid} /T /F`); fs.rmdirSync(userDataDir, { recursive: true }); }, context: async ({ browser }, use) => { const context = browser.contexts()[0]; await use(context); }, page: async ({ context }, use) => { const page = context.pages()[0]; await use(page); },});export { expect } from '@playwright/test';example.spec.tsimport { test, expect } from './webView2Test';test('test WebView2', async ({ page }) => { await page.goto('https://playwright.dev'); const getStarted = page.getByText('Get Started'); await expect(getStarted).toBeVisible();});
```

Debugging Inside your webview2 control, you can just right-click to open the context menu and select "Inspect" to open the DevTools or press F12. You can also use the `WebView2.CoreWebView2.OpenDevToolsWindow` method to open the DevTools programmatically. For debugging tests, see the Playwright Debugging guide.

Migrating from Protractor
 Migration Principles
 Cheat Sheet
 Example
 Polyfilling
 wait
 For Angular
 Playwright Test
 Super Powers
 Further Reading
 Migration Principles
 No need for "webdriver-manager" / Selenium.
 Protractor's ElementFinder !
 Playwright Test Locator
 Protractor's wait
 For Angular !
 Playwright Test auto-waiting
 Don't forget to await in Playwright Test
 Cheat Sheet
 Protractor
 Playwright
 Test
 element(by.buttonText('...'))
 page.locator('button, input[type="button"], input[type="submit"]')
 >>
 text="..."
 element(by.css('...'))
 page.locator('...')
 element(by.cssContainingText('..1..', '..2..'))
 page.locator('..1.. >>
 text=..2..')
 element(by.id('...'))
 page.locator('#...')
 element(by.model('...'))
 page.locator('[ng-model="..."]')
 element(by.repeater('...'))
 page.locator('[ng-repeat="..."]')
 element(by.xpath('..'))
 page.locator('xpath=...')
 element.all
 page.locator
 browser.get(url)
 await

```

page.goto(url)browser.getCurrentUrl()page.url()Example Protractor:describe('angularjs
homepage todo list', function() { it('should add a todo', function() { browser.get('https://
angularjs.org'); element(by.model('todoList.todoText')).sendKeys('first test');
element(by.css('[value="add"]')).click(); const todoList = element.all(by.repeater('todo
in todoList.todos')); expect(todoList.count()).toEqual(3);
expect(todoList.get(2).getText()).toEqual('first test'); // You wrote your first test, cross it
off the list todoList.get(2).element(by.css('input')).click(); const completedAmount =
element.all(by.css('.done-true'));
expect(completedAmount.count()).toEqual(2); }));});Line-by-line migration to Playwright
Test:const { test, expect } = require('@playwright/test'); // 1test.describe('angularjs
homepage todo list', () => { test('should add a todo', async ({ page }) => { // 2, 3 await
page.goto('https://angularjs.org'); // 4 await page.locator('[ng-
model="todoList.todoText"]').fill('first test'); await page.locator('[value="add"]').click();
const todoList = page.locator('[ng-repeat="todo in todoList.todos"]'); // 5 await
expect(todoList).toHaveCount(3); await expect(todoList.nth(2)).toHaveText('first test',
{ useInnerText: true, }); // You wrote your first test, cross it off the list await
todoList.nth(2).getByRole('textbox').click(); const completedAmount =
page.locator('.done-true'); await
expect(completedAmount).toHaveCount(2); }));});Migration highlights (see inline
comments in the Playwright Test code snippet):Each Playwright Test file has explicit
import of the test and expect functionsTest function is marked with asyncPlaywright
Test is given a page as one of its parameters. This is one of the many useful fixtures in
Playwright Test.Almost all Playwright calls are prefixed with awaitLocator creation with
page.locator() is one of the few methods that is sync.Polyfilling waitForAngular
Playwright Test has built-in auto-waiting that makes protractor's waitForAngular
unnneeded in general case.However, it might come handy in some edge cases. Here's
how to polyfill waitForAngular function in Playwright Test:Make sure you have protractor
installed in your package.jsonPolyfill functionasync function waitForAngular(page)
{ const clientSideScripts = require('protractor/built/clientsidescripts.js'); async function
executeScriptAsync(page, script, ...scriptArgs) { await page.evaluate(` new
Promise((resolve, reject) => { const callback = (errorMessage) => { if
(errorMessage) reject(new Error(errorMessage)); else
resolve(); }); (function() {${script}}).apply(null, [...${JSON.stringify(scriptArgs)},
callback]); }) `); } await executeScriptAsync(page,
clientSideScripts.waitForAngular, "");If you don't want to keep a version protractor
around, you can also use this simpler approach using this function (only works for
Angular 2+):async function waitForAngular(page) { await page.evaluate(async () =>
{ // @ts-expect-error if (window.getAllAngularTestabilities) { // @ts-expect-
error await Promise.all(window.getAllAngularTestabilities().map(whenStable)); //
@ts-expect-error async function whenStable(testability) { return new
Promise(res => testability.whenStable(res)); } } });}Polyfill usageconst page =
await context.newPage();await page.goto('https://example.org');await
waitForAngular(page);Playwright Test Super Powers Once you're on Playwright Test,
you get a lot!Full zero-configuration TypeScript supportRun tests across all web
engines (Chrome, Firefox, Safari) on any popular operating system (Windows, macOS,
Ubuntu)Full support for multiple origins, (i)frames, tabs and contextsRun tests in

```

parallel across multiple browsers

Built-in test artifact collection: video recording, screenshots and playwright traces

You also get all these 'awesome tools' that come bundled with Playwright Test:

- Playwright Inspector
- Playwright Test Code generation
- Playwright Tracing for post-mortem debugging

Further Reading

Learn more about Playwright Test runner:

- Getting Started
- Fixtures
- Locators
- Assertions
- Auto-waiting
- Migrating from Puppeteer
- Migration Principles
- Cheat Sheet
- Examples
- Testing Playwright
- Test Super Powers
- Further Reading
- Migration Principles

This guide describes migration to Playwright Library and Playwright Test from Puppeteer. The APIs have similarities, but Playwright offers much more possibilities for web testing and cross-browser automation.

Most Puppeteer APIs can be used as is

The use of `ElementHandle` is discouraged, use `Locator` objects and web-first assertions instead.

Playwright is cross-browser

You probably don't need explicit `wait`

Cheat Sheet

Puppeteer

Playwright

Library

```

await puppeteer.launch()
await playwright.chromium.launch()
puppeteer.launch({product: 'firefox'})
await playwright.firefox.launch()
WebKit is not supported by Puppeteer
await playwright.webkit.launch()
await browser.createIncognitoBrowserContext(...)
await browser.newContext(...)
await page.setViewport(...)
await page.setViewportSize(...)
await page.waitForXPath(XPathSelector)
await page.waitForSelector(XPathSelector)
await page.waitForNetworkIdle(...)
await page.waitForLoadState('networkidle')
await page.$eval(...)
Assertions can often be used instead to verify text, attribute, class...
await page.$(...)
Discouraged, use Locators instead
await page.$x(xpath_selector)
Discouraged, use Locators instead
No methods dedicated to checkbox or radio input
await page.locator(selector).check()
await page.locator(selector).uncheck()
await page.click(selector)
await page.locator(selector).click()
await page.focus(selector)
await page.locator(selector).focus()
await page.hover(selector)
await page.locator(selector).hover()
await page.select(selector, values)
await page.locator(selector).selectOption(values)
await page.tap(selector)
await page.locator(selector).tap()
await page.type(selector, ...)
await page.locator(selector).type(...)
Please also consider locator.fill()
await page.waitForFileChooser(...)
await elementHandle.uploadFile(...)
await page.locator(selector).setInputFiles(...)
await page.cookies([...urls])
await browserContext.cookies([urls])
await page.deleteCookie(...cookies)
await browserContext.clearCookies()
await page.setCookie(...cookies)
await browserContext.addCookies(cookies)
page.on(...)
page.on(...)
In order to intercept and mutate requests, see page.route()
page.waitForNavigation and page.waitForSelector remain, but in many cases will not be necessary due to auto-waiting.
The use of ElementHandle is discouraged, use Locator objects and web-first assertions instead.
Locators are the central piece of Playwright's auto-waiting and retry-ability.
Locators are strict. This means that all operations on locators that imply some target DOM element will throw an exception if more than one element matches a given selector.
Examples
Automation example
Puppeteer:
const puppeteer = require('puppeteer');
(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.setViewport({ width: 1280, height: 800 });
  await page.goto('https://playwright.dev/', {
    waitUntil: 'networkidle2',
  });
  await page.screenshot({ path: 'example.png' });
  await browser.close();
})();
Line-by-line migration to Playwright:
const { chromium } = require('playwright');
// 1
(async () =>

```

```

{ const browser = await chromium.launch(); const page = await browser.newPage(); //
2 await page.setViewportSize({ width: 1280, height: 800 }); // 3 await page.goto('https://
playwright.dev/', { waitUntil: 'networkidle', // 4 }); await page.screenshot({ path:
'example.png' }); await browser.close();});Migration highlights (see inline comments in
the Playwright code snippet):Each Playwright Library file has explicit import of
chromium. Other browsers webkit or firefox can be used.For browser state isolation,
consider browser contextssetViewport becomes setViewportSizenetworkidle2 becomes
networkidle. Please note that in most cases it is not useful, thanks to auto-waiting.Test
example Puppeteer with Jest:import puppeteer from 'puppeteer';describe('Playwright
homepage', () => { let browser; let page; beforeAll(async () => { browser = await
puppeteer.launch(); page = await browser.newPage(); }); it('contains hero title', async
() => { await page.goto('https://playwright.dev/'); await
page.waitForSelector('.hero__title'); const text = await page.$eval('.hero__title', e =>
e.textContent); expect(text).toContain('Playwright enables reliable end-to-end
testing'); // 5 }); afterAll(() => browser.close());});Line-by-line migration to Playwright
Test:import { test, expect } from '@playwright/test'; // 1test.describe('Playwright
homepage', () => { test('contains hero title', async ({ page }) => { // 2, 3 await
page.goto('https://playwright.dev/'); const titleLocator = page.locator('.hero__title'); //
4 await expect(titleLocator).toContainText( // 5 'Playwright enables reliable end-to-
end testing' ); });});Each Playwright Test file has explicit import of the test and expect
functionsTest function is marked with asyncPlaywright Test is given a page as one of its
parameters. This is one of the many useful fixtures in Playwright Test. Playwright Test
creates an isolated Page object for each test. However, if you'd like to reuse a single
Page object between multiple tests, you can create your own in test.beforeAll() and
close it in test.afterAll().Locator creation with page.locator() is one of the few methods
that is sync.Use assertions to verify the state instead of page.$eval().Testing To improve
testing, it is advised to use Locators and web-first Assertions. See Writing TestsIt is
common with Puppeteer to use page.evaluate() or page.$eval() to inspect an
ElementHandle and extract the value of text content, attribute, class... Web-first
Assertions offers several matchers for this purpose, it is more reliable and
readable.Playwright Test is our first-party recommended test runner to be used with
Playwright. It provides several features like Page Object Model, parallelism, fixtures or
reporters.Playwright Test Super Powers Once you're on Playwright Test, you get a lot!
Full zero-configuration TypeScript supportRun tests across all web engines (Chrome,
Firefox, Safari) on any popular operating system (Windows, macOS, Ubuntu)Full
support for multiple origins, (i)frames, tabs and contextsRun tests in isolation in parallel
across multiple browsersBuilt-in test artifact collection: video recording, screenshots
and playwright tracesYou also get all these '( awesome tools '( that come bundled with
Playwright Test:Playwright InspectorPlaywright Test Code generationPlaywright Tracing
for post-mortem debuggingFurther Reading Learn more about Playwright Test
runner:Getting StartedFixturesLocatorsAssertionsAuto-waiting
Migrating from Testing LibraryMigration principlesCheat SheetExampleMigrating
queriesReplacing waitForReplacing withinPlaywright Test Super PowersFurther
ReadingMigration principles This guide describes migration to Playwright's Experimental
Component Testing from DOM Testing Library, React Testing Library, Vue Testing
Library and Svelte Testing Library.notelf you use DOM Testing Library in the browser

```

(for example, you bundle end-to-end tests with webpack), you can switch directly to Playwright Test. Examples below are focused on component tests, but for end-to-end test you just need to replace `await mount` with `await page.goto('http://localhost:3000/')` to open the page under test.

Cheat Sheet

Testing Library: `screen`, `user`, `events`, `actions`, `await`

component queries: `locators`, `async`, `helpers`, `assertions`, `user`, `events`, `actions`, `await`

`user.click(screen.getByText('Click me'))`
`await component.getByText('Click me').click()`
`await user.click(await screen.findByText('Click me'))`
`await component.getByText('Click me').click()`
`await user.type(screen.getByLabel('Password'), 'secret')`
`await component.getByLabel('Password').fill('secret')`
`expect(screen.getByLabel('Password')).toHaveValue('secret')`
`await expect(component.getByLabel('Password')).toHaveValue('secret')`
`screen.findByText('...')`
`component.getByText('...')`
`screen.getByTestId('...')`
`component.getByTestId('...')`
`screen.queryByPlaceholderText('...')`
`component.getByPlaceholderText('...')`
`screen.getByRole('button', { pressed: true })`
`component.getByRole('button', { pressed: true })`

Example

Testing Library: `import React from 'react'; import { render, screen } from '@testing-library/react'; import userEvent from '@testing-library/user-event';`

`test('should sign in', async () => {`
 // Setup the page.
`const user = userEvent.setup();`
`render(<SignInPage />);`
 // Perform actions.
`await user.type(screen.getByLabel('Username'), 'John');`
`await user.type(screen.getByLabel('Password'), 'secret');`
`await user.click(screen.getByText('Sign in'));`
 // Verify signed in state by waiting until "Welcome" message appears.
`await screen.findByText('Welcome, John');`
`});`

Line-by-line migration to Playwright Test:

```
const { test, expect } = require('@playwright/experimental-ct-react');
// 1
test('should sign in', async ({ page, mount }) => {
  // 2 // Setup the page.
  const component = await mount(<SignInPage />);
  // 3 // Perform actions.
  await component.getByText('Username').fill('John');
  // 4
  await component.getByText('Password').fill('secret');
  await component.getByText('Sign in').click();
  // Verify signed in state by waiting until "Welcome" message appears.
  await expect(component.getByText('Welcome, John')).toBeVisible();
  // 5
});
```

Migration highlights (see inline comments in the Playwright Test code snippet):

- Import everything from `@playwright/experimental-ct-react` (or `-vue`, `-svelte`) for component tests, or from `@playwright/test` for end-to-end tests.
- Test function is given a page that is isolated from other tests, and `mount` that renders a component in this page. These are two of the useful fixtures in Playwright Test.
- Replace `render` with `mount` that returns a component locator.
- Use locators created with `locator.locator()` or `page.locator()` to perform most of the actions.
- Use assertions to verify the state.

Migrating queries

All queries like `getBy...`, `findBy...`, `queryBy...` and their multi-element counterparts are replaced with `component.getBy...` locators. Locators always auto-wait and retry when needed, so you don't have to worry about choosing the right method. When you want to do a list operation, e.g. assert a list of texts, Playwright automatically performs multi-element operations.

Replacing `waitFor`

Playwright includes assertions that automatically wait for the condition, so you don't usually need an explicit `waitFor`.

`waitForElementToBeRemoved` call.

Testing Library: `await waitFor(() => { expect(getByText('the lion king')).toBeInTheDocument(); });`
`await waitForElementToBeRemoved(() => queryByText('the mummy'));`

Playwright: `await expect(page.getByText('the lion king')).toBeVisible();`
`await expect(page.getByText('the mummy')).toBeHidden();`

When you cannot find a suitable assertion, use `expect.poll`

```

instead.await expect.poll(async () => { const response = await page.request.get('https://
api.example.com'); return response.status();}).toBe(200);
Replacing within You can create a locator inside another locator with locator.locator() method.
// Testing Library
const messages = document.getElementById('messages');
const helloMessage = within(messages).getByText('hello');
// Playwright
const messages = component.locator('id=messages');
const helloMessage = messages.getByText('hello');

```

Playwright Test Super Powers Once you're on Playwright Test, you get a lot!

- Full zero-configuration TypeScript support
- Run tests across all web engines (Chrome, Firefox, Safari) on any popular operating system (Windows, macOS, Ubuntu)
- Full support for multiple origins, (i)frames, tabs and contexts
- Run tests in isolation in parallel across multiple browsers
- Built-in test artifact collection: video recording, screenshots and playwright traces
- You also get all these 'awesome tools' that come bundled with Playwright Test:
 - Playwright Inspector
 - Playwright Test Code generation
 - Playwright Tracing for post-mortem debugging

Further Reading Learn more about Playwright Test runner:

- Getting Started
- Experimental Component Testing
- Locators
- Assertions
- Auto-waiting

Docker Dockerfile.jammy can be used to run Playwright scripts in Docker environment. This image includes all the dependencies needed to run browsers in a Docker container, and also include the browsers themselves.

Usage **Image tags** **Development Usage** This Docker image is published to Microsoft Artifact Registry.info This Docker image is intended to be used for testing and development purposes only. It is not recommended to use this Docker image to visit untrusted websites. Pull the image `docker pull mcr.microsoft.com/playwright:v1.36.0-jammy` Run the image By default, the Docker image will use the root user to run the browsers. This will disable the Chromium sandbox which is not available with root. If you run trusted code (e.g. End-to-end tests) and want to avoid the hassle of managing separate user then the root user may be fine. For web scraping or crawling, we recommend to create a separate user inside the Docker container and use the seccomp profile.

End-to-end tests On trusted websites, you can avoid creating a separate user and use root for it since you trust the code which will run on the browsers. `docker run -it --rm --ipc=host mcr.microsoft.com/playwright:v1.36.0-jammy /bin/bash`

Crawling and scraping On untrusted websites, it's recommended to use a separate user for launching the browsers in combination with the seccomp profile. Inside the container or if you are using the Docker image as a base image you have to use `adduser` for it. `docker run -it --rm --ipc=host --user pwuser --security-opt seccomp=seccomp_profile.json mcr.microsoft.com/playwright:v1.36.0-jammy /bin/bash`

`seccomp_profile.json` is needed to run Chromium with sandbox. This is a default Docker seccomp profile with extra user namespace cloning permissions:


```
{
  "comment": "Allow create user namespaces",
  "names": [ "clone", "setns", "unshare" ],
  "action": "SCMP_ACT_ALLOW",
  "args": [],
  "includes": {},
  "excludes": {}
}
```

note Using `--ipc=host` is recommended when using Chrome (Docker docs). Chrome can run out of memory without this flag.

Using on CI See our Continuous Integration guides for sample configs.

Image tags See all available image tags. Docker images are published automatically by GitHub Actions. We currently publish images with the following tags (v1.33.0 in this case is an example):

- `next` - tip-of-tree image version based on Ubuntu 22.04 LTS (Jammy Jellyfish).
- `next-jammy` - tip-of-tree image version based on Ubuntu 22.04 LTS (Jammy Jellyfish).
- `v1.33.0` - Playwright

v1.33.0 release docker image based on Ubuntu 22.04 LTS (Jammy Jellyfish):v1.33.0-jammy - Playwright v1.33.0 release docker image based on Ubuntu 22.04 LTS (Jammy Jellyfish):v1.33.0-focal - Playwright v1.33.0 release docker image based on Ubuntu 20.04 LTS (Focal Fossa):sha-XXXXXXX - docker image for every commit that changed docker files or browsers, marked with a short sha (first 7 digits of the SHA commit).notelt is recommended to always pin your Docker image to a specific version if possible. If the Playwright version in your Docker image does not match the version in your project/tests, Playwright will be unable to locate browser executables.Base images We currently publish images based on the following Ubuntu versions:Ubuntu 22.04 LTS (Jammy Jellyfish), image tags include jammyUbuntu 20.04 LTS (Focal Fossa), image tags include focalAlpine Browser builds for Firefox and WebKit are built for the glibc library. Alpine Linux and other distributions that are based on the musl standard library are not supported.Development Build the image Use //utils/docker/build.sh to build the image../utils/docker/build.sh jammy playwright:localbuild-jammyThe image will be tagged as playwright:localbuild-jammy and could be run as:docker run --rm -it playwright:localbuild /bin/bash

Continuous IntegrationPlaywright tests can be executed in CI environments. We have created sample configurations for common CI providers.Introduction 3 steps to get your tests running on CI:Ensure CI agent can run browsers: Use our Docker image in Linux agents or install your dependencies using the CLI.Install Playwright:# Install NPM packagesnpm ci# Install Playwright browsers and dependenciesnpx playwright install --with-depsRun your tests:npx playwright testWorkers We recommend setting workers to "1" in CI environments to prioritize stability and reproducibility. Running tests sequentially ensures each test gets the full system resources, avoiding potential conflicts. However, if you have a powerful self-hosted CI system, you may enable parallel tests. For wider parallelization, consider sharding - distributing tests across multiple CI jobs.playwright.config.tsimport { defineConfig, devices } from '@playwright/test';export default defineConfig({ // Opt out of parallel tests on CI. workers: process.env.CI ? 1 : undefined,});CI configurations The Command line tools can be used to install all operating system dependencies on GitHub Actions.GitHub Actions On push/pull_request name: Playwright Testson: push: branches: [main, master] pull_request: branches: [main, master]jobs: test: timeout-minutes: 60 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 Browsers run: npx playwright install --with-deps - name: Run Playwright tests run: npx playwright test - uses: actions/upload-artifact@v3 if: always() with: name: playwright-report path: playwright-report/ retention-days: 30On push/pull_request (sharded) GitHub Actions supports sharding tests between multiple jobs using the jobs.<job_id>.strategy.matrix option. The matrix option will run a separate job for every possible combination of the provided options. In the example below, we have 2 project values, 10 shardIndex values and 1 shardTotal value, resulting in a total of 20 jobs to be run. So it will split up the tests between 20 jobs, each running a different browser and a different subset of tests, see here for more details.name: Playwright Testson: push: branches: [main, master] pull_request: branches: [main, master]jobs: playwright: name: 'Playwright Tests - \${{ matrix.project }} - Shard \${{ matrix.shardIndex }} of \${{ matrix.shardTotal }}' runs-

on: ubuntu-latest strategy: fail-fast: false matrix: project: [chromium, webkit] shardIndex: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] shardTotal: [10] steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: 18 - name: Install dependencies run: npm ci - name: Install browsers run: npx playwright install --with-deps - name: Run your tests run: npx playwright test --project=\${{ matrix.project }} --shard=\${{ matrix.shardIndex }}/\${{ matrix.shardTotal }}

Note: The `${{ <expression> }}` is the expression syntax that allows accessing the current context. In this example, we are using the matrix context to set the job variants.

Via Containers GitHub Actions support running jobs in a container by using the `jobs.<job_id>.container` option. This is useful to not pollute the host environment with dependencies and to have a consistent environment for e.g. screenshots/visual regression testing across different operating systems.

name: Playwright Test on: push: branches: [main, master] pull_request: branches: [main, master] jobs: playwright: name: 'Playwright Tests' runs-on: ubuntu-latest container: image: mcr.microsoft.com/playwright:v1.36.0-jammy steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: 18 - name: Install dependencies run: npm ci - name: Run your tests run: npx playwright test

Via Containers (sharded) GitHub Actions supports sharding tests between multiple jobs using the `jobs.<job_id>.strategy.matrix` option. The matrix option will run a separate job for every possible combination of the provided options. In the example below, we have 2 project values, 10 shardIndex values and 1 shardTotal value, resulting in a total of 20 jobs to be run. So it will split up the tests between 20 jobs, each running a different browser and a different subset of tests, see here for more details.

name: Playwright Test on: push: branches: [main, master] pull_request: branches: [main, master] jobs: playwright: name: 'Playwright Tests - `${{ matrix.project }}` - Shard `${{ matrix.shardIndex }}` of `${{ matrix.shardTotal }}`' runs-on: ubuntu-latest container: image: mcr.microsoft.com/playwright:v1.36.0-jammy strategy: fail-fast: false matrix: project: [chromium, webkit] shardIndex: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] shardTotal: [10] steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: 18 - name: Install dependencies run: npm ci - name: Run your tests run: npx playwright test --project=\${{ matrix.project }} --shard=\${{ matrix.shardIndex }}/\${{ matrix.shardTotal }}

On deployment This will start the tests after a GitHub Deployment went into the success state. Services like Vercel use this pattern so you can run your end-to-end tests on their deployed environment.

name: Playwright Test on: deployment_status: jobs: test: timeout-minutes: 60 runs-on: ubuntu-latest if: github.event.deployment_status.state == 'success' 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 Playwright tests run: npx playwright test

env: PLAYWRIGHT_TEST_BASE_URL: `${{ github.event.deployment_status.target_url }}`

Docker We have a pre-built Docker image which can either be used directly, or as a reference to update your existing Docker definitions. Suggested configuration Using `--ipc=host` is also recommended when using Chromium. Without it Chromium can run out of memory and crash. Learn more about this option in Docker docs. Seeing other weird errors when launching Chromium?

Try running your container with `docker run --cap-add=SYS_ADMIN` when developing locally. Using `--init` Docker flag or `dumb-init` is recommended to avoid special treatment for processes with `PID=1`. This is a common reason for zombie processes.

Azure Pipelines

For Windows or macOS agents, no additional configuration required, just install Playwright and run your tests. For Linux agents, you can use our Docker container with Azure Pipelines support running containerized jobs. Alternatively, you can use Command line tools to install all necessary dependencies.

For running the Playwright tests use this pipeline task:

```
trigger:- mainpool: vmlImage: ubuntu-latest
steps:- task: NodeTool@0 inputs: versionSpec: '18' displayName: 'Install Node.js'- script: npm ci displayName: 'npm ci'- script: npx playwright install --with-deps displayName: 'Install Playwright browsers'- script: npx playwright test displayName: 'Run Playwright tests'
```

Uploading playwright-report folder with Azure Pipelines This will make the pipeline run fail if any of the playwright tests fails. If you also want to integrate the test results with Azure DevOps, use the task `PublishTestResults` task like so:

```
trigger:- mainpool: vmlImage: ubuntu-latest
steps:- task: NodeTool@0 inputs: versionSpec: '18' displayName: 'Install Node.js'- script: npm ci displayName: 'npm ci'- script: npx playwright install --with-deps displayName: 'Install Playwright browsers'- script: npx playwright test displayName: 'Run Playwright tests'- task: PublishTestResults@2 displayName: 'Publish test results' inputs: searchFolder: 'test-results' testResultsFormat: 'JUnit' testResultsFiles: 'e2e-junit-results.xml' mergeTestResults: true failTaskOnFailedTests: true testRunTitle: 'My End-To-End Tests' condition: succeededOrFailed()- task: PublishPipelineArtifact@1 inputs: targetPath: playwright-report artifact: playwright-report publishLocation: 'pipeline' condition: succeededOrFailed()
Note: The JUnit reporter needs to be configured accordingly via
import { defineConfig } from '@playwright/test';
export default defineConfig({ reporter: ['junit', { outputFile: 'test-results/e2e-junit-results.xml' }], });
in playwright.config.ts.
```

Azure Pipelines (sharded)

```
trigger:- mainpool: vmlImage: ubuntu-latest
strategy: matrix:
  chromium-1: project: chromium shardIndex: 1 shardTotal: 3
  chromium-2: project: chromium shardIndex: 2 shardTotal: 3
  chromium-3: project: chromium shardIndex: 3 shardTotal: 3
  firefox-1: project: firefox shardIndex: 1 shardTotal: 3
  firefox-2: project: firefox shardIndex: 2 shardTotal: 3
  firefox-3: project: firefox shardIndex: 3 shardTotal: 3
  webkit-1: project: webkit shardIndex: 1 shardTotal: 3
  webkit-2: project: webkit shardIndex: 2 shardTotal: 3
  webkit-3: project: webkit shardIndex: 3 shardTotal: 3
steps:- task: NodeTool@0 inputs: versionSpec: '18' displayName: 'Install Node.js'- script: npm ci displayName: 'npm ci'- script: npx playwright install --with-deps displayName: 'Install Playwright browsers'- script: npx playwright test --project=$(project) --shard=$(shardIndex)/$(shardTotal) displayName: 'Run Playwright tests'
```

Azure Pipelines (containerized)

```
trigger:- mainpool: vmlImage: ubuntu-latest
container: mcr.microsoft.com/playwright:v1.36.0-jammy
steps:- task: NodeTool@0 inputs: versionSpec: '18' displayName: 'Install Node.js'- script: npm ci displayName: 'npm ci'- script: npx playwright test displayName: 'Run Playwright tests'
```

CircleCI

Running Playwright on CircleCI is very similar to running on GitHub Actions. In order to specify the pre-built Playwright Docker image, simply modify the agent definition with `docker` in your config like so:

```
executors: pw-jammy-development:
  docker:
    - image: mcr.microsoft.com/playwright:v1.36.0-jammy
```

Note: When using the

docker agent definition, you are specifying the resource class of where playwright runs to the 'medium' tier here. The default behavior of Playwright is to set the number of workers to the detected core count (2 in the case of the medium tier). Overriding the number of workers to greater than this number will cause unnecessary timeouts and failures.

Sharding in CircleCI

Sharding in CircleCI is indexed with 0 which means that you will need to override the default parallelism ENV VARS. The following example demonstrates how to run Playwright with a CircleCI Parallelism of 4 by adding 1 to the CIRCLE_NODE_INDEX to pass into the --shard cli arg.

```
playwright-job-name:
  executor: pw-jammy-development
  parallelism: 4
  steps:
    - run:
        SHARD="$(( ${CIRCLE_NODE_INDEX} + 1 ))";
        npx playwright test -- --shard=${SHARD}/${CIRCLE_NODE_TOTAL}
```

Jenkins

Jenkins supports Docker agents for pipelines. Use the Playwright Docker image to run tests on Jenkins.

```
pipeline {
  agent { docker { image 'mcr.microsoft.com/playwright:v1.36.0-jammy' } }
  stages {
    stage('e2e-tests') {
      steps {
        sh 'ci'
        sh 'npx playwright test'
      }
    }
  }
}
```

Bitbucket Pipelines

Bitbucket Pipelines can use public Docker images as build environments. To run Playwright tests on Bitbucket, use our public Docker image (see Dockerfile).

```
image: mcr.microsoft.com/playwright:v1.36.0-jammy
GitLab CI
```

GitLab CI

To run Playwright tests on GitLab, use our public Docker image (see Dockerfile).

```
stages:
  - test
  tests:
    stage: test
    image: mcr.microsoft.com/playwright:v1.36.0-jammy
    script: ...
```

Sharding GitLab CI

GitLab CI supports sharding tests between multiple jobs using the parallel keyword. The test job will be split into multiple smaller jobs that run in parallel. Parallel jobs are named sequentially from job_name 1/N to job_name N/N.

```
stages:
  - test
  tests:
    stage: test
    image: mcr.microsoft.com/playwright:v1.36.0-jammy
    parallel: 7
    script:
      - npm ci
      - npx playwright test --shard=$CI_NODE_INDEX/$CI_NODE_TOTAL
```

GitLab CI also supports sharding tests between multiple jobs using the parallel:matrix option.

The test job will run multiple times in parallel in a single pipeline, but with different variable values for each instance of the job. In the example below, we have 2 PROJECT values, 10 SHARD_INDEX values and 1 SHARD_TOTAL value, resulting in a total of 20 jobs to be run.

```
stages:
  - test
  tests:
    stage: test
    image: mcr.microsoft.com/playwright:v1.36.0-jammy
    parallel:
      matrix:
        - PROJECT: ['chromium', 'webkit']
          SHARD_INDEX: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
          SHARD_TOTAL: 10
      script:
        - npm ci
        - npx playwright test --project=$PROJECT --shard=$SHARD_INDEX/$SHARD_TOTAL
```

Caching browsers

Caching browser binaries is not recommended, since the amount of time it takes to restore the cache is comparable to the time it takes to download the binaries. Especially under Linux, operating system dependencies need to be installed, which are not cacheable. If you still want to cache the browser binaries between CI runs, cache these directories in your CI configuration, against a hash of the Playwright version.

Debugging browser launches

Playwright supports the DEBUG environment variable to output debug logs during execution. Setting it to pw:browser is helpful while debugging.

```
Error: Failed to launch browser
errors.DEBUG=pw:browser
npx playwright test
```

Running headed

By default, Playwright launches browsers in headless mode. This can be changed by passing a flag when the browser is launched.

```
// Works across chromium, firefox and webkit
const { chromium } = require('playwright');
const browser = await chromium.launch({ headless: false });
```

On Linux agents, headed execution requires Xvfb to be installed.

Our Docker image and GitHub Action have Xvfb pre-installed. To run browsers in headed mode with Xvfb, add xvfb-run before the Node.js command.

```
xvfb-
```

run node index.js

Selenium Grid Playwright can connect to Selenium Grid Hub that runs Selenium 4 to launch Google Chrome or Microsoft Edge browser, instead of running browser on the local machine. Before connecting Playwright to your Selenium Grid, make sure that grid works with Selenium WebDriver. For example, run one of the examples and pass `SELENIUM_REMOTE_URL` environment variable. If webdriver example does not work, look for any errors at your Selenium hub/node/standalone output and search Selenium issues for a possible solution.

Starting Selenium Grid If you run distributed Selenium Grid, Playwright needs selenium nodes to be registered with an accessible address, so that it could connect to the browsers. To make sure it works as expected, set `SE_NODE_GRID_URL` environment variable pointing to the hub when running selenium nodes. # Start selenium node `SE_NODE_GRID_URL="http://<selenium-hub-ip>:4444" java -jar selenium-server-<version>.jar node`

Connecting Playwright to Selenium Grid To connect Playwright to Selenium Grid 4, set `SELENIUM_REMOTE_URL` environment variable pointing to your Selenium Grid Hub. Note that this only works for Google Chrome and Microsoft Edge. `SELENIUM_REMOTE_URL=http://<selenium-hub-ip>:4444 npx playwright test`

You don't have to change your code, just use your testing harness or `browserType.launch()` as usual. When using Selenium Grid Hub, you can skip browser downloads.

Passing additional capabilities If your grid requires additional capabilities to be set (for example, you use an external service), you can set `SELENIUM_REMOTE_CAPABILITIES` environment variable to provide JSON-serialized capabilities. `SELENIUM_REMOTE_URL=http://<selenium-hub-ip>:4444 SELENIUM_REMOTE_CAPABILITIES="{\"mygrid:options\": {\"os\":\"windows\", \"username\":\"John\", \"password\":\"secure\"}}\" npx playwright test`

Passing additional headers If your grid requires additional headers to be set (for example, you should provide authorization token to use browsers in your cloud), you can set `SELENIUM_REMOTE_HEADERS` environment variable to provide JSON-serialized headers. `SELENIUM_REMOTE_URL=http://<selenium-hub-ip>:4444 SELENIUM_REMOTE_HEADERS="{\"Authorization\":\"OAuth 12345\"}\" npx playwright test`

Detailed logs Run with `DEBUG=pw:browser*` environment variable to see how Playwright is connecting to Selenium Grid. `DEBUG=pw:browser* SELENIUM_REMOTE_URL=http://internal.grid:4444 npx playwright test`

If you file an issue, please include this log.

Using Selenium Docker One easy way to use Selenium Grid is to run official docker containers. Read more in [selenium docker images documentation](#). For experimental arm images, see [docker-seleniarm](#).

Standalone mode Here is an example of running selenium standalone and connecting Playwright to it. Note that hub and node are on the same localhost, and we pass `SE_NODE_GRID_URL` environment variable pointing to it.

First start Selenium. `docker run -d -p 4444:4444 --shm-size="2g" -e SE_NODE_GRID_URL="http://localhost:4444" selenium/standalone-chrome:4.3.0-20220726`

Alternatively for arm architecture `docker run -d -p 4444:4444 --shm-size="2g" -e SE_NODE_GRID_URL="http://localhost:4444" seleniarm/standalone-chromium:103.0`

Then run Playwright. `SELENIUM_REMOTE_URL=http://localhost:4444 npx playwright test`

Hub and nodes mode Here is an example of running selenium hub and a single selenium node, and connecting Playwright to the hub. Note that hub and node have different IPs,

and we pass SE_NODE_GRID_URL environment variable pointing to the hub when starting node containers. First start the hub container and one or more node containers.

```
docker run -d -p 4442-4444:4442-4444 --name selenium-hub selenium/hub:4.3.0-20220726
docker run -d -p 5555:5555 \ --shm-size="2g" \ -e
SE_EVENT_BUS_HOST=<selenium-hub-ip> \ -e
SE_EVENT_BUS_PUBLISH_PORT=4442 \ -e
SE_EVENT_BUS_SUBSCRIBE_PORT=4443 \ -e SE_NODE_GRID_URL="http://
<selenium-hub-ip>:4444" selenium/node-chrome:4.3.0-20220726
```

Alternatively for arm architecture

```
docker run -d -p 4442-4444:4442-4444 --name selenium-hub
seleniarm/hub:4.3.0-20220728
docker run -d -p 5555:5555 \ --shm-size="2g" \ -e
SE_EVENT_BUS_HOST=<selenium-hub-ip> \ -e
SE_EVENT_BUS_PUBLISH_PORT=4442 \ -e
SE_EVENT_BUS_SUBSCRIBE_PORT=4443 \ -e SE_NODE_GRID_URL="http://
<selenium-hub-ip>:4444" seleniarm/node-chromium:103.0
```

Then run Playwright.

```
SELENIUM_REMOTE_URL=http://<selenium-hub-ip>:4444 npx playwright
testSelenium3
```

Internally, Playwright connects to the browser using Chrome DevTools Protocol websocket. Selenium 4 exposes this capability, while Selenium 3 does not. This means that Selenium 3 is supported in a best-effort manner, where Playwright tries to connect to the grid node directly. Grid nodes must be directly accessible from the machine that runs Playwright.