

THE COMPLETE REFERENCE

The Playwright Glossary Every Automation Tester Needs

1,000 Essential Terms Explained Simply

1,000

Terms

25

Categories

Manjunath N P

manjunathnp.in · github.com/manjunathnp · linkedin.com/in/manjunathnp

The Playwright Glossary Every Automation Tester Needs

1,000 Essential Terms Explained Simply

Editorial rule

Named Playwright APIs and Playwright-specific behavior are aligned to the current official Playwright documentation. Supporting JavaScript, TypeScript, DOM, HTTP and accessibility concepts use relevant primary documentation and are not presented as Playwright APIs.

Author

Manjunath N P

Website

manjunathnp.in

GitHub

github.com/manjunathnp

LinkedIn

linkedin.com/in/manjunathnp

How to read this glossary

- Definitions are intentionally short: normally one sentence, occasionally two when a limitation, deprecation, or experimental status matters.
- Deprecated, discouraged, experimental, non-native, or potentially misleading concepts are called out directly.
- Exact Playwright API names are preserved so readers can search the official reference quickly.
- Supporting concepts appear only when they materially help with real Playwright automation.

Official documentation coverage

- Current named classes/interfaces listed in the Playwright JavaScript API reference: ****covered in this glossary****.
- Missing named classes/interfaces from that reference : ****0****.
- Additional current Test Runner concepts included: ****`test.abort`**** and ****named test locks****.

Status badges used in this glossary

API

EXPERIMENTAL

DEPRECATED

SUPPORTING

CONTENTS

Category index

01	Playwright Foundations & Terminology	25 terms	6
02	Installation, Setup & Configuration	25 terms	8
03	Browser, BrowserContext & Page	50 terms	10
04	Locators	65 terms	13
05	Selectors & Element Identification	30 terms	17
06	User Actions & Input	45 terms	20
07	Navigation & Page Loading	30 terms	23
08	Assertions & Expectations	60 terms	26
09	Auto-Waiting & Actionability	30 terms	30
10	Playwright Test & Test Structure	50 terms	33
11	Fixtures, Hooks & Test Lifecycle	50 terms	36
12	Projects, Devices & Emulation	35 terms	39
13	Authentication, Cookies & Storage	30 terms	42
14	Network, Requests, Responses & Routing	60 terms	45
15	API Testing	40 terms	49
16	Frames, Popups, Dialogs, Downloads & Uploads	40 terms	52
17	Debugging, Inspector, Trace Viewer & Codegen	35 terms	55
18	Reports, Screenshots, Videos & Test Artifacts	30 terms	58
19	Parallelism, Workers, Retries, Sharding & Timeouts	35 terms	61
20	CI/CD, Docker & Execution Environments	30 terms	64
21	Accessibility Testing	25 terms	67
22	Visual Testing	20 terms	69

23	JavaScript, TypeScript & Async Concepts	50 terms	71
24	DOM, HTML, Browser & Web Platform Concepts	55 terms	74
25	Framework Design, Patterns, Reliability & Flakiness	55 terms	77

CHAPTER

01

Playwright Foundations & Terminology

Scope

Playwright + testing foundations

Terms in this chapter

25

Primary references

<https://playwright.dev/docs/intro>

<https://playwright.dev/docs/browsers>

<https://playwright.dev/docs/best-practices>

- **Playwright**

An open-source browser automation library from Microsoft for testing web applications across Chromium, Firefox, and WebKit.

- **Playwright Test** API

The Playwright test runner for end-to-end testing, with fixtures, assertions, projects, parallel execution, retries, reporters, and other test features.

- **Browser automation**

Using software to control a browser and perform actions such as navigation, clicking, typing, and checking application behavior.

- **End-to-end testing**

Testing a user flow across multiple parts of an application, often from the browser UI through backend integrations.

- **Cross-browser testing**

Checking that the same application behavior works across different browser engines or browser configurations.

- **Headless browser**

A browser running without its normal visible window, commonly used for automated and CI execution.

- **Headed browser**

A browser running with a visible window, useful when observing or debugging automated interactions.

- **Test runner**

Software that discovers, schedules, runs, and reports tests. Playwright Test is Playwright's built-in test runner.

- **Test script**

Code that automates a test flow, including setup, actions, assertions, and cleanup where needed.

- **Test case**

A defined check of behavior under specific conditions, represented in Playwright Test by a `test()` block.

- **Test suite**

A collection of related tests, often organized by files, describe blocks, projects, or configuration.

- **Test scenario**

A higher-level description of behavior or a user flow that can be covered by one or more test cases.

- **Test step**

A named or logical action within a test. Playwright Test can represent steps explicitly with `test.step()`.

- **Test automation**

Using code and tools to execute repeatable test activities and verify expected behavior automatically.

- **Browser engine**

The browser implementation layer that renders pages and executes web-platform behavior, such as Chromium, Firefox, or WebKit.

- **Chromium**

An open-source browser project and engine family supported directly by Playwright.

- **Firefox**

A browser engine supported directly by Playwright for cross-browser automation.

- **WebKit**

A browser engine used by Safari and supported directly by Playwright for cross-browser automation.

- **Test isolation**

Keeping tests from affecting one another. Playwright commonly achieves this with a fresh `BrowserContext` for each test.

- **Test determinism**

The property of a test producing the same outcome when the application state and inputs are the same.

- **Test reproducibility**

The ability to repeat a test under equivalent conditions and obtain a consistent result.

- **Test reliability**

The degree to which automated tests consistently report correct results and avoid false failures or false passes.

- **Test flakiness**

Inconsistent test behavior where a test can pass and fail without a meaningful product change.

- **Web-first testing**

A testing style that interacts with the application through user-observable web behavior. In Playwright, user-facing locators and web-first assertions support this approach.

- **Automation framework**

The organized structure of test code, configuration, reusable components, data, utilities, reporting, and execution practices around automation.

CHAPTER

02

Installation, Setup & Configuration

Scope

Playwright tooling

Terms in this chapter

25

Primary references

<https://playwright.dev/docs/intro>

<https://playwright.dev/docs/test-configuration>

<https://playwright.dev/docs/test-cli>

- **Node.js**

A JavaScript runtime commonly used to install and run Playwright tests written in JavaScript or TypeScript.

- **npm**

The Node.js package manager commonly used to install Playwright packages and run project scripts.

- **Package manager**

A tool that installs, updates, and tracks project dependencies, such as npm, pnpm, or Yarn.

- **package.json**

The Node.js project manifest that records package metadata, scripts, and dependency declarations.

- **package-lock.json**

An npm-generated lock file that records resolved dependency versions so installations can be reproduced more consistently.

- **Playwright package**

The core playwright library package used for browser automation without necessarily using the Playwright Test runner.

- **@playwright/test**

The package that provides Playwright Test together with its test runner, fixtures, assertions, configuration, and reporting features.

- **Browser binaries**

The installed browser executables that Playwright launches for supported browser engines.

- **Playwright installation**

The process of adding Playwright packages and installing the browser binaries and system dependencies required by the project.

- **npm install**

An npm command that installs dependencies declared in package.json; Playwright browser binaries may still need a separate Playwright install command.

- **npm**

A package runner included with npm that can execute locally installed commands, such as npx playwright test.

- **Playwright CLI**

The command-line interface used for Playwright tasks such as running tests, installing browsers, opening reports, generating tests, and viewing traces.

- **playwright.config**

The Playwright Test configuration file, commonly playwright.config.ts, used to define projects, use options, timeouts, reporters, and other settings.

- **defineConfig**

A helper exported by @playwright/test that provides typed, structured Playwright configuration.

- **Test directory**

The folder where Playwright Test looks for test files; it can be configured through testDir.

- **testDir**

A configuration property that specifies the directory in which Playwright Test searches for tests.

- **testMatch**

A configuration pattern that controls which files are treated as test files.

- **testIgnore**

A configuration pattern that excludes matching files from test discovery.

- **baseUrl**

A Playwright use option that supplies a base address so relative URLs in navigation and related APIs can be resolved against it.

- **use configuration**

The shared options under use that are applied to browser contexts, pages, and built-in fixtures unless overridden.

- **Global configuration**

Configuration that applies broadly to the Playwright Test run rather than to a single test or project.

- **Environment variable**

A named value supplied by the operating environment, often used for URLs, credentials, feature flags, or CI-specific configuration.

- **dotenv**

A commonly used third-party Node.js package for loading key-value settings from a .env file into environment variables; it is not a Playwright feature.

- **Configuration override**

A more specific setting that replaces or changes a broader default, for example a project or test-level value overriding shared configuration.

- **Command-line option**

An option passed when running a command, such as --project, --headed, --grep, or --workers, to control that execution.

CHAPTER

03

Browser, BrowserContext & Page

Scope

Playwright core

Terms in this chapter

50

Primary references

<https://playwright.dev/docs/api/class-playwright>

<https://playwright.dev/docs/api/class-browser>

<https://playwright.dev/docs/api/class-browsercontext>

<https://playwright.dev/docs/api/class-page>

- **Browser** API

A Playwright object representing a launched or connected browser instance.

- **BrowserType** API

The Playwright API for a browser family such as chromium, firefox, or webkit, used to launch or connect to browsers.

- **BrowserContext** API

An isolated browser environment with separate cookies and storage. Playwright Test normally creates an isolated context for each test.

- **Page** API

A Playwright object representing a browser tab or page, used for navigation, locating elements, input, events, and other page interactions.

- **newContext**

`Browser.newContext()`, which creates a new isolated `BrowserContext` with configurable context options.

- **newPage**

A method that creates a new `Page`. In Playwright Test, the built-in page fixture usually provides the test page automatically.

- **launch**

`BrowserType.launch()`, which starts a browser process and returns a `Browser` object.

- **launchPersistentContext**

`BrowserType.launchPersistentContext()`, which launches a browser using a user-data directory and returns a persistent `BrowserContext`.

- **connect**

A `BrowserType` method for connecting Playwright to a browser server through Playwright's connection protocol.

- **connectOverCDP**

A Chromium-only connection method that attaches to an existing Chromium browser through the Chrome DevTools Protocol, with lower fidelity than Playwright's normal connection.

- **browser.close**

`Browser.close()`, which closes the browser and its contexts and pages.

- **context.close**

`BrowserContext.close()`, which closes the context and all pages belonging to it.

- **page.close**

`Page.close()`, which closes that page or tab.

- **BrowserServer** API

A Playwright object representing a browser server launched for remote connection, with methods to inspect its process, WebSocket endpoint, and lifecycle.

- **CDPSession** API

A Playwright session for sending raw Chrome DevTools Protocol commands and receiving CDP events. CDP sessions are supported only on Chromium-based browsers.

- **Isolated session**

A browser environment whose cookies, local storage, and other context state are separated from other contexts.

- **Incognito-like context**

A description often used for non-persistent `BrowserContexts` because they isolate browsing data similarly to private browsing; the precise Playwright object is `BrowserContext`.

- **Persistent context**

A `BrowserContext` backed by a user-data directory, created with `launchPersistentContext()`, so browser profile data can persist on disk.

- **User data directory**

The directory containing persistent browser profile data when using `launchPersistentContext()`.

- **Context options**

Settings supplied when creating a `BrowserContext`, such as viewport, locale, timezone, permissions, proxy, or storage state.

- **Clock** API

A Playwright API for controlling time and timers across a `BrowserContext`, useful for testing time-dependent behavior without waiting for real time.

- **Page event**

An event emitted by `Page`, such as request, response, popup, dialog, download, console, or pageerror.

- **Browser event**

An event emitted by `Browser`, for example when the browser becomes disconnected.

- **Context event**

An event emitted by `BrowserContext`, for example when a new page is created or a service worker appears.

- **Popup page**

A `Page` opened by another page, usually through `window.open()` or a link that opens a new browsing context.

- **Service worker**

A web-platform worker that can intercept requests and run independently of a page. Playwright can observe service workers at `BrowserContext` level.

- **Web worker**

A background web worker created by page JavaScript. Playwright exposes worker-related page events and Worker objects.

- **Dedicated worker**

A Web Worker dedicated to the script that created it, unlike a shared or service worker.

- **Page URL**

The current address of a Page, available through `page.url()`.

- **Page title**

The document title of a Page, available through `page.title()` and commonly asserted with `toHaveTitle()`.

- **Viewport**

The page area available for layout and rendering inside the browser context.

- **Viewport size**

The configured viewport width and height, usually expressed in CSS pixels.

- **Screen size**

The emulated screen width and height. It is separate from viewport size and is used only when viewport emulation is enabled.

- **Device scale factor**

A scale factor that changes the relationship between CSS pixels and device pixels, useful for high-DPI emulation.

- **Color scheme**

The emulated preferred color scheme, such as light, dark, or no preference.

- **Reduced motion**

The emulated reduced-motion preference exposed through media queries.

- **Forced colors**

The emulated forced-colors preference used to test interfaces under high-contrast or forced-color conditions.

- **Locale**

The browser context language and regional setting that can affect `navigator.language`, `Accept-Language`, and formatting behavior.

- **Timezone**

The timezone used by the browser context for date and time behavior.

- **Geolocation**

An emulated latitude and longitude supplied to the browser context, subject to geolocation permission.

- **Permissions**

Browser permissions granted to a context or origin, such as geolocation, clipboard access, or notifications where supported.

- **Extra HTTP headers**

Additional HTTP headers configured on a BrowserContext or Page and sent with requests made from that scope.

- **HTTP credentials**

Username and password credentials used by the browser for HTTP authentication challenges.

- **Offline mode**

A BrowserContext setting that makes the browser behave as if the network is offline.

- **JavaScript enabled**

A context option that controls whether JavaScript execution is enabled in pages.

- **Accept downloads**

A context option that controls whether browser downloads are accepted. Playwright Test normally manages downloads for the test context.

- **Bypass CSP**

A context option that bypasses the page's Content-Security-Policy enforcement; use only when a test specifically requires it.

- **Ignore HTTPS errors**

A context option that allows HTTPS certificate errors to be ignored. It can hide real TLS problems, so use it deliberately.

- **Proxy**

A network intermediary configured for a browser or context through Playwright proxy settings.

- **Client certificate**

A client-side TLS certificate configured for matching origins so the browser can authenticate to servers that request mutual TLS.

CHAPTER

04

Locators

Scope

Playwright core

Terms in this chapter

65

Primary references

<https://playwright.dev/docs/locators>

<https://playwright.dev/docs/api/class-locator>

<https://playwright.dev/docs/other-locators>

- **Locator** API

A Playwright object that represents how to find one or more elements and re-resolves the current DOM element when an action or assertion runs.

- **page.locator**

page.locator(), which creates a Locator from a selector and optional filtering conditions.

- **getByRole**

A recommended built-in locator that finds elements by ARIA role and, optionally, accessible name and state.

- **getByText**

A built-in locator that finds elements by rendered text, with support for substring, exact-string, and regular-expression matching.

- **getByLabel**

A built-in locator that finds form controls by the text of an associated label or accessible labeling attribute.

- **getByPlaceholder**

A built-in locator that finds input elements by their placeholder text.

- **getByAltText**

A built-in locator that finds elements, commonly images, by alternative text.

- **getTitle**

A built-in locator that finds elements by their title attribute.

- **getByTestId**

A built-in locator that finds elements by a configured test-id attribute, data-testid by default.

- **FrameLocator** API

A Playwright object for locating elements inside an iframe without manually switching frame context.

- **Locator chaining**

Combining locator operations so each step narrows or transforms the set of matched elements.

- **Locator filtering**

Narrowing locator matches using conditions such as text or the presence or absence of another relative locator.

- **filter**

locator.filter(), which returns a locator narrowed by has, hasNot, hasText, or hasNotText conditions.

- **has**

A locator filter condition that keeps elements containing a match for another relative locator.

- **hasText**

A locator filter condition that keeps elements containing specified text or matching a regular expression.

- **hasNot**

A locator filter condition that keeps elements that do not contain a match for another relative locator.

- **hasNotText**

A locator filter condition that keeps elements that do not contain specified text or regular-expression matches.

- **Locator strictness**

Playwright's rule that operations requiring one target fail when a locator resolves to multiple elements.

- **Strict mode violation**

An error raised when a strict locator operation matches more than one element instead of a unique target.

- **ElementHandle** API

A handle to one specific DOM element. Playwright generally recommends Locator instead because locators re-resolve elements and work better with auto-waiting.

- **Last locator**

A locator intentionally narrowed to the last matching element.

- **Nth locator**

A locator intentionally narrowed to an element at a zero-based position in the match list.

- **locator.first**

locator.first(), which returns a Locator for the first matched element.

- **locator.last**

locator.last(), which returns a Locator for the last matched element.

- **locator.nth**

locator.nth(index), which returns a Locator for the match at the given zero-based index.

- **Locator count**

The number of elements currently matching a locator.

- **locator.count**

locator.count(), which returns the number of elements that currently match the locator.

- **locator.all**

locator.all(), which returns an array of locators for the elements currently matching. It does not wait for a dynamic list to become stable.

- **locator.allTextContents**

locator.allTextContents(), which returns textContent values for all matching elements without assertion-style retrying.

- **locator.allInnerTexts**

locator.allInnerTexts(), which returns innerText values for all matching elements without assertion-style retrying.

- **locator.and**

locator.and(), which creates a locator that matches elements satisfying both locators.

- **locator.or**

locator.or(), which creates a locator matching elements from either locator; strictness still applies to single-element operations.

- **locator.isVisible**

locator.isVisible(), which immediately checks whether the matched element is visible; it is not an auto-retrying assertion.

- **locator.isHidden**

locator.isHidden(), which immediately checks whether the matched element is hidden; it is not an auto-retrying assertion.

- **locator.isEnabled**

locator.isEnabled(), which immediately checks whether the matched element is enabled.

- **locator.isDisabled**

locator.isDisabled(), which immediately checks whether the matched element is disabled.

- **locator.isEditable**

locator.isEditable(), which immediately checks whether the matched element is editable.

- **locator.isChecked**

locator.isChecked(), which checks the checked state of a checkbox, radio button, or supported role.

- **locator.isFocused**

locator.isFocused(), which checks whether the matched element currently has focus.

- **locator.boundingBox**

locator.boundingBox(), which returns the element's bounding rectangle relative to the main-frame viewport, or null when it is not visible.

- **locator.evaluate**

locator.evaluate(), which runs JavaScript in the page with the matched element passed to the provided function.

- **locator.evaluateAll**

locator.evaluateAll(), which runs JavaScript in the page with an array of all matched elements.

- **locator.innerText**

locator.innerText(), which returns the rendered innerText of the matched element.

- **locator.textContent**

locator.textContent(), which returns the DOM textContent of the matched node.

- **locator.inputValue**

locator.inputValue(), which returns the current value of an input, textarea, or select element.

- **locator.getAttribute**

locator.getAttribute(), which returns the value of a named DOM attribute or null when the attribute is absent.

- **locator.press**

locator.press(), which focuses the element and sends a key or keyboard shortcut such as Enter or Control+A.

- **locator.fill**

locator.fill(), which focuses an editable field and sets its value, triggering the appropriate input event.

- **locator.clear**

locator.clear(), which clears an editable input value.

- **locator.check**

locator.check(), which checks a checkbox or radio control after Playwright actionability checks.

- **locator.uncheck**

locator.uncheck(), which clears a checkbox control after Playwright actionability checks.

- **locator.selectOption**

locator.selectOption(), which selects one or more options in a select element by value, label, index, or element reference.

- **locator.setInputFiles**

locator.setInputFiles(), which sets files on a file input, supports multiple files where allowed, and can also clear a selection.

- **locator.hover**

locator.hover(), which moves the mouse over the element after relevant actionability checks.

- **locator.focus**

locator.focus(), which gives focus to the matched element.

- **locator.blur**

locator.blur(), which removes focus from the matched element.

- **locator.scrollIntoViewIfNeeded**

locator.scrollIntoViewIfNeeded(), which waits for actionability stability and scrolls the element into view when required.

- **locator.screenshot**

locator.screenshot(), which captures an image of the element represented by the locator.

- **locator.waitFor**

locator.waitFor(), which waits for the locator to satisfy an attached, detached, visible, or hidden state.

- **locator.dispatchEvent**

locator.dispatchEvent(), which programmatically dispatches a DOM event on the matched element rather than performing a real user action.

- **locator.dragTo**

locator.dragTo(), which performs a drag-and-drop operation from the source locator to a target locator.

- **Locator re-resolution**

The behavior where Playwright locators find the current matching DOM element again before each action, helping with re-rendered interfaces.

- **User-facing locator**

A locator based on information visible or meaningful to users, such as role, label, or text; Playwright recommends these for resilient tests.

- **Test-id locator**

A locator based on an explicit testing contract such as data-testid, useful when user-facing attributes are not suitable.

- **Accessible-name locator**

A locator that uses an element's computed accessible name, commonly together with getByRole().

CHAPTER

05

Selectors & Element Identification

Scope

Playwright + web platform

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/locators>

<https://playwright.dev/docs/api/class-selectors>

<https://playwright.dev/docs/other-locators>

- **Selector**

A query expression used to identify elements. Playwright locators can be created from selectors, though user-facing locator APIs are often preferred.

- **CSS selector**

A selector written with CSS selector syntax, supported by Playwright's locator engine.

- **XPath selector**

A selector written with XPath syntax. Playwright supports XPath, but DOM-structure-dependent XPath is usually less resilient than user-facing locators.

- **Text selector**

A text-based way of identifying elements. Prefer Playwright's `getByText()` API rather than relying on legacy selector syntax.

- **ID selector**

A CSS selector that targets an element by its id attribute, such as `#submit`.

- **Class selector**

A CSS selector that targets elements containing a class, such as `.menu-item`.

- **Attribute selector**

A CSS selector that matches elements by the presence or value of an attribute.

- **Pseudo-class**

A CSS selector feature that matches an element based on state or structural conditions, such as `:checked` or `:nth-child()`.

- **Pseudo-element**

A CSS construct such as `::before` or `::after` that represents a generated or special part of an element; it is not a normal DOM element target.

- **Descendant selector**

A CSS selector that matches descendants anywhere below another matched element.

- **Child selector**

A CSS selector that matches direct child elements using the `>` combinator.

- **Sibling selector**

A CSS selector using `+` or `~` to match elements based on sibling relationships.

- **Compound selector**

A selector that combines multiple simple selector conditions on the same element, such as `button.primary[disabled]`.

- **Selector engine**

The matching mechanism that interprets selector syntax. Playwright provides built-in selector engines and locator APIs.

- **Custom selector engine**

A user-registered selector engine created through `selectors.register()`; use only when built-in locators cannot express the requirement well.

- **Test ID**

An explicit attribute or value added primarily to give automation a stable way to identify an element.

- **data-testid**

The default HTML attribute used by `getByTestId()`, unless Playwright is configured to use another test-id attribute.

- **Selectors** API

The Playwright API for registering custom selector engines and configuring selector behavior such as the test-id attribute.

- **Accessible name**

The name exposed to assistive technologies for an element, computed from sources such as text, labels, alt text, and ARIA attributes.

- **Label text**

The visible or accessible text associated with a form control through a label relationship.

- **Placeholder text**

Text placed in an input as a hint when no value is present; `getByPlaceholder()` can locate by it.

- **Alt text**

Text that provides an alternative for content such as images and can be used by `getByAltText()`.

- **Title attribute**

The HTML title attribute, which Playwright can target with `getTitle()`.

- **Exact text match**

A text match that requires the complete normalized string rather than a substring match.

- **Partial text match**

A text match that accepts a substring within the normalized element text.

- **Regular expression selector**

Using a RegExp value in a Playwright locator or filter to match text or another supported string property by pattern.

- **Shadow DOM**

The DOM subtree encapsulated by a web component shadow root. Playwright locators pierce open shadow DOM for supported selector operations.

- **Closed shadow root**

A shadow root created in closed mode. Playwright locators do not pierce closed shadow roots.

- **DOM structure dependency**

A locator or selector that depends heavily on the current DOM hierarchy, making it more likely to break when markup changes.

- **Selector resilience**

The ability of a locator to keep identifying the intended element despite unrelated implementation or layout changes.

CHAPTER

06

User Actions & Input

Scope

Playwright core

Terms in this chapter

45

Primary references

<https://playwright.dev/docs/input>

<https://playwright.dev/docs/api/class-mouse>

<https://playwright.dev/docs/api/class-keyboard>

- **Click**

A pointer action that activates an element. `locator.click()` performs actionability checks before dispatching the click.

- **Double click**

A rapid two-click interaction, available through `locator.dblclick()`.

- **Right click**

A click using the secondary mouse button, commonly performed with `locator.click({ button: "right" })`.

- **Mouse** API

The page-level input API for coordinate-based mouse operations such as moving, pressing buttons, releasing buttons, and clicking.

- **Mouse move**

Moving the mouse pointer to page coordinates with the Mouse API.

- **Mouse down**

Pressing and holding a mouse button with `mouse.down()`, usually as part of lower-level pointer interactions.

- **Mouse up**

Releasing a mouse button with `mouse.up()`, completing a lower-level mouse interaction.

- **Keyboard** API

The page-level input API for sending keyboard input such as key presses, key-down events, and key-up events to the browser page.

- **Key press**

A single logical key or shortcut sent with `locator.press()` or `keyboard.press()`.

- **Key down**

Pressing a key without releasing it, using `keyboard.down()`.

- **Key up**

Releasing a key that was previously pressed, using `keyboard.up()`.

- **Typing**

Entering characters as keyboard input. For normal form fields, Playwright documentation recommends `locator.fill()` in most cases instead of simulating character-by-character typing.

- **Filling**

Setting the value of an editable field with `locator.fill()`, which focuses the element and triggers an input event.

- **Clearing input**

Removing the current value from an editable field, for example with `locator.clear()`.

- **Focus**

Making an element the active focused element, available through `locator.focus()`.

- **Blur**

Removing focus from an element, available through `locator.blur()`.

- **Hover**

Moving the pointer over an element with `locator.hover()`, after relevant actionability checks.

- **Check checkbox**

Setting a checkbox or radio control to checked, commonly with `locator.check()` or `locator.setChecked(true)`.

- **Uncheck checkbox**

Setting a checkbox to unchecked, commonly with `locator.uncheck()` or `locator.setChecked(false)`.

- **Select option**

Choosing one or more options from a select element with `locator.selectOption()`.

- **Upload file**

Assigning one or more files to a file input with `locator.setInputFiles()`, or through a `FileChooser` when needed.

- **Drag and drop**

Moving content from a source element to a destination. Playwright provides `locator.dragTo()` and lower-level mouse control.

- **Scroll**

Moving the page or an element's scroll position. Locator actions also scroll targets into view automatically when required.

- **Tap**

A touch-style pointer interaction. Playwright can emulate touch-capable devices and exposes `touchscreen.tap()` when touch is enabled.

- **Touchscreen** API

Playwright's Touchscreen API for tap interactions in contexts created with touch support.

- **Pointer action**

An interaction produced by a mouse, pen, or touch pointer. Playwright primarily exposes mouse and touchscreen APIs plus locator actions.

- **Modifier key**

A key such as Shift, Control, Alt, or Meta held while another input action occurs.

- **Keyboard shortcut**

A combination of keys sent together, such as Control+A, supported by `locator.press()` and `keyboard.press()`.

- **Enter key**

The Enter keyboard key, often used to submit forms or activate focused controls.

- **Tab key**

The Tab key, normally used by browsers to move focus forward through focusable elements.

- **Shift+Tab**

Shift combined with Tab, normally used to move keyboard focus backward.

- **Escape key**

The Escape key, commonly used by web applications to dismiss dialogs, menus, or other transient UI.

- **Space key**

The Space key, which can insert a space or activate certain focused controls according to browser behavior.

- **Meta key**

The Meta modifier key, typically Command on macOS and the Windows key on Windows.

- **Alt key**

The Alt modifier key used alone or as part of keyboard shortcuts.

- **Shift key**

The Shift modifier key used for capitalization, selection, or shortcuts.

- **Force action**

Using force: true on supported locator actions to skip selected non-essential actionability checks. It should be used deliberately because it can bypass real user constraints.

- **Trial action**

Using trial: true on supported actions to run actionability checks without performing the main action, useful for waiting until an element is actionable.

- **Position option**

An action option that specifies the coordinates within an element where a pointer action should occur.

- **Click count**

An option controlling how many clicks are issued as part of a pointer action.

- **Button option**

An option selecting the mouse button used for an action, such as left, right, or middle.

- **Delay option**

An input option that adds a delay between generated input events. Delays make tests slower and are rarely needed for normal automation.

- **No-wait-after option**

An older action option intended to change navigation waiting behavior. In current Playwright APIs it is deprecated or has no effect on many actions; explicit navigation synchronization is preferred.

- **Dispatch event**

Programmatically firing a DOM event with `locator.dispatchEvent()` rather than reproducing the full physical user interaction.

- **Synthetic event**

An event created and dispatched by script rather than originating from real hardware input; `locator.dispatchEvent()` is one way to produce one.

CHAPTER

07

Navigation & Page Loading

Scope

Playwright + web platform

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/navigations>

<https://playwright.dev/docs/api/class-page>

- **Navigation**

Changing the document or URL shown by a page or frame, whether through `page.goto()`, links, history, redirects, or application routing.

- **page.goto**

`page.goto()`, which navigates the page's main frame to a URL and normally waits for the configured load condition.

- **page.reload**

`page.reload()`, which reloads the current page and waits for the requested load condition.

- **page.goBack**

`page.goBack()`, which navigates to the previous history entry when one exists.

- **page.goForward**

`page.goForward()`, which navigates to the next history entry when one exists.

- **URL**

A Uniform Resource Locator identifying a web resource or location.

- **Relative URL**

A URL that omits some leading parts and is resolved against a base URL.

- **Absolute URL**

A complete URL containing the information needed to locate a resource independently of a base URL.

- **Navigation timeout**

The maximum time allowed for navigation-related operations before Playwright throws a `TimeoutError`.

- **Load state**

A page lifecycle milestone that Playwright can wait for, such as `domcontentloaded`, `load`, `commit`, or `networkidle`.

- **load**

The load state fired when the document's load event has completed.

- **commit load state**

The earliest Playwright load state, reached when a network response is received and the document starts loading.

- **networkidle**

The `networkidle` load state, reached after there are no network connections for a short period. Playwright documentation discourages using it as a test-readiness signal; prefer web assertions.

- **Document request**

The network request that retrieves the document for a page or frame navigation.

- **Main frame navigation**

A navigation of the page's main frame to a different URL or document.

- **Same-document navigation**

A URL change that keeps the same document, such as some History API or fragment changes.

- **Cross-document navigation**

A navigation that replaces the current document with a newly loaded document.

- **Redirect**

An HTTP or browser navigation step that sends the client from one URL to another.

- **Redirect chain**

The sequence of requests produced when one navigation follows one or more redirects.

- **History navigation**

Moving through the browser session history, such as with `page.goBack()` or `page.goForward()`.

- **Hash navigation**

A navigation that changes the URL fragment after `#`, typically without loading a new document.

- **Navigation response**

The Response associated with a successful document navigation where Playwright can return one; some navigations do not have a new network response.

- **Navigation failure**

A navigation that cannot complete because of an error, timeout, interruption, or another navigation taking over.

- **waitForURL**

`page.waitForURL()`, which waits until the main frame's URL matches a string, glob, regular expression, or predicate.

- **waitForLoadState**

`page.waitForLoadState()`, which waits until the page reaches a specified lifecycle state. It is often unnecessary because Playwright actions auto-wait.

- **waitForNavigation**

`page.waitForNavigation()`, an older API for waiting for main-frame navigation. Playwright marks it deprecated and recommends `waitForURL()` because `waitForNavigation()` is inherently racy.

- **Base URL resolution**

Resolving a relative URL against the configured `baseURL`, including `page.goto()` and other APIs that support base URL merging.

- **DOMContentLoaded**

The DOMContentLoaded lifecycle event, fired after the initial HTML has been parsed without waiting for all subresources.

- **Page load event**

The browser load event, fired after the document and dependent resources required by the event have finished loading.

- **Navigation interruption**

A navigation that is stopped or superseded, for example because another navigation starts before the first completes.

CHAPTER

08

Assertions & Expectations

Scope

Playwright Test

Terms in this chapter

60

Primary references

<https://playwright.dev/docs/test-assertions>

<https://playwright.dev/docs/api/class-genericassertions>

<https://playwright.dev/docs/api/class-locatorassertions>

<https://playwright.dev/docs/api/class-pageassertions>

<https://playwright.dev/docs/api/class-snapshotassertions>

- **Assertion**

A check that compares actual behavior or state with an expected condition.

- **expect**

The Playwright Test assertion function. It provides generic matchers and web-specific async matchers for Playwright objects.

- **GenericAssertions** API

The Playwright Test assertion interface for checking general JavaScript values with `expect()`, rather than page- or locator-specific state.

- **Auto-retrying assertion**

An async Playwright assertion that repeatedly checks until it passes or reaches its assertion timeout.

- **LocatorAssertions** API

The Playwright Test assertion interface for auto-retrying checks against a Locator, such as visibility, text, value, or state.

- **Web-first assertion**

A Playwright assertion designed around live web state and automatic retrying, such as `await expect(locator).toBeVisible()`.

- **Soft assertion**

An assertion that records its failure but allows the test to continue so additional checks can run; Playwright supports this with `expect.soft()`.

- **PageAssertions** API

The Playwright Test assertion interface for auto-retrying checks against a Page, such as its URL or title.

- **SnapshotAssertions** API

The Playwright Test assertion interface for comparing values against stored snapshots.

- **expect.soft**

Playwright's soft assertion form, which records assertion failures without immediately stopping the test.

- **expect.poll**

A polling assertion that repeatedly evaluates a supplied function until its returned value satisfies a matcher or the poll timeout is reached.

- **expect.toPass**

A retrying assertion for a block of code: `expect(async () => { ... }).toPass()` reruns the block until it succeeds or its configured timeout is reached.

- **toBeVisible**

An auto-retrying locator assertion that requires the element to be visible.

- **toBeHidden**

An auto-retrying locator assertion that requires the element to be hidden or detached.

- **toBeAttached**

An auto-retrying locator assertion that checks whether the element is connected to a Document or ShadowRoot, or the opposite when configured.

- **toBeChecked**

An auto-retrying locator assertion that checks the checked state of a checkbox, radio button, or supported role.

- **toBeDisabled**

An auto-retrying locator assertion that requires the matched element to be disabled.

- **toBeEditable**

An auto-retrying locator assertion that requires the matched element to be editable.

- **toBeEmpty**

An auto-retrying locator assertion that checks whether the element has no text content or an input has no value, according to matcher behavior.

- **toBeEnabled**

An auto-retrying locator assertion that requires the matched element to be enabled.

- **toBeFocused**

An auto-retrying locator assertion that requires the matched element to have focus.

- **toBeInViewport**

An auto-retrying locator assertion that checks whether the element intersects the viewport by the required ratio.

- **toContainText**

An auto-retrying locator assertion that checks whether the element text contains the expected string, pattern, or list values.

- **toHaveText**

An auto-retrying locator assertion that checks the element's text against the expected value or list.

- **toHaveValue**

An auto-retrying locator assertion that checks the current value of an input-like element.

- **toHaveValues**

An auto-retrying locator assertion that checks the selected values of a multi-select element.

- **toHaveAttribute**

An auto-retrying locator assertion that checks a named DOM attribute and, optionally, its expected value.

- **toHaveClass**

An auto-retrying locator assertion that checks the element's class attribute or class list against the expectation.

- **toHaveCSS**

An auto-retrying locator assertion that checks the computed value of a CSS property.

- **toHaveCount**

An auto-retrying locator assertion that checks the number of elements matching the locator.

- **toHaveId**

An auto-retrying locator assertion that checks the element's id attribute.

- **toHaveJSPROPERTY**

An auto-retrying locator assertion that checks a JavaScript property on the matched element.

- **toHaveRole**

An auto-retrying locator assertion that checks the computed ARIA role of the element.

- **toHaveAccessibleName**

An auto-retrying locator assertion that checks the element's computed accessible name.

- **toHaveAccessibleDescription**

An auto-retrying locator assertion that checks the element's computed accessible description.

- **toHaveAccessibleErrorMessage**

An auto-retrying locator assertion that checks the accessible error message associated with the element.

- **toHaveScreenshot**

A Playwright visual assertion that compares a page or locator screenshot with a stored reference image.

- **toMatchAriaSnapshot**

An assertion that compares a page or locator accessibility-tree representation against an ARIA snapshot template.

- **toHaveURL**

An auto-retrying Page assertion that checks the page URL against a string, regular expression, or predicate where supported.

- **toHaveTitle**

An auto-retrying Page assertion that checks the document title.

- **toBeOK**

An APIResponse assertion that checks whether the response status is in the successful HTTP range.

- **toEqual**

A generic matcher that recursively compares values for equality.

- **toStrictEqual**

A generic matcher that performs a stricter deep equality comparison, distinguishing some type and structural differences that toEqual() may ignore.

- **toContain**

A generic matcher that checks whether a string, array, or collection contains the expected value.

- **toContainEqual**

A generic matcher that checks whether an array contains an item deeply equal to the expected value.

- **toBe**

A generic matcher using Object.is-style identity/value comparison, best suited to primitive values or exact object identity.

- **toBeTruthy**

A generic matcher that checks whether a value is truthy according to JavaScript truthiness rules.

- **toBeFalsy**

A generic matcher that checks whether a value is falsy according to JavaScript truthiness rules.

- **toBeNull**

A generic matcher that checks whether a value is exactly null.

- **toBeUndefined**

A generic matcher that checks whether a value is undefined.

- **toBeDefined**

A generic matcher that checks whether a value is not undefined.

- **toBeGreaterThan**

A numeric matcher that checks whether the actual value is greater than the expected number.

- **toBeGreaterThanOrEqual**

A numeric matcher that checks whether the actual value is greater than or equal to the expected number.

- **toBeLessThan**

A numeric matcher that checks whether the actual value is less than the expected number.

- **toBeLessThanOrEqual**

A numeric matcher that checks whether the actual value is less than or equal to the expected number.

- **toMatch**

A matcher that checks whether a string matches a substring or regular-expression expectation, depending on the supplied value.

- **toThrow**

A matcher that checks whether calling a function throws an error matching the expectation.

- **Assertion timeout**

The maximum time an auto-retrying assertion waits for its condition before failing.

- **Custom matcher**

A project-defined matcher added to expect, usually with expect.extend(), to express domain-specific assertions.

- **Expectation message**

An optional custom message supplied to expect() so a failure explains the intent of the assertion more clearly.

CHAPTER

09

Auto-Waiting & Actionability

Scope

Playwright core

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/actionability>

<https://playwright.dev/docs/api/class-timeouterror>

- **Auto-waiting**

Playwright's behavior of waiting for required conditions before actions and retrying web assertions until they succeed or time out.

- **Actionability**

The set of checks Playwright performs before an element action so the interaction behaves like a real usable action.

- **Visibility**

The condition that an element has a non-empty bounding box and is not hidden by `visibility:hidden`; content opacity does not make it invisible to Playwright.

- **Stability**

The actionability condition that an element has maintained the same bounding box for the required animation frames and is not moving through an active animation.

- **Receives events**

The condition that the element is the target of pointer events at the action point rather than being obscured by another element.

- **Enabled state**

The state in which an element is not disabled according to the HTML or supported ARIA disabled semantics used by Playwright.

- **Editable state**

The state in which an element can accept editing, such as an enabled, non-readonly input or an editable content area.

- **Attached state**

The state in which an element is connected to a Document or ShadowRoot.

- **Detached state**

The state in which an element is no longer connected to the document or shadow tree.

- **Hidden state**

The state in which an element is not visible. A detached element is also considered hidden by relevant Playwright assertions.

- **Visible state**

The state in which an element satisfies Playwright's visibility rules.

- **Stable element**

An element whose bounding box is no longer changing across Playwright's stability check.

- **Element obscured**

An element whose target point cannot receive pointer events because another element or overlay is in the way.

- **Element covered**

Another way to describe a target blocked by a different element, causing the `Receives Events` actionability check to fail.

- **Animation stability**

Waiting for element movement or animation to settle so Playwright can safely perform an action that requires stability.

- **Scrolling into view**

Moving an element into the visible scroll area when an action requires it. Playwright performs this automatically for relevant actions.

- **Action timeout**

The time limit applied to Playwright actions before they fail with a `TimeoutError` when required conditions are not met.

- **Waiting**

The general act of delaying progress until a condition, event, state, or time limit is reached.

- **Explicit wait**

A wait requested directly in test code, such as `waitForURL()`, `waitForResponse()`, or `locator.waitFor()`. Prefer condition-based waits over fixed sleeps.

- **Implicit wait**

A Selenium-style configurable global element wait. Playwright does not use this model; locator actions use built-in auto-waiting and assertions can retry.

- **Polling**

Repeatedly checking a value or condition at intervals until it succeeds or a timeout expires.

- **Retryability**

The ability of Playwright locators and web assertions to re-check current page state instead of relying on one stale observation.

- **Retry loop**

A repeated evaluation cycle used until a success condition is met or a timeout ends the operation.

- **Wait condition**

The event or state that must become true before execution can safely continue.

- **TimeoutError** API

A Playwright-specific error class used when an operation exceeds its configured timeout.

- **Race condition**

A timing problem where correctness depends on which of two or more asynchronous operations happens first.

- **Action readiness**

The point at which the target satisfies the conditions required for the requested interaction.

- **Actionability check**

One individual requirement in Playwright's actionability process, such as Visible, Stable, Receives Events, Enabled, or Editable.

- **Element state**

A current property of an element relevant to testing, such as visible, hidden, checked, enabled, disabled, editable, attached, or focused.

- **Wait-free test**

An informal goal of writing tests that rely on Playwright's condition-based synchronization rather than arbitrary fixed delays.

CHAPTER

10

Playwright Test & Test Structure

Scope

Playwright Test

Terms in this chapter

50

Primary references

<https://playwright.dev/docs/api/class-test>

<https://playwright.dev/docs/api/class-testconfig>

<https://playwright.dev/docs/api/class-testproject>

<https://playwright.dev/docs/test-annotations>

- **test**

The Playwright Test function used to declare a test case.

- **test.describe**

test.describe(), which groups related tests and can provide a shared title or configuration scope.

- **test.only**

test.only(), which focuses a test so only focused tests run; it should not normally be committed to shared test suites.

- **test.skip**

test.skip(), which marks a test as skipped, optionally based on a condition or with a reason.

- **test.fixme**

test.fixme(), which marks a test as known to be broken or unsuitable to run until fixed.

- **test.fail**

test.fail(), which marks a test as expected to fail; an unexpected pass then becomes noteworthy to the runner.

- **test.slow**

test.slow(), which marks a test as slow and increases its test timeout relative to the normal timeout.

- **test.step**

test.step(), which groups test actions into a named step that appears in reports and traces.

- **test.info**

test.info(), which returns TestInfo for the currently running test.

- **test.use**

test.use(), which sets fixture or context options for a file or describe scope.

- **test.extend**

test.extend(), which creates an extended test object with additional or overridden fixtures and options.

- **test.describe.only**

test.describe.only(), which focuses every test in a describe group.

- **test.describe.skip**

test.describe.skip(), which skips a describe group.

- **test.describe.configure**

test.describe.configure(), which changes execution settings such as mode, retries, or timeout for a describe scope.

- **test.beforeEach**

test.beforeEach(), which registers a hook that runs before each test in its scope.

- **test.afterEach**

test.afterEach(), which registers a hook that runs after each test in its scope.

- **test.beforeAll**

test.beforeAll(), which registers a hook that runs once before tests in its scope for a worker execution.

- **test.afterAll**

test.afterAll(), which registers a hook that runs once after tests in its scope for a worker execution.

- **FullConfig** API

The fully resolved Playwright Test configuration available at runtime, including through testInfo.config and reporter APIs.

- **FullProject** API

The fully resolved configuration for one Playwright Test project after project settings have been processed.

- **Location** API

A source-code location containing file, line, and column information for tests, steps, annotations, or related runner data.

- **TestConfig** API

The top-level Playwright Test configuration shape used to define settings such as projects, timeouts, reporters, retries, and shared use options.

- **TestInfoError** API

Structured information about an error thrown during test execution, including details such as message, stack, cause, or the thrown value.

- **TestOptions** API

The Playwright Test options used to configure test and browser-context behavior, including settings such as browser, viewport, locale, trace, and video.

- **Test annotation**

Metadata attached to a test result, such as skip, fail, fixme, or custom annotations available through TestInfo.

- **Tag**

A label used to categorize tests for filtering or reporting; Playwright supports tags on tests and describe groups.

- **TestProject** API

The configuration interface describing one Playwright Test project and its project-specific execution settings.

- **TestInfo** API

The Playwright object exposing runtime information about the currently executing test, including project, retry, status, attachments, and output helpers.

- **TestResult** API

A result object representing one execution of a test, including status, duration, errors, attachments, and retry information.

- **TestStepInfo** API

Information and controls for the currently running test.step(), including step attachments, annotations, skipping, and its title path.

- **Suite** API

A reporter-facing object representing a group of discovered tests and nested suites.

- **TestCase** API

A reporter-facing object describing a discovered Playwright test and its metadata.

- **Test timeout**

The maximum duration allowed for a test, including its relevant fixture setup, test body, and hooks according to Playwright Test timeout rules.

- **TestError** API

Structured error information exposed by Playwright Test and reporters for a test failure.

- **Step timeout**

A timeout applied to an individual test.step() when configured.

- **TestStep** API

A reporter-facing record of an executed test step, including its title, timing, category, errors, and nested steps where applicable.

- **Test attachment**

A file or in-memory body attached to a test result through TestInfo for inclusion in reports.

- **Test output directory**

The project-specific directory where Playwright stores artifacts generated for a test run.

- **Output path**

A path created through TestInfo.outputPath() inside that test's output directory.

- **test.abort** API

A Playwright Test method that immediately aborts the currently running test by throwing an error and marking the test as failed.

- **Test lock** API

A named lock declared in test details that prevents tests sharing that lock name from running concurrently while unrelated tests can still run in parallel.

- **Fully parallel**

An execution setting that allows tests within files to run in parallel, rather than only running separate files concurrently.

- **Serial mode**

A describe execution mode where tests in the group run serially and later tests can be skipped after a failure. Playwright documentation generally recommends isolated tests instead.

- **Parallel mode**

A describe execution mode where tests in the group may run concurrently in separate workers.

- **Test dependency**

A project relationship requiring one project to complete successfully before a dependent project runs.

- **Setup project**

A Playwright project, commonly named setup, used as a dependency to prepare shared prerequisites such as authentication state.

- **Teardown project**

A Playwright project configured to run after dependent projects, useful for cleanup that should occur after project execution.

- **Global setup**

A configuration hook that runs once before the full test run. Project dependencies are often preferred when setup should appear in reports and traces.

- **Global teardown**

A configuration hook that runs once after the full test run. Project teardown is an alternative when project dependency behavior is desired.

- **Focused test**

A test or group marked with only so Playwright limits execution to focused items.

CHAPTER

11

Fixtures, Hooks & Test Lifecycle

Scope

Playwright Test

Terms in this chapter

50

Primary references

<https://playwright.dev/docs/test-fixtures>

<https://playwright.dev/docs/api/class-fixtures>

<https://playwright.dev/docs/api/class-testinfo>

- **Fixture**

A Playwright Test mechanism that prepares a value or environment for a test and cleans it up after use.

- **Built-in fixture**

A fixture provided by Playwright Test, such as page, context, browser, request, or browserName.

- **Custom fixture**

A project-defined fixture created with `test.extend()` to provide reusable setup, data, services, or helpers.

- **Test-scoped fixture**

A fixture created separately for each test and torn down after that test finishes.

- **Worker-scoped fixture**

A fixture created once per worker process and reused by tests that run in that worker when the worker environment matches.

- **Automatic fixture**

A fixture configured to run automatically even when the test does not request it explicitly.

- **Option fixture**

A fixture declared as a configurable option so projects or test scopes can supply values through use settings.

- **Fixture dependency**

A relationship where one fixture requests another fixture and therefore depends on its setup and value.

- **Fixture setup**

The work a fixture performs before calling `use()` to make its value available to the test or dependent fixture.

- **Fixture teardown**

The cleanup work a fixture performs after the `use()` callback returns.

- **Fixture timeout**

A timeout that limits fixture setup or teardown. Worker fixtures can have their own fixture timeout independent of the test timeout.

- **Fixture value**

The object, primitive, service, or helper supplied by a fixture to the test through the `use()` callback.

- **Fixture factory**

The function implementing a fixture, receiving dependencies and a use callback and optionally `TestInfo` or `WorkerInfo`.

- **Fixture extension**

Creating a new test object from an existing one with additional or overridden fixtures by using `test.extend()`.

- **Fixtures** API

The Playwright Test fixture type describing built-in and custom fixtures that can be supplied to tests and hooks.

- **mergeTests**

Combining fixtures from multiple extended test objects into one test object with `mergeTests()`.

- **mergeExpect**

Combining custom expect instances into a single expect object with `mergeExpect()`.

- **Base test**

The original test object that is extended to build a project-specific fixture layer, commonly imported as `test` as `base`.

- **Extended test**

A test object returned from `test.extend()` with custom fixtures or options available to its tests.

- **page fixture**

The built-in fixture that provides an isolated Page for the current test.

- **context fixture**

The built-in fixture that provides the isolated `BrowserContext` used by the current test.

- **browser fixture**

The built-in worker-scoped fixture that provides the Browser instance shared by tests in the worker.

- **browserName fixture**

A built-in fixture identifying the active browser project as `chromium`, `firefox`, or `webkit`.

- **request fixture**

The built-in fixture that provides an isolated `APIRequestContext` for making HTTP requests in tests.

- **playwright fixture**

The built-in worker-scoped fixture exposing the Playwright API object.

- **headless fixture**

A built-in option fixture controlling whether the browser runs without a visible UI.

- **viewport fixture**

A built-in option controlling the browser context viewport size, or disabling the fixed viewport when set accordingly.

- **Test lifecycle**

The sequence of fixture setup, hooks, test execution, teardown, and result processing for one test.

- **Worker lifecycle**

The lifetime of a Playwright worker process, including worker-scoped fixture setup, execution of assigned tests, and teardown.

- **Hook**

A function registered to run before or after tests at a defined lifecycle point.

- **Before-each hook**

A beforeEach hook that runs before every test within its applicable file or describe scope.

- **After-each hook**

An afterEach hook that runs after every test within its applicable file or describe scope.

- **Before-all hook**

A beforeAll hook that runs once before the tests in its scope within a worker execution.

- **After-all hook**

An afterAll hook that runs once after the tests in its scope within a worker execution.

- **Hook timeout**

The time available to a hook. Hooks participate in Playwright Test timeout rules and can have timeout adjusted through TestInfo where appropriate.

- **Hook failure**

A failure thrown from a lifecycle hook. Depending on hook type and execution mode, it can prevent or affect tests in that scope.

- **Setup phase**

The lifecycle period in which required fixtures and before hooks prepare the test environment.

- **Test phase**

The lifecycle period in which the test callback and its steps execute.

- **Teardown phase**

The lifecycle period in which after hooks and fixture cleanup run after the test body.

- **Fixture isolation**

The separation of fixture state between tests, especially for test-scoped fixtures such as page and context.

- **Fixture reuse**

Reusing worker-scoped fixture values across multiple tests in the same compatible worker process.

- **Worker reuse**

Reusing a Playwright worker for multiple test files when its worker-scoped fixture environment remains compatible.

- **Lazy fixture**

The fixture behavior where setup occurs only when the fixture is actually requested, unless it is configured as automatic.

- **Automatic setup**

Fixture configuration that starts setup without the test explicitly listing the fixture parameter.

- **Dependency injection**

A pattern where required objects are supplied to test functions through fixture parameters instead of being constructed directly inside the test.

- **use callback**

The callback a fixture calls to provide its value and suspend fixture teardown until the consumer has finished.

- **Fixture parameter**

A named value requested from the fixture object in a test or another fixture function.

- **WorkerInfo** API

The Playwright object containing worker-level information such as workerIndex, parallelIndex, project, and config.

- **Fixture option**

A fixture configured with option: true so its value can be supplied through test.use(), projects, or configuration.

- **Fixture override**

Redefining an existing fixture in an extended test object to change its setup or value while keeping the same fixture name.

CHAPTER

12

Projects, Devices & Emulation

Scope

Playwright Test + experimental device APIs

Terms in this chapter

35

Primary references

<https://playwright.dev/docs/test-projects>

<https://playwright.dev/docs/emulation>

<https://playwright.dev/docs/api/class-android>

- **Project**

A configured Playwright Test execution profile, commonly used to run the same tests with different browsers, devices, settings, or dependencies.

- **Test project**

Another name for a Playwright project defined in the `projects` array of the test configuration.

- **Project name**

The unique human-readable name assigned to a project and used for filtering, reports, and project dependencies.

- **Project configuration**

The settings attached to one project, including use options, test matching, retries, dependencies, metadata, and other project-level values.

- **Project dependency**

A relationship where one project must finish successfully before another dependent project starts.

- **Project teardown**

A project configured to run after projects that depend on it, typically for cleanup.

- **Project filtering**

Running or selecting only particular projects, for example with the `--project` command-line option.

- **Browser project**

A Playwright Test project whose main purpose is to run tests against a specific browser or browser engine configuration.

- **Mobile emulation**

Configuring a browser context to approximate a mobile device using viewport, user agent, touch, mobile behavior, and related settings.

- **Device descriptor**

A predefined Playwright device settings object containing values such as `userAgent`, `viewport`, `deviceScaleFactor`, `isMobile`, and `hasTouch`.

- **devices**

The exported Playwright device descriptor collection used to apply predefined desktop or mobile device settings.

- **Android** EXPERIMENTAL

Experimental Playwright support for Android automation, including Chrome for Android and Android WebView on connected devices or emulators.

- **AndroidDevice** EXPERIMENTAL

An experimental Playwright object representing a connected Android device and providing device, shell, input, browser, and WebView automation capabilities.

- **AndroidInput** EXPERIMENTAL

An experimental Android input API for gestures and input operations such as tap, drag, swipe, key press, and typing.

- **AndroidSocket** EXPERIMENTAL

An experimental Playwright object for communicating with a process launched on an `AndroidDevice` through a readable and writable socket.

- **AndroidWebView** EXPERIMENTAL

An experimental Playwright object representing an Android WebView that can be connected to and controlled through a regular Playwright Page.

- **Emulation**

Changing browser-context characteristics so tests can simulate device, locale, timezone, media, permissions, geolocation, or network conditions.

- **User agent**

The browser identification string exposed to websites. Playwright can override it at `BrowserContext` creation.

- **Device viewport**

The viewport dimensions supplied by a device descriptor or context settings.

- **Touch support**

Whether the context supports touch events, commonly supplied through a device descriptor or `hasTouch` option.

- **Mobile flag**

A context setting that enables mobile-specific browser behavior where supported; it is separate from touch support.

- **High-DPI display**

An emulated display with a `deviceScaleFactor` greater than one so multiple device pixels correspond to a CSS pixel.

- **Dark mode**

An emulated dark preferred color scheme exposed through `prefers-color-scheme`.

- **Light mode**

An emulated light preferred color scheme exposed through `prefers-color-scheme`.

- **Reduced motion emulation**

Emulating the `prefers-reduced-motion` media feature through context or page media settings.

- **Forced colors emulation**

Emulating the `forced-colors` media feature used by high-contrast display modes.

- **Locale emulation**

Running a context with a configured locale so language and regional browser behavior can be tested.

- **Timezone emulation**

Running a context with an emulated timezone identifier for date/time behavior.

- **Geolocation emulation**

Running a context with a specified latitude and longitude, with matching geolocation permission where required.

- **Permission emulation**

Granting, clearing, or configuring browser permissions for test contexts and origins.

- **Offline emulation**

Using BrowserContext offline mode to make network requests behave as if connectivity is unavailable.

- **Media emulation**

Changing CSS media features such as media type, color scheme, reduced motion, contrast, or forced colors with Playwright emulation APIs.

- **isMobile**

A context option indicating whether mobile-specific meta viewport and related behavior should be enabled for supported browsers.

- **hasTouch**

A context option indicating whether touch events are enabled.

- **Device profile**

A reusable group of device-emulation settings, usually selected from Playwright's devices registry and spread into project use options.

CHAPTER

13

Authentication, Cookies & Storage

Scope

Playwright + web platform

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/auth>

<https://playwright.dev/docs/api/class-credentials>

<https://playwright.dev/docs/api/class-webstorage>

- **Authentication**

Proving a user or client identity to an application so protected functionality can be accessed.

- **Authenticated state**

The browser or API state that represents a signed-in session, commonly including cookies and origin storage.

- **Storage state**

A serializable snapshot of browser authentication-related state that can be saved and reused to start contexts already signed in.

- **storageState**

The BrowserContext or APIRequestContext method that reads current storage state, and the corresponding context/test option used to restore it.

- **Cookie**

A small name-value record stored by the browser and sent with matching HTTP requests according to its attributes.

- **Session cookie**

A cookie without a long-lived expiration that is generally tied to the browser session.

- **Persistent cookie**

A cookie with an expiry or max-age that can survive browser restarts until it expires or is removed.

- **Cookie domain**

The host scope controlling which domains can receive a cookie.

- **Cookie path**

The URL path scope controlling which requests can receive a cookie.

- **SameSite**

A cookie attribute controlling when cookies are sent with cross-site requests, commonly Strict, Lax, or None.

- **HttpOnly**

A cookie attribute that prevents page JavaScript from reading the cookie through document.cookie.

- **Secure cookie**

A cookie attribute requiring the cookie to be sent only over secure HTTPS connections, subject to browser rules.

- **Set-Cookie**

The HTTP response header used by a server to create or update a cookie in the client.

- **Credentials** API

A virtual WebAuthn authenticator scoped to a BrowserContext, allowing tests to create, capture, restore, and use passkey credentials without physical hardware.

- **Local storage**

The origin-scoped Web Storage area whose data persists beyond a single page session. Playwright storage state can capture it.

- **Session storage**

The origin- and tab-scoped Web Storage area that lasts for the page session. Playwright does not directly persist sessionStorage through storageState.

- **IndexedDB**

A browser database API for structured client-side data. Playwright storage state can include IndexedDB data when requested or supported by the current API.

- **Origin**

The combination of scheme, host, and port that defines a web security and storage boundary.

- **Authentication setup**

The process of establishing sign-in state before tests, commonly in a setup project, then saving storageState for dependent projects.

- **Login reuse**

Reusing previously saved authenticated storage state so each test does not need to repeat the login UI flow.

- **Shared account**

Using the same authenticated account across tests when tests do not conflict through shared server-side state.

- **Per-worker account**

Using a separate account for each parallel worker so tests that modify server-side state do not interfere with one another.

- **Authentication fixture**

A custom fixture that creates, loads, or supplies authenticated state or authenticated pages/contexts to tests.

- **HTTP authentication**

Authentication performed through the HTTP protocol, such as Basic authentication, configured through httpCredentials for a browser context or request context.

- **Basic authentication**

The standard HTTP authentication scheme that sends a base64-encoded username and password in an Authorization header after or with a challenge flow.

- **WebStorage** API

A Playwright API exposing the current origin's localStorage and sessionStorage through asynchronous page storage interfaces.

- **Access token**

A credential token presented by a client to authorize access to protected resources or APIs.

- **CSRF token**

A token used by applications to defend against cross-site request forgery; test code may need to obtain and send it when exercising protected endpoints.

- **Logout**

Ending an authenticated session and removing or invalidating the related browser/server state.

- **State file**

A file, often JSON, containing serialized Playwright storage state. Because it can contain sensitive credentials, Playwright documentation recommends keeping it out of source control.

CHAPTER

14

Network, Requests, Responses & Routing

Scope

Playwright + HTTP/web platform

Terms in this chapter

60

Primary references

<https://playwright.dev/docs/network>

<https://playwright.dev/docs/mock>

<https://playwright.dev/docs/api/class-websocketroute>

- **Network request**

An HTTP, HTTPS, or supported browser network operation initiated to obtain or send a resource.

- **Network response**

Data returned by a server for a network request, including status, headers, and optionally a body.

- **Request** API

The Playwright object representing an individual network request made by a page or frame.

- **Response** API

The Playwright object representing the HTTP response associated with a request.

- **Route** API

The Playwright object passed to a routing handler so the request can be continued, fulfilled, aborted, fetched, or delegated.

- **Routing**

Intercepting matching browser network requests and deciding how they should proceed.

- **Request interception**

Capturing a browser request before it reaches its normal destination so a test can inspect or modify its handling.

- **Response mocking**

Replacing the real server response with test-controlled status, headers, and body, commonly with `route.fulfill()`.

- **Network mocking**

Controlling browser network behavior in tests by intercepting requests and returning, changing, blocking, or replaying responses.

- **page.route**

`page.route()`, which registers a routing handler for requests made by that page.

- **browserContext.route**

`browserContext.route()`, which registers a routing handler for matching requests in pages belonging to the context.

- **route.continue**

`route.continue()`, which sends the intercepted request to the network, optionally overriding request properties.

- **route.fulfill**

`route.fulfill()`, which completes an intercepted request with a supplied or fetched response.

- **route.abort**

`route.abort()`, which stops an intercepted request, optionally using a specified failure code.

- **route.fallback**

`route.fallback()`, which passes handling to an earlier matching route handler, optionally changing request properties.

- **route.fetch**

`route.fetch()`, which performs the request and returns an `APIResponse` so the handler can inspect or modify the real response before fulfilling.

- **unroute**

Removing previously registered route handlers with `page.unroute()` or `browserContext.unroute()`.

- **URL pattern**

The string, regular expression, predicate, or supported URL-matching expression used to select requests for routing or waiting.

- **Glob pattern**

A wildcard URL-matching syntax used by Playwright in APIs that accept glob patterns.

- **Request method**

The HTTP verb used by a request, such as GET, POST, PUT, PATCH, or DELETE.

- **Request URL**

The target URL of a network Request.

- **Request headers**

The HTTP header fields sent with a Request.

- **Request body**

The data sent as the request payload, when the method and request include a body.

- **POST data**

The request payload exposed by Playwright Request APIs for methods such as POST.

- **Response status**

The numeric HTTP status code returned by a Response.

- **Status text**

The textual phrase associated with an HTTP status response when available.

- **Response headers**

The HTTP header fields returned by a Response.

- **Response body**

The response payload returned by the server, which can be read as text, JSON, bytes, or through relevant Playwright APIs.

- **Content type**

The media type identified by the Content-Type response or request header, such as application/json or text/html.

- **Redirected request**

A request that resulted from following an HTTP redirect from another request.

- **Request timing**

Timing information associated with request processing, such as start and response-related timestamps exposed by Request timing data.

- **Resource type**

The browser classification of a request, such as document, stylesheet, image, media, font, script, xhr, fetch, or websocket.

- **Document resource**

A network resource whose type is document, normally the HTML document loaded for a frame navigation.

- **Stylesheet resource**

A resource classified by the browser as a stylesheet.

- **Image resource**

A resource classified by the browser as an image.

- **Media resource**

A resource classified by the browser as media, such as audio or video.

- **Font resource**

A resource classified by the browser as a font.

- **Script resource**

A resource classified by the browser as a script.

- **XHR**

XMLHttpRequest, a browser API for asynchronous HTTP requests; Playwright reports these with resource type xhr.

- **Fetch request**

A request initiated through the browser Fetch API; Playwright reports these with resource type fetch.

- **WebSocket** API

A persistent bidirectional connection upgraded from HTTP. Playwright can observe WebSocket creation, frames, errors, and closure.

- **WebSocket frame**

A message sent or received over a WebSocket connection.

- **Server-sent events**

A one-way HTTP streaming mechanism where a server pushes events to the browser over a long-lived response.

- **Service worker interception**

Network handling performed by a page's Service Worker. Service Workers can hide requests from ordinary page/context routing unless configured appropriately.

- **HAR**

HTTP Archive, a structured format containing recorded request and response information.

- **HAR recording**

Capturing network traffic into a HAR file so it can later be inspected or replayed.

- **HAR replay**

Using previously recorded HAR entries to serve matching requests instead of relying on the live backend.

- **routeFromHAR**

page.routeFromHAR() or browserContext.routeFromHAR(), which serves or updates requests using a HAR file.

- **Network event**

An event emitted as browser network activity progresses, such as request, response, requestfinished, or requestfailed.

- **request event**

The Page or BrowserContext request event emitted when a request is issued.

- **response event**

The response event emitted when response status and headers are received.

- **requestfinished event**

The event emitted when the response body has finished downloading and the request completes successfully from the HTTP transport perspective.

- **requestfailed event**

The event emitted when a request fails at the network level. HTTP error statuses such as 404 still normally produce requestfinished, not requestfailed.

- **waitForRequest**

page.waitForRequest(), which waits for a request matching a URL, pattern, or predicate.

- **waitForResponse**

page.waitForResponse(), which waits for a response matching a URL, pattern, or predicate.

- **WebSocketRoute** API

A Playwright object used when routing WebSocket connections so tests can mock, inspect, modify, or proxy WebSocket communication.

- **Offline network**

Network behavior when a BrowserContext has offline mode enabled.

- **HTTP cache**

The browser's caching of HTTP resources according to response headers and browser rules. Routing can affect caching behavior.

- **CORS**

Cross-Origin Resource Sharing, the HTTP mechanism that lets servers permit selected cross-origin browser requests.

- **Preflight request**

An automatic CORS OPTIONS request a browser can send before certain cross-origin requests to check allowed methods, headers, and origin.

CHAPTER

15

API Testing

Scope

Playwright API testing

Terms in this chapter

40

Primary references

<https://playwright.dev/docs/api-testing>

<https://playwright.dev/docs/api/class-apirequest>

<https://playwright.dev/docs/api/class-apirequestcontext>

<https://playwright.dev/docs/api/class-apireponse>

- **API testing**

Testing HTTP services directly by sending requests and validating status, headers, body, side effects, and contracts without relying on the browser UI.

- **APIRequestContext** API

Playwright's HTTP client context used to send API requests and maintain related cookies and configuration.

- **APIResponse** API

Playwright's object representing an HTTP response returned by APIRequestContext.

- **apiRequest.newContext**

apiRequest.newContext(), which creates an isolated APIRequestContext with options such as baseURL, headers, credentials, proxy, timeout, or storage state.

- **request.get**

APIRequestContext.get(), a convenience method for sending an HTTP GET request.

- **request.post**

APIRequestContext.post(), a convenience method for sending an HTTP POST request.

- **request.put**

APIRequestContext.put(), a convenience method for sending an HTTP PUT request.

- **request.patch**

APIRequestContext.patch(), a convenience method for sending an HTTP PATCH request.

- **request.delete**

APIRequestContext.delete(), a convenience method for sending an HTTP DELETE request.

- **request.head**

APIRequestContext.head(), a convenience method for sending an HTTP HEAD request.

- **request.fetch**

APIRequestContext.fetch(), the general request method that can send a specified method and request options.

- **APIRequest** API

The Playwright API object used to create isolated APIRequestContext instances for sending HTTP requests outside a browser page.

- **Global request context**

An isolated APIRequestContext created directly with request.newContext(), independent of a BrowserContext unless state is explicitly shared.

- **Context-bound request**

The APIRequestContext exposed through browserContext.request or page.request, which shares cookie storage with that BrowserContext.

- **API base URL**

The baseURL configured for an APIRequestContext so relative request URLs can be resolved against it.

- **API headers**

Default or per-request HTTP headers sent with APIRequestContext requests.

- **API authentication**

Credentials or tokens supplied to authenticate API requests, such as HTTP credentials or authorization headers.

- **API cookie sharing**

The behavior where a context-bound APIRequestContext reads cookies from and updates cookies in its associated BrowserContext.

- **API storage state**

Serialized cookies and origin storage that can be exchanged between APIRequestContext and BrowserContext to reuse authentication.

- **API request body**

Data sent with an API request, commonly as JSON data, form fields, multipart data, or raw content.

- **JSON payload**

A request body encoded as JSON, commonly supplied through the data option and sent with an appropriate content type.

- **Form payload**

URL-encoded form data supplied through the form request option.

- **Multipart request**

A multipart/form-data request, commonly used for file uploads or mixed form fields and file content.

- **Query parameter**

A name-value value encoded in the URL query string, such as ?page=2.

- **Path parameter**

A value embedded in the URL path itself, such as the 42 in /users/42. Playwright does not provide special path-parameter syntax; the URL is constructed by test code.

- **API response body**

The payload returned by an APIResponse, available through json(), text(), or body().

- **API status code**

The numeric HTTP status code available through `APIResponse.status()`.

- **API response headers**

The HTTP response headers returned by the API, available through `APIResponse` header methods.

- **API timeout**

The maximum time allowed for an `APIRequestContext` request before it fails with a timeout.

- **API retry**

Retrying an HTTP request after eligible network errors. Playwright supports `maxRetries` for selected connection errors; it is not a general retry of arbitrary HTTP status codes.

- **failOnStatusCode**

An API request option that makes Playwright throw when the response status is outside the successful range instead of returning the response normally.

- **maxRedirects**

An API request option limiting how many HTTP redirects Playwright follows.

- **maxRetries**

An API request option controlling retries for supported network errors such as connection reset; it does not retry based on HTTP response status.

- **Response JSON**

`APIResponse.json()`, which parses the response body as JSON.

- **Response text**

`APIResponse.text()`, which reads the response body as text.

- **Response buffer**

`APIResponse.body()`, which returns the response body as a `Buffer`.

- **Response disposal**

`APIResponse.dispose()`, which releases the response body retained in memory by the `APIRequestContext` when it is no longer needed.

- **APIResponseAssertions** API

The Playwright assertion interface for `APIResponse` objects, including assertions such as `toBeOK()`.

- **API test fixture**

The built-in request fixture that provides an isolated `APIRequestContext` to each Playwright Test test.

- **UI and API hybrid test**

A test that combines direct API operations with browser UI interactions, for example creating test data through an API and verifying it through the page.

CHAPTER

16

Frames, Popups, Dialogs, Downloads & Uploads

Scope

Playwright + web platform

Terms in this chapter

40

Primary references

<https://playwright.dev/docs/frames>

<https://playwright.dev/docs/input>

<https://playwright.dev/docs/api/class-dialog>

<https://playwright.dev/docs/api/class-download>

- **Frame** API

A Playwright object representing one document frame within a page. A page always has a main frame and may contain child frames.

- **Main frame**

The top-level Frame associated with the page itself, available through `page.mainFrame()`.

- **Child frame**

A Frame nested under another frame, commonly created by an `iframe` element.

- **iframe**

An HTML element that embeds another browsing context and document inside the current page.

- **Frame locator**

A locator-style Playwright object used to find and interact with elements inside an `iframe`.

- **Nested frame**

An `iframe` located inside another frame, producing multiple levels in the page frame tree.

- **Frame URL**

The current URL loaded in a Frame, available through `frame.url()`.

- **Frame name**

The frame name, commonly derived from the `iframe` name attribute and available through `frame.name()`.

- **Frame navigation**

A navigation occurring inside a Frame, including the main frame or a child frame.

- **Frame attachment**

The lifecycle event in which a new Frame becomes attached to the page.

- **Frame detachment**

The lifecycle event in which a Frame is removed from the page.

- **Popup**

A new Page opened from another page, usually through `window.open()` or a `target=_blank` link.

- **Popup event**

The page popup event emitted when an action creates a popup Page.

- **New tab**

A newly opened browser Page represented as a separate tab in a typical browser UI.

- **New window**

A newly opened browser Page represented as a separate window by the browser or operating environment; Playwright still exposes it as a Page.

- **Dialog** API

A browser modal created by JavaScript APIs such as `alert()`, `confirm()`, `prompt()`, or `beforeunload`.

- **Alert dialog**

A JavaScript `alert()` dialog containing a message and an OK-style acknowledgement.

- **Confirm dialog**

A JavaScript `confirm()` dialog that lets script receive an accept or dismiss result.

- **Prompt dialog**

A JavaScript `prompt()` dialog that can accept text input before being accepted or dismissed.

- **Beforeunload dialog**

A dialog associated with leaving a page that has registered a `beforeunload` handler.

- **dialog.accept**

`dialog.accept()`, which accepts a browser dialog and can optionally provide prompt text.

- **dialog.dismiss**

`dialog.dismiss()`, which dismisses a browser dialog.

- **Download** API

A file transfer initiated by a page and represented in Playwright by a Download object.

- **Download event**

The page download event emitted when a download starts and a Download object becomes available.

- **Suggested filename**

The filename suggested by the server or browser for a downloaded file, available through `download.suggestedFilename()`.

- **Download path**

The temporary filesystem path for a completed download where supported, available through `download.path()`.

- **Download saveAs**

`download.saveAs()`, which waits for a download if necessary and copies it to a specified path.

- **Download cancel**

`download.cancel()`, which cancels an in-progress download.

- **Download failure**

The error message associated with a failed download, available through `download.failure()`; null indicates successful completion.

- **Upload**

Supplying one or more files to an application through a file input or file chooser.

- **File chooser**

A browser file-selection prompt. Playwright can wait for the `page.filechooser` event and control the resulting `FileChooser` object.

- **FileChooser** API

The Playwright object used to set files after a page opens a file chooser.

- **filechooser event**

The page event emitted when a file chooser is requested.

- **Input file**

A file assigned to an `input[type=file]` element with `locator.setInputFiles()` or `FileChooser.setFiles()`.

- **Multiple file upload**

Setting more than one file on a file input that supports multiple file selection.

- **In-memory file**

A file supplied directly from memory using an object containing `name`, `mimeType`, and `Buffer` content instead of a disk path.

- **File payload**

The in-memory object used by Playwright to describe uploaded content, including its `name`, `MIME` type, and `buffer`.

- **File name**

The filename exposed to the application for an uploaded or downloaded file.

- **MIME type**

The media type describing file content, such as `text/plain` or `image/png`.

- **Temporary file**

A short-lived file created for intermediate use. Playwright downloads are stored in a temporary location and are removed when their `BrowserContext` closes unless saved elsewhere.

CHAPTER

17

Debugging, Inspector, Trace Viewer & Codegen

Scope

Playwright tooling

Terms in this chapter

35

Primary references

<https://playwright.dev/docs/debug>

<https://playwright.dev/docs/trace-viewer>

<https://playwright.dev/docs/codegen>

<https://playwright.dev/docs/api/class-debugger>

<https://playwright.dev/docs/api/class-coverage>

- **Debugging**

The process of investigating why a test or application behaved differently from expectation.

- **Playwright Inspector**

Playwright's interactive debugging tool that can pause execution, inspect actions, and help explore locators.

- **PWDEBUG**

An environment variable used to launch Playwright in debugging mode, commonly opening the Inspector and adjusting timeout behavior.

- **Debug mode**

Running tests with debugging aids enabled so execution can be observed and stepped through.

- **Step-through debugging**

Executing a test action by action while inspecting current state and control flow.

- **Debugger** API

The Playwright API for controlling debugger execution, including pausing, inspecting the current pause, stepping to the next action, resuming, and running to a source location.

- **VS Code debugger**

Using the Visual Studio Code debugger integration to run and inspect Playwright tests from the editor.

- **ConsoleMessage** API

A Playwright object representing a browser or Electron console message, with access to its text, type, arguments, and source location.

- **page.pause**

`page.pause()`, which pauses script execution and opens Playwright Inspector when debugging conditions allow.

- **Codegen**

Playwright's code generator, launched with the `codegen` command, which records browser actions and generates test code and locators.

- **Locator picker**

The interactive selector tool in Inspector or codegen used to point at page elements and generate locator suggestions.

- **Recorded action**

An interaction captured by codegen, such as clicking, filling, checking, or navigating.

- **Generated test**

Test code produced by Playwright codegen from recorded browser interactions and assertions.

- **Trace**

A recorded debugging artifact containing actions and supporting information that can be opened in Trace Viewer.

- **Tracing** API

The process or Playwright API feature that records browser operations and related diagnostic data into a trace.

- **Trace Viewer**

Playwright's graphical tool for exploring a recorded trace, including actions, DOM snapshots, logs, source, console, errors, and network information.

- **trace.zip**

The common archive file containing a saved Playwright trace.

- **Action snapshot**

The recorded page state associated with a traced action, allowing before/after inspection in Trace Viewer.

- **DOM snapshot**

A captured DOM state inside a trace that Trace Viewer can render and inspect interactively.

- **Screenshot snapshot**

A screenshot recorded as part of tracing for visual inspection of test execution.

- **Network snapshot**

The network information stored in a trace and displayed in Trace Viewer, including requests, status, timing, and headers.

- **Trace action**

An individual operation recorded in a trace, such as a locator action, navigation, or test step.

- **Trace timeline**

The chronological visualization in Trace Viewer used to move through actions and inspect state over time.

- **Trace metadata**

Diagnostic information about the traced test, such as browser, viewport, duration, and other execution details.

- **Trace attachment**

A file or body attached to the test result and visible through reporting or the Trace Viewer attachments area.

- **Call log**

The detailed action log showing what Playwright attempted and which checks or substeps occurred.

- **Logger** DEPRECATED

Deprecated Playwright logging interface for a pluggable logger sink. The official documentation recommends tracing instead because logger output is incomplete.

- **Console message**

A message produced by page JavaScript through browser console APIs and exposed through Playwright console events.

- **Page error**

An uncaught exception from page JavaScript exposed through the `pageerror` event.

- **WebError** API

A Playwright object representing an unhandled exception thrown in a page and emitted through the `BrowserContext.weberror` event.

- **Stack trace**

The sequence of function calls captured when an error is thrown, used to identify the path to the failure.

- **Source location**

The file, line, and column associated with a test, step, or error in source code.

- **UI Mode**

Playwright Test's interactive UI for running, filtering, watching, and debugging tests with time-travel-style inspection.

- **Coverage** API

A Playwright API for collecting JavaScript and CSS usage information from a page. Coverage APIs are supported only on Chromium-based browsers.

- **JSHandle** API

A Playwright reference to a JavaScript object or value inside the page, allowing evaluation against that value until the handle is disposed.

CHAPTER

18

Reports, Screenshots, Videos & Test Artifacts

Scope

Playwright Test/tooling

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/test-reporters>

<https://playwright.dev/docs/screenshots>

<https://playwright.dev/docs/videos>

<https://playwright.dev/docs/api/class-screencast>

- **Reporter** API

A component that receives test-run events and produces human-readable or machine-readable output.

- **HTML reporter**

Playwright's built-in HTML reporter, which creates an interactive report with tests, errors, steps, attachments, and artifacts.

- **List reporter**

A built-in reporter that prints one line or structured status output for tests as they execute.

- **Line reporter**

A built-in concise terminal reporter that prints each test result on a single line.

- **Dot reporter**

A compact reporter that prints a character for each test outcome, useful for low-noise CI logs.

- **JSON reporter**

A built-in reporter that writes structured test results as JSON.

- **JUnit reporter**

A built-in reporter that writes test results in JUnit-style XML for CI and test-management integrations.

- **GitHub reporter**

A reporter intended to produce GitHub Actions annotations and CI-friendly failure output.

- **Custom reporter**

A project-defined reporter implementing Playwright's reporter interface to process test-run events in a custom way.

- **Report output**

The files or console output produced by a reporter for a completed test run.

- **Test report**

The summarized record of executed tests, statuses, durations, errors, steps, metadata, and related artifacts.

- **Test artifact**

A file or data item retained from a test run, such as a screenshot, trace, video, log, or custom attachment.

- **Attachment**

A named file or in-memory body associated with a test result through TestInfo and made available to reporters.

- **Screenshot**

An image capture of a page or element created with Playwright screenshot APIs.

- **Full-page screenshot**

A screenshot capturing the full scrollable page rather than only the current viewport.

- **Element screenshot**

A screenshot limited to the element matched by a Locator.

- **Screenshot mask**

A screenshot option that overlays specified locator regions so dynamic or sensitive areas do not affect the image in the usual way.

- **Screenshot animation**

The handling of animations during screenshot capture. Playwright screenshot options can disable animations to improve consistency.

- **Video** API

A recording of browser-page output captured for a BrowserContext when video recording is enabled.

- **Video recording**

Collecting page video during a test by enabling Playwright video settings on the context or test use configuration.

- **Video retention**

The policy controlling when recorded videos are kept, such as always, on failure, or on first retry depending on configured Playwright Test video mode.

- **Trace artifact**

A saved Playwright trace associated with a test result and typically opened through the HTML report or Trace Viewer.

- **Output directory**

The configured folder where Playwright Test writes per-test run artifacts and result files.

- **Snapshot directory**

The directory or path pattern used for stored snapshots such as visual or ARIA snapshot references.

- **Test results directory**

The output area containing test-run result artifacts, commonly controlled by outputDir and reporter-specific output settings.

- **Blob reporter**

Playwright's blob reporter, which writes report data intended to be merged later, especially from sharded runs.

- **Blob report**

The archive/data produced by the blob reporter for later report merging.

- **Merge reports**

The Playwright CLI process that combines blob reports from separate runs or shards into a consolidated report.

- **Report metadata**

Additional structured information attached to a run or project and exposed to reporters where supported.

- **Screencast** API

A Playwright interface for capturing page screencast frames, including recording video to a file or receiving JPEG frames through a callback.

CHAPTER

19

Parallelism, Workers, Retries, Sharding & Timeouts

Scope

Playwright Test

Terms in this chapter

35

Primary references

<https://playwright.dev/docs/test-parallel>

<https://playwright.dev/docs/test-retries>

<https://playwright.dev/docs/test-sharding>

<https://playwright.dev/docs/test-timeouts>

- **Parallelism**

Running multiple tests at the same time. Playwright Test uses worker processes to execute test files concurrently by default.

- **Worker** API

An execution unit used by Playwright Test to run assigned tests. In documentation, the worker is an independent operating-system process.

- **Worker process**

The OS process created and orchestrated by Playwright Test to run tests with its own browser and worker-scoped fixtures.

- **Worker index**

A worker identifier that increases when a worker is replaced; available through TestInfo or environment variables.

- **Parallel worker index**

The stable zero-based parallel slot assigned among configured workers. A restarted worker keeps the same parallel index but receives a new worker index.

- **Worker count**

The number of worker processes Playwright Test may run concurrently.

- **workers**

The Playwright Test configuration or CLI setting that controls worker count.

- **Fully parallel mode**

Running all tests, including tests within the same file where applicable, in parallel through fullyParallel configuration.

- **Serial execution**

Running tests one after another instead of concurrently.

- **Parallel execution**

Running tests at overlapping times in separate worker processes.

- **Test retry**

Re-running a failed test according to configured retries.

- **Retry count**

The configured number of additional attempts Playwright may make after the initial test failure.

- **First retry**

The first additional attempt after an initial failure, represented by retry index 1.

- **Last retry**

The final retry attempt allowed by the configured retry count.

- **Retry classification**

The way Playwright classifies outcomes across attempts, for example passed, flaky, or failed after considering retries.

- **Flaky result**

A Playwright Test outcome where a test fails initially but passes on a retry.

- **Passed result**

A result in which the test execution satisfies its expected status and completes successfully.

- **Failed result**

A result in which the test ends with a failure that does not match its expected status.

- **Skipped result**

A result in which the test was not executed because it was skipped by configuration, annotation, dependency behavior, or runtime conditions.

- **Interrupted result**

A result in which test execution was stopped before normal completion, for example because the run was interrupted.

- **Sharding**

Dividing a test suite into independent parts so different machines or CI jobs can run those parts concurrently.

- **Shard**

One partition of a sharded Playwright Test suite.

- **Shard index**

The one-based number identifying which shard a particular run executes, such as the 2 in 2/3.

- **Shard total**

The total number of shards into which the suite is divided, such as the 3 in 2/3.

- **Test distribution**

The assignment of discovered tests or test groups across workers or shards for execution.

- **maxFailures**

A Playwright configuration or CLI limit that stops scheduling further tests after the specified number of unexpected failures is reached.

- **Fail fast**

A strategy of ending or limiting a run after enough failures occur. Playwright supports this through maxFailures rather than a separate fail-fast API.

- **Global timeout**

The maximum time allowed for the entire Playwright Test run before it is terminated.

- **Timeout**

A maximum allowed duration for an operation or test phase before failure.

- **Default timeout**

The default timeout inherited by many Playwright operations unless a more specific timeout is supplied.

- **Navigation timeout setting**

The configured default limit for navigation operations, set through navigation timeout APIs or relevant configuration.

- **Assertion timeout setting**

The timeout used by auto-retrying expect assertions, configured through Playwright Test expect settings or per assertion.

- **Fixture timeout setting**

A separate timeout that can be assigned to a fixture, especially useful for expensive worker-scoped setup.

- **Slow test**

A test marked with `test.slow()`, which increases its allowed test timeout using Playwright Test's slow-test behavior.

- **Timeout multiplier**

The factor by which a timeout is increased for a slow test; Playwright's `test.slow()` applies its documented multiplier rather than an arbitrary per-call concept.

CHAPTER

20

CI/CD, Docker & Execution Environments

Scope

Playwright + delivery tooling

Terms in this chapter

30

Primary references

<https://playwright.dev/docs/ci>

<https://playwright.dev/docs/docker>

<https://playwright.dev/docs/api/class-electron>

- **Continuous integration**

Automatically building and testing changes in a shared system whenever code is integrated.

- **Continuous delivery**

Automatically preparing software changes so they remain deployable, and in some workflows deploying them after required checks pass.

- **CI pipeline**

The automated sequence of CI tasks that installs dependencies, runs tests, publishes reports, and performs other validation steps.

- **CI job**

One independently scheduled unit of work in a CI system, such as a job that runs a Playwright shard.

- **CI runner**

The machine, virtual machine, or container that executes a CI job.

- **Electron** EXPERIMENTAL

Experimental Playwright support for launching and automating Electron applications through the `_electron` namespace.

- **Pipeline stage**

A logical section of a pipeline, such as build, test, package, or deploy.

- **Test stage**

The stage in which automated tests such as Playwright tests are executed.

- **Environment**

The hardware, operating system, runtime, browser, configuration, and external services surrounding a test run.

- **Test environment**

An environment prepared for executing tests, including application endpoints, data, credentials, dependencies, and browser/runtime setup.

- **Environment parity**

The degree to which two execution environments match in factors that can affect results, especially important for visual comparisons.

- **ElectronApplication** EXPERIMENTAL

An experimental Playwright representation of a launched Electron application, providing access to its main process, browser context, and application windows.

- **Docker**

A container platform commonly used to run Playwright in consistent Linux environments.

- **Docker image**

A packaged container filesystem and metadata used to create Docker containers. Playwright publishes images containing browsers and system dependencies.

- **Docker container**

A running instance of a Docker image.

- **Headless execution**

Running browsers without a visible UI, commonly used in automated CI execution.

- **Linux runner**

A CI worker using Linux. Playwright CI documentation supports browser dependency installation or container-based execution on Linux.

- **GitHub Actions**

A CI service from GitHub. Playwright provides example GitHub Actions workflows for installing browsers and running tests.

- **Jenkins**

A CI automation server that can execute Playwright commands as pipeline steps when Node.js, browsers, and dependencies are available.

- **GitLab CI**

A CI/CD service in GitLab that can execute Playwright tests through configured runners and pipeline jobs.

- **Azure Pipelines**

Microsoft's CI/CD service, for which Playwright documentation provides pipeline examples.

- **CI artifact**

A file retained by a CI system after a job, such as a Playwright HTML report, blob report, trace, screenshot, or test result file.

- **CI cache**

Saved reusable files intended to speed later CI runs, for example dependencies when caching is safe and appropriate.

- **Dependency cache**

A cache of installed project dependencies or package-manager data. Playwright documentation generally recommends installing matching browser binaries explicitly rather than blindly caching browser installs.

- **Package cache**

A cache of package-manager downloads or resolved packages used to reduce installation time.

- **Matrix build**

A CI strategy that runs combinations of parameters, such as browser, operating system, or shard, as separate jobs.

- **Environment secret**

A protected credential or token stored by the CI platform and injected into jobs without committing it to source control.

- **Secret variable**

An environment variable whose value is sourced from a protected CI secret store.

- **Exit code**

A numeric process result used by CI to determine whether a command succeeded or failed.

- **CI test report**

A report generated from CI test execution and published or retained for reviewing outcomes, failures, and evidence.

CHAPTER

21

Accessibility Testing

Scope

Playwright + accessibility standards

Terms in this chapter

25

Primary references

<https://playwright.dev/docs/accessibility-testing>

<https://playwright.dev/docs/aria-snapshots>

<https://www.w3.org/WAI/ARIA/apg/practices/names-and-descriptions/>

- **Accessibility**

Testing whether people with disabilities can perceive, understand, navigate, and operate an application using appropriate browsers and assistive technologies.

- **Accessible role**

The semantic role exposed for an element in the accessibility tree, such as button, heading, checkbox, or navigation.

- **Accessible name computation**

The browser-defined process for calculating the name exposed to assistive technologies from sources such as labels, element text, alt text, and ARIA.

- **Accessible description**

Additional descriptive text exposed to assistive technologies beyond an element's accessible name.

- **ARIA**

Accessible Rich Internet Applications, a WAI specification for exposing roles, states, properties, and relationships when native HTML semantics are insufficient.

- **ARIA role**

A role defined by ARIA or native HTML semantics that describes an element's purpose to assistive technologies.

- **ARIA attribute**

An aria-* attribute used to expose accessibility state, properties, labels, or relationships to assistive technologies.

- **ARIA snapshot**

A YAML representation of accessible page structure used by Playwright for accessibility-tree snapshot testing.

- **ariaSnapshot**

page.ariaSnapshot() or locator.ariaSnapshot(), which returns a YAML representation of accessible elements within the selected scope.

- **Accessibility tree**

The browser-generated representation of page content used by accessibility APIs and assistive technologies.

- **Semantic HTML**

Using native HTML elements according to their intended meaning so browsers can expose correct semantics and behavior without unnecessary ARIA.

- **Keyboard accessibility**

The ability to operate interactive functionality with a keyboard, including logical focus movement and activation.

- **Focus order**

The sequence in which focus moves through focusable elements when navigating with the keyboard.

- **Focus indicator**

A visible indication showing which element currently has keyboard focus.

- **Form label**

The accessible labeling relationship that gives a form control a meaningful name, commonly through a label element or supported ARIA labeling.

- **Alternative text**

Text that communicates the purpose or content of non-text content such as an image when the visual content cannot be used.

- **Landmark**

A semantic region such as main, navigation, banner, or contentinfo that helps users understand and move through page structure.

- **Heading hierarchy**

The logical organization of heading levels so page sections and subsections form a meaningful structure.

- **Name-role-value**

The accessibility principle that interactive components expose programmatically determinable names and roles, with states and values available to assistive technologies.

- **Screen reader**

Software that presents digital content through synthesized speech, braille, or other non-visual output using platform accessibility information.

- **WCAG**

The Web Content Accessibility Guidelines published by W3C WAI, defining testable success criteria for accessible web content.

- **axe-core**

An open-source accessibility testing engine. Playwright documentation demonstrates integration through the separate @axe-core/playwright package.

- **Accessibility assertion**

A test assertion focused on accessibility-related state, such as accessible name, role, description, error message, focus, or ARIA snapshot structure.

- **Accessibility audit**

A broader review of accessibility that can combine automated scanning, manual checks, keyboard testing, screen-reader testing, and inclusive user evaluation.

- **Accessible error message**

The programmatically exposed message describing an error associated with a control; Playwright can assert it with toHaveAccessibleErrorMessage().

CHAPTER

22

Visual Testing

Scope

Playwright Test

Terms in this chapter

20

Primary references

<https://playwright.dev/docs/test-snapshots>

<https://playwright.dev/docs/api/class-snapshotassertions>

- **Visual testing**

Testing rendered appearance by comparing visual output with an expected reference.

- **Visual regression testing**

Detecting unintended UI appearance changes by comparing current screenshots with approved reference images.

- **Screenshot comparison**

Comparing a newly captured image against a stored reference and evaluating whether differences exceed configured limits.

- **Baseline screenshot**

The approved reference image against which future screenshots are compared.

- **Actual screenshot**

The screenshot produced by the current test execution.

- **Diff image**

An image highlighting pixels that differ between the actual and reference screenshots.

- **Pixel difference**

The measured difference between corresponding pixels in the actual and expected images, evaluated against the configured comparison threshold.

- **Snapshot**

A stored representation of expected output. In Playwright visual testing, a screenshot snapshot is used as the comparison reference.

- **Golden image**

Another name commonly used for an approved baseline image in visual regression testing.

- **Image threshold**

The allowed per-pixel color-distance tolerance used during image comparison.

- **Max diff pixels**

The maximum number of differing pixels permitted before a screenshot assertion fails.

- **Max diff pixel ratio**

The maximum allowed fraction of differing pixels relative to the image size.

- **Screenshot stability**

The degree to which repeated screenshot captures produce the same output under equivalent conditions.

- **Font rendering**

Variation in glyph appearance caused by operating system, browser, font files, rasterization, scaling, or rendering environment differences.

- **Animation masking**

Stabilizing screenshots by disabling animations or covering animated areas so transient motion does not create irrelevant differences.

- **Dynamic content masking**

Using screenshot masks to cover changing page regions so expected dynamic content does not dominate the visual comparison.

- **Clip region**

A rectangular screenshot area defined by coordinates and dimensions rather than capturing the full page or a locator.

- **Scale option**

A screenshot option controlling whether output uses CSS pixel scale or device pixel scale where supported by the API.

- **Visual noise**

Irrelevant visual variation, such as animations, timestamps, ads, font differences, or environment-specific rendering, that creates unstable comparisons.

- **Platform rendering difference**

Differences in rendering caused by operating system, browser engine, browser build, hardware, settings, device scale, fonts, or execution mode.

CHAPTER

23

JavaScript, TypeScript & Async Concepts

Scope

Supporting JavaScript/TypeScript

Terms in this chapter

50

Primary references

<https://developer.mozilla.org/docs/Web/JavaScript>

<https://www.typescriptlang.org/docs/>

- **JavaScript** **SUPPORTING**

The programming language executed by browsers and Node.js and commonly used to write Playwright tests.

- **TypeScript** **SUPPORTING**

A statically typed language that builds on JavaScript and is commonly used with Playwright for editor support and compile-time type checking.

- **Variable** **SUPPORTING**

A named binding that stores or refers to a value.

- **const** **SUPPORTING**

A JavaScript declaration that creates a block-scoped binding that cannot be reassigned after initialization.

- **let** **SUPPORTING**

A JavaScript declaration that creates a block-scoped binding whose value can be reassigned.

- **Function** **SUPPORTING**

A reusable block of code that can receive arguments and return a value.

- **Arrow function** **SUPPORTING**

A concise JavaScript function expression written with => syntax.

- **Callback** **SUPPORTING**

A function passed to another function to be called later or when a specific operation occurs.

- **Promise** **SUPPORTING**

A JavaScript object representing the eventual completion or failure of an asynchronous operation.

- **Async function** **SUPPORTING**

A function declared with async that always returns a Promise and allows await inside its body.

- **async** **SUPPORTING**

The JavaScript keyword used to declare an asynchronous function.

- **await** **SUPPORTING**

The JavaScript keyword that pauses an async function until a Promise settles, then returns its fulfilled value or throws its rejection.

- **Event loop** **SUPPORTING**

The runtime mechanism that coordinates the call stack and queued asynchronous work so JavaScript callbacks can execute when the stack becomes available.

- **Call stack** **SUPPORTING**

The runtime stack containing currently executing JavaScript function calls.

- **Microtask** **SUPPORTING**

A queued job processed after the current JavaScript call stack and before the next ordinary task, including Promise reactions.

- **Macrotask** **SUPPORTING**

A commonly used informal term for a normal event-loop task such as timers or many browser events; standards more often call these tasks rather than macrotasks.

- **Asynchronous operation** **SUPPORTING**

Work that completes later, allowing other JavaScript execution to continue in the meantime.

- **Synchronous operation** **SUPPORTING**

Work that completes before control moves to the next statement in the same execution flow.

- **Promise resolution** **SUPPORTING**

The transition of a Promise to a fulfilled state, either directly or after adopting another value or Promise.

- **Promise rejection** **SUPPORTING**

The transition of a Promise to a rejected state with a reason or error.

- **Unhandled rejection** **SUPPORTING**

A Promise rejection that is not handled by an appropriate rejection handler within the runtime's handling window.

- **try** **SUPPORTING**

The JavaScript keyword that begins a block whose thrown errors can be handled by catch and whose cleanup can use finally.

- **catch** **SUPPORTING**

The JavaScript clause that receives an error thrown from an associated try block or awaited rejected Promise.

- **finally** **SUPPORTING**

The JavaScript clause that runs after try/catch completion whether or not an error occurred.

- **throw** **SUPPORTING**

The JavaScript keyword used to raise an exception value.

- **Error** **SUPPORTING**

An object representing a runtime problem, normally containing a message and often a stack trace.

- **Error handling** **SUPPORTING**

The practice of detecting, propagating, logging, recovering from, or intentionally failing on errors in a controlled way.

- **Optional chaining** **SUPPORTING**

The ?. JavaScript operator that safely stops a property access or call chain and returns undefined when the preceding value is null or undefined.

- **Nullish coalescing** **SUPPORTING**

The ?? JavaScript operator that uses its right-hand value only when the left-hand value is null or undefined.

- **Destructuring** **SUPPORTING**

JavaScript syntax that extracts properties or array items into variables.

- **Object** **SUPPORTING**

A JavaScript value composed of named properties and associated values or methods.

- **Array** **SUPPORTING**

An ordered JavaScript collection whose items are addressed by numeric indexes.

- **String** **SUPPORTING**

A JavaScript primitive type representing text.

- **Number** **SUPPORTING**

A JavaScript primitive type representing numeric values, including integers, floating-point values, NaN, and Infinity.

- **Boolean** **SUPPORTING**

A JavaScript primitive type with true and false values.

- **Regular expression** **SUPPORTING**

A pattern object used for matching text, commonly written as /pattern/ or created with RegExp.

- **Template literal** **SUPPORTING**

A JavaScript string literal written with backticks that supports interpolation and multiline text.

- **Import** **SUPPORTING**

JavaScript/TypeScript syntax used to bring exported bindings from another module into the current module.

- **Export** **SUPPORTING**

JavaScript/TypeScript syntax used to make bindings available for import by other modules.

- **Module** **SUPPORTING**

A file or code unit with its own scope that exports or imports values or types using a module system.

- **Type** **SUPPORTING**

A TypeScript description of the set of values and operations allowed for a program value.

- **Interface** **SUPPORTING**

A TypeScript declaration describing the shape of an object-like type and supporting extension and declaration merging.

- **Type inference** **SUPPORTING**

TypeScript's ability to determine a value's type from its initializer, usage, control flow, or surrounding context without an explicit annotation.

- **Union type** **SUPPORTING**

A TypeScript type that allows a value to be one of several specified types, written with |.

- **Generic** **SUPPORTING**

A TypeScript construct that uses type parameters so reusable code can preserve relationships between input and output types.

- **Optional property** **SUPPORTING**

A TypeScript object property marked with ? so values of that type may omit the property.

- **Readonly property** **SUPPORTING**

A TypeScript property marked readonly so assignment through that typed reference is restricted after initialization.

- **Enum** **SUPPORTING**

A TypeScript feature defining a named set of related constants. Modern TypeScript code can also use union literals or const objects depending on needs.

- **Source map** **SUPPORTING**

A mapping generated by tooling that connects transformed JavaScript back to original TypeScript or source files for debugging.

- **Top-level await** **SUPPORTING**

Using await at the top level of an ECMAScript module rather than inside an async function, where the runtime and module configuration support it.

CHAPTER

24

DOM, HTML, Browser & Web Platform Concepts

Scope

Supporting web platform

Terms in this chapter

55

Primary references

https://developer.mozilla.org/docs/Web/API/Document_Object_Model

<https://developer.mozilla.org/docs/Web/HTML>

<https://developer.mozilla.org/docs/Web/HTTP>

- **DOM** **SUPPORTING**

The Document Object Model, a tree of objects representing the structure and content of an HTML or XML document.

- **Document** **SUPPORTING**

The top-level DOM object representing the loaded web document.

- **Element** **SUPPORTING**

A DOM node representing an HTML or XML element and its attributes, children, and element-specific behavior.

- **Node** **SUPPORTING**

The base DOM interface for items in the document tree, including elements, text nodes, comments, and the document itself.

- **HTML** **SUPPORTING**

The markup language used to structure content and controls in web pages.

- **HTML attribute** **SUPPORTING**

A name-value setting written in HTML markup, such as id, href, disabled, or aria-label.

- **HTML property** **SUPPORTING**

A JavaScript property on a DOM object. A property may reflect an HTML attribute but is runtime state and is not always identical to the attribute.

- **Form control** **SUPPORTING**

An interactive HTML control used to collect or change user input, such as input, textarea, button, or select.

- **Button element** **SUPPORTING**

The HTML button element, a native interactive control with built-in button semantics and keyboard behavior.

- **Input element** **SUPPORTING**

The HTML input element used for many form-control types such as text, checkbox, radio, date, or file.

- **Select element** **SUPPORTING**

The HTML select element used to choose one or more options from an option list.

- **Option element** **SUPPORTING**

The HTML option element representing one selectable choice inside a select element.

- **Textarea** **SUPPORTING**

The HTML textarea element used for multi-line text input.

- **Checkbox** **SUPPORTING**

An input control that represents a binary checked/unchecked choice; multiple checkboxes can generally be selected independently.

- **Radio button** **SUPPORTING**

An input control used for one selection within a related group of options.

- **Anchor element** **SUPPORTING**

The HTML anchor element, normally used to create hyperlinks through its href attribute.

- **Image element** **SUPPORTING**

The HTML img element used to embed images and normally requiring appropriate alternative text when the image conveys meaning.

- **Form element** **SUPPORTING**

The HTML form element grouping controls that can be submitted together.

- **Label element** **SUPPORTING**

The HTML label element used to associate visible labeling text with a form control.

- **Heading element** **SUPPORTING**

An h1 through h6 HTML element representing a section heading with a semantic level.

- **Landmark element** **SUPPORTING**

An element whose semantics define a navigational region, such as main, navigation, banner, complementary, or contentinfo.

- **Custom element** **SUPPORTING**

A developer-defined HTML element registered through the Custom Elements API.

- **Web component** **SUPPORTING**

A reusable web-platform component commonly built with Custom Elements, Shadow DOM, templates, and related standards.

- **Shadow tree** **SUPPORTING**

The DOM subtree rooted at a ShadowRoot and encapsulated from the ordinary light DOM.

- **Shadow host** **SUPPORTING**

The DOM element to which a shadow root is attached.

- **Event** **SUPPORTING**

An object representing something that happened in the browser, such as a click, input, keydown, submit, or load.

- **Event target** **SUPPORTING**

The object to which an event was originally dispatched, exposed through event.target.

- **Event bubbling** **SUPPORTING**

The phase in which many DOM events propagate from the target upward through ancestor elements.

- **Event capturing** SUPPORTING

The phase in which an event travels from ancestors toward its target before the target/bubbling phases.

- **Default action** SUPPORTING

The browser behavior that normally occurs for an event when script does not cancel it, such as following a link or submitting a form.

- **Mutation** SUPPORTING

A change to the DOM tree, attributes, or text content.

- **DOM mutation** SUPPORTING

Adding, removing, moving, or modifying DOM nodes, attributes, or text while a page is running.

- **Re-render** SUPPORTING

Updating rendered DOM output after application state changes, common in client-side UI frameworks.

- **Single-page application** SUPPORTING

A web application that updates views through client-side JavaScript without requiring a full document load for every screen change.

- **Client-side routing** SUPPORTING

Changing application views and URL state in the browser through client-side code, commonly using the History API.

- **Server-side rendering** SUPPORTING

Producing HTML on the server before sending it to the browser.

- **Hydration** SUPPORTING

Attaching client-side application logic and event handlers to server-rendered HTML so the page becomes interactive.

- **URL origin** SUPPORTING

The URL security boundary defined by scheme, hostname, and port.

- **Same-origin policy** SUPPORTING

The browser security policy that restricts scripts from freely reading resources or documents from a different origin.

- **Cross-origin request** SUPPORTING

An HTTP request from one origin to a different origin, subject to browser security controls such as CORS.

- **HTTP** SUPPORTING

The Hypertext Transfer Protocol used for request-response communication on the web.

- **HTTPS** SUPPORTING

HTTP protected by TLS encryption and server authentication.

- **HTTP method** SUPPORTING

The request verb describing the intended HTTP operation, such as GET, POST, PUT, PATCH, DELETE, or HEAD.

- **HTTP status code** SUPPORTING

The three-digit code in an HTTP response indicating the result category and specific outcome of the request.

- **Request header** SUPPORTING

A metadata field sent in an HTTP request, such as Accept, Authorization, Cookie, or Content-Type.

- **Response header** SUPPORTING

A metadata field returned in an HTTP response, such as Content-Type, Set-Cookie, or Cache-Control.

- **Content-Type header** SUPPORTING

The HTTP header that identifies the media type and optional parameters of a request or response body.

- **Authorization header** SUPPORTING

The HTTP request header commonly used to carry credentials such as Bearer tokens or Basic authentication data.

- **Cache-Control** SUPPORTING

The HTTP header that controls caching behavior for requests and responses.

- **Browser cache** SUPPORTING

Browser storage of previously fetched resources so later requests can sometimes reuse them instead of downloading identical content again.

- **Web storage** SUPPORTING

The browser Web Storage APIs, localStorage and sessionStorage, for origin-scoped key-value data.

- **Cookie storage** SUPPORTING

Browser-managed cookie data associated with domains and paths and automatically included with matching HTTP requests subject to cookie rules.

- **IndexedDB storage** SUPPORTING

The browser's IndexedDB data store for larger structured, origin-scoped client-side data.

- **Viewport coordinate** SUPPORTING

An x/y position measured relative to the current viewport, commonly used for mouse actions or layout calculations.

- **Bounding box** SUPPORTING

The rectangle enclosing an element or rendered box, typically represented by x, y, width, and height values.

CHAPTER

25

Framework Design, Patterns, Reliability & Flakiness

Scope

Playwright best practices + test engineering

Terms in this chapter

55

Primary references

<https://playwright.dev/docs/best-practices>

<https://playwright.dev/docs/pom>

<https://playwright.dev/docs/test-fixtures>

- **Framework architecture**

The high-level organization of automation code, configuration, data, fixtures, utilities, execution, and reporting responsibilities.

- **Page Object Model**

A test-automation design pattern where page-related locators and operations are represented through higher-level page classes or objects.

- **Page object**

An object representing a page or meaningful page area and exposing reusable operations while hiding lower-level locators and interaction details.

- **Component object**

An object representing a reusable UI component, such as a header, modal, table, or product card, across one or more pages.

- **Service object**

An object that wraps calls to an external service or API so tests interact through a clear reusable interface.

- **Test data**

The input values, records, accounts, files, and application state required to execute a test.

- **Test data factory**

A function or object that creates valid test data, often allowing selected fields to vary per scenario.

- **Test data builder**

A construction pattern that incrementally assembles complex test data with readable defaults and overrides.

- **Test utility**

Reusable supporting code that helps tests perform non-domain-specific work such as parsing, file handling, dates, or common transformations.

- **Helper function**

A reusable function that performs a small supporting operation for tests or framework code.

- **Reusable fixture**

A custom fixture designed for repeated use across many tests or files while preserving Playwright's fixture lifecycle.

- **Abstraction**

Hiding unnecessary implementation detail behind a simpler interface so tests express intent rather than low-level mechanics.

- **Encapsulation**

Keeping internal data and implementation details inside a component while exposing only the operations consumers need.

- **Separation of concerns**

Dividing code so distinct responsibilities such as page interaction, data creation, assertions, and API access do not become tightly mixed.

- **Single responsibility**

Designing a module or class around one clear reason to change rather than unrelated responsibilities.

- **DRY principle**

"Don't Repeat Yourself": reducing unnecessary duplicated knowledge or logic while avoiding abstractions that make tests harder to understand.

- **Over-abstraction**

Creating abstraction layers that add indirection or complexity without enough reuse, clarity, or stability benefit.

- **Test readability**

The ease with which a reader can understand a test's purpose, setup, actions, and expected outcome.

- **Test maintainability**

The ease of changing test code safely as the product, tooling, and test requirements evolve.

- **Test scalability**

The ability of a test suite and its supporting framework to remain practical as test volume, contributors, browsers, environments, and execution demands grow.

- **Test stability**

The tendency of a test to produce dependable results without intermittent false failures.

- **Deterministic test**

A test whose result depends only on defined inputs and system behavior, not uncontrolled timing, ordering, or random external factors.

- **Independent test**

A test that can run by itself without depending on another test having executed first.

- **Isolated test**

A test whose browser/session state and test data do not unintentionally leak into or depend on other tests.

- **Idempotent test**

A test or setup operation that can be repeated without producing unintended additional side effects after the desired state has already been reached.

- **Flaky test detection**

Identifying tests that produce inconsistent results across repeated equivalent runs.

- **Flake triage**

Investigating a flaky test to determine whether the instability comes from the product, test code, environment, data, timing, or external dependencies.

- **Retry masking**

Using retries in a way that hides persistent test defects or product instability instead of diagnosing the initial failure.

- **Root cause analysis**

The systematic process of finding the underlying cause that produced a failure rather than treating only the visible symptom.

- **Failure diagnosis**

Using errors, traces, screenshots, videos, network data, logs, and application evidence to determine why a test failed.

- **Failure reproducibility**

The ability to recreate the same failure under known conditions so its cause can be investigated and verified.

- **Test observability**

The amount and quality of evidence a test system exposes about execution behavior, timing, environment, requests, errors, and artifacts.

- **Logging strategy**

A deliberate approach to what gets logged, at what detail, and where logs are retained so failures can be diagnosed without excessive noise.

- **Artifact strategy**

A deliberate approach to traces, screenshots, videos, reports, and attachments so useful failure evidence is retained at reasonable cost.

- **Locator strategy**

A consistent rule for choosing resilient locators, generally prioritizing user-facing attributes and explicit test contracts over brittle DOM structure.

- **Assertion strategy**

A consistent rule for what to verify and which assertion style to use, favoring Playwright auto-retrying web assertions for asynchronous UI state.

- **Wait strategy**

A consistent synchronization approach that relies on Playwright auto-waiting, web assertions, and explicit event/state waits instead of arbitrary sleeps.

- **Test data strategy**

A defined approach for creating, isolating, cleaning, and reusing test data across tests and environments.

- **Environment strategy**

A defined approach for selecting, configuring, resetting, and observing the environments where tests run.

- **Authentication strategy**

A defined approach for creating and reusing signed-in state while protecting credentials and avoiding account conflicts across parallel tests.

- **Parallelization strategy**

A plan for using Playwright workers and fully-parallel settings while keeping tests independent and protecting shared resources.

- **Sharding strategy**

A plan for dividing the test suite across machines or CI jobs with Playwright sharding.

- **Test tagging strategy**

A consistent use of tags or metadata so tests can be grouped, filtered, reported, or targeted by purpose.

- **Test selection**

Choosing which tests to execute for a particular change, risk, environment, branch, or pipeline stage.

- **Smoke test**

A small, fast set of tests that checks whether the most essential application flows are working well enough for deeper testing to proceed.

- **Regression test**

A set of tests intended to detect unintended breakage in behavior that previously worked.

- **Sanity test**

A focused confidence check commonly run after a limited change or fix; the exact meaning varies by organization and is not a Playwright-specific concept.

- **Critical-path test**

A test covering a business-essential flow whose failure would have high impact.

- **Happy-path test**

A test covering the expected successful path using valid inputs and normal user behavior.

- **Negative test**

A test covering invalid input, rejected actions, errors, or other conditions where the system should not follow the successful path.

- **Boundary test**

A test focused on values at, just below, and just above meaningful limits or partitions.

- **Data-driven test**

A test that runs the same behavior against multiple data sets.

- **Parameterized test**

A test generated or repeated with different parameters so one test definition can cover multiple input combinations.

- **Risk-based testing**

Prioritizing testing based on the likelihood and impact of failures so effort is concentrated on the most important product risks.

- **Test pyramid**

A general testing strategy model that uses many lower-level tests and progressively fewer higher-level integration or end-to-end tests. It is not a Playwright-specific requirement.

END OF GLOSSARY

The Playwright Glossary Every Automation Tester Needs

1,000 terms across 25 categories, compiled for automation testers working with Playwright.

Author

Manjunath N P

Website

manjunathnp.in

GitHub

github.com/manjunathnp

LinkedIn

linkedin.com/in/manjunathnp