Skip to main content

PlaywrightDocsAPI
Node.js

- Node.js
- Python
- Java
- .NET

Community
Search⌘K

- Getting Started
  - Installation
  - Writing tests
  - Generating tests
  - Running and debugging tests
  - Trace viewer
  - Setting up CI
- Getting started - VS Code
- Release notes
- Canary releases
- Playwright Test
  - Test configuration
  - Test use options
  - Annotations
  - Command line

  - Emulation
  - Fixtures
  - Global setup and teardown
  - Parallelism
  - Parameterize tests
  - Projects
  - Reporters
  - Retries
  - Sharding
  - Timeouts
  - TypeScript
  - UI Mode
  - Web server
- Guides
  - Library
  - Accessibility testing
  - Actions
  - Assertions
  - API testing
  - Authentication
  - Auto-waiting
  - Best Practices
  - Browsers
  - Chrome extensions
  - Clock
  - Components (experimental)
  - Debugging Tests
  - Dialogs
  - Downloads
  - Evaluating JavaScript
  - Events
  - Extensibility
  - Frames
  - Handles
  - Isolation
  - Locators
  - Mock APIs
  - Mock browser APIs
  - Navigations
  - Network
  - Other locators
  - Page object models
  - Pages
  - Screenshots
  - Visual comparisons
  - Test generator
  - Trace viewer
  - Videos
  - WebView2
- Migration

    - Migrating from Protractor
    - Migrating from Puppeteer
    - Migrating from Testing Library
- Integrations
    - Docker
    - Continuous Integration
    - Selenium Grid (experimental)
- Supported languages

On this page

Continuous Integration

Introduction

Playwright tests can be executed in CI environments. We have created sample configurations for common CI providers.

3 steps to get your tests running on CI:

1. **Ensure CI agent can run browsers**: Use our Docker image in Linux agents or install your dependencies using the CLI.
2. **Install Playwright**:

   ```
   # Install NPM packages
   npm ci

   # Install Playwright browsers and dependencies
   npx playwright install --with-deps
   ```

3. **Run your tests**:

   ```
   npx playwright test
   ```

Workers

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.ts
```
import { 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 in CI.

GitHub Actions

On push/pull_request

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.

.github/workflows/playwright.yml

```
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      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@v4
      if: ${{ !cancelled() }}
      with:
        name: playwright-report
        path: playwright-report/
        retention-days: 30
```

On push/pull_request (sharded)

GitHub Actions supports sharding tests between multiple jobs. Check out our sharding doc to learn more about sharding and to see a GitHub actions example of how to configure a job to run your tests on multiple machines as well as how to merge the HTML reports.

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.

.github/workflows/playwright.yml
```
name: Playwright Tests
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:v1.47.0-jammy
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 18
      - name: Install dependencies
        run: npm ci
      - name: Run your tests
        run: npx playwright test
        env:
          HOME: /root
```

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.

.github/workflows/playwright.yml
```
name: Playwright Tests
on:
  deployment_status:
jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    if: github.event.deployment_status.state == 'success'
    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      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 }}
```

Fail-Fast

Large test suites can take very long to execute. By executing a preliminary test run with the `--only-changed` flag, you can run test files that are likely to fail first. This will give you a faster feedback loop and slightly lower CI consumption while working on Pull Requests. To detect test files affected by your changeset, `--only-changed` analyses your suites' dependency graph. This is a heuristic and might miss tests, so it's important that you always run the full test suite after the preliminary test run.

```
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
      with:
        # Force a non-shallow checkout, so that we can reference
$GITHUB_BASE_REF.
        # See https://github.com/actions/checkout for more details.
        fetch-depth: 0
    - uses: actions/setup-node@v4
      with:
        node-version: 18
    - name: Install dependencies
      run: npm ci
    - name: Install Playwright Browsers
      run: npx playwright install --with-deps
    - name: Run changed Playwright tests
      run: npx playwright test --only-changed=$GITHUB_BASE_REF
      if: github.event_name == 'pull_request'
    - name: Run Playwright tests
      run: npx playwright test
    - uses: actions/upload-artifact@v4
      if: ${{ !cancelled() }}
      with:
        name: playwright-report
        path: playwright-report/
        retention-days: 30
```

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

1. 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.
2. Seeing other weird errors when launching Chromium? Try running your container with `docker run --cap-add=SYS_ADMIN` when developing locally.

3. 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:
- main

pool:
  vmImage: 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'
  env:
    CI: 'true'
```

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:
- main

pool:
  vmImage: 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
```

```yaml
  displayName: 'Run Playwright tests'
  env:
    CI: 'true'
- 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

```js
import { defineConfig } from '@playwright/test';

export default defineConfig({
  reporter: [['junit', { outputFile: 'test-results/e2e-junit-results.xml'
}]],
});
```

in `playwright.config.ts`.

Azure Pipelines (sharded)

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

strategy:
  matrix:
    chromium-1:
      project: chromium
      shard: 1/3
    chromium-2:
      project: chromium
      shard: 2/3
    chromium-3:
      project: chromium
      shard: 3/3
    firefox-1:
      project: firefox
      shard: 1/3
    firefox-2:
      project: firefox
      shard: 2/3
    firefox-3:
      project: firefox
      shard: 3/3
    webkit-1:
      project: webkit
      shard: 1/3
    webkit-2:
      project: webkit
      shard: 2/3
    webkit-3:
      project: webkit
      shard: 3/3
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=$(project) --shard=$(shard)
  displayName: 'Run Playwright tests'
  env:
    CI: 'true'
```

Azure Pipelines (containerized)

```
trigger:
- main

pool:
  vmImage: ubuntu-latest
container: mcr.microsoft.com/playwright:v1.47.0-noble

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'
  env:
    CI: 'true'
```

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:v1.47.0-noble
```

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:v1.47.0-noble' } }
   stages {
      stage('e2e-tests') {
         steps {
            sh 'npm 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:v1.47.0-noble
```

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:v1.47.0-noble
  script:
  ...
```

Sharding

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:v1.47.0-noble
  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 and 10 `SHARD` values, resulting in a total of 20 jobs to be run.

```
stages:
  - test

tests:
  stage: test
  image: mcr.microsoft.com/playwright:v1.47.0-noble
  parallel:
    matrix:
      - PROJECT: ['chromium', 'webkit']
        SHARD: ['1/10', '2/10', '3/10', '4/10', '5/10', '6/10', '7/10',
'8/10', '9/10', '10/10']
  script:
    - npm ci
    - npx playwright test --project=$PROJECT --shard=$SHARD
```

Google Cloud Build

To run Playwright tests on Google Cloud Build, use our public Docker image (see Dockerfile).

```
steps:
- name: mcr.microsoft.com/playwright:v1.47.0-noble
  script:
  ...
  env:
  - 'CI=true'
```

Drone

To run Playwright tests on Drone, use our public Docker image (see Dockerfile).

```
kind: pipeline
name: default
type: docker

steps:
  - name: test
    image: mcr.microsoft.com/playwright:v1.47.0-jammy
    commands:
      - npx playwright test
```

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. See in our Running tests guide how to run tests in headed mode.

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 actual command.

```
xvfb-run npx playwright test
```

Previous
Docker

Next
Selenium Grid (experimental)

- Introduction
- Workers
- CI configurations
    - GitHub Actions
    - Docker

    - Azure Pipelines
    - CircleCI
    - Jenkins
    - Bitbucket Pipelines
    - GitLab CI
    - Google Cloud Build
    - Drone
- Caching browsers
- Debugging browser launches
- Running headed

Learn

- Getting started
- Playwright Training
- Learn Videos
- Feature Videos

Community

- Stack Overflow
- Discord
- Twitter
- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors