Auto-waiting

Auto-Waiting Mechanism

Playwright has a powerful feature called **Auto-Waiting**, which simplifies test automation by automatically waiting for the right conditions before performing actions.

What is Auto-Waiting?

Auto-waiting means that **Playwright automatically waits** for the necessary conditions (*Actionable checks*) before executing actions like **click(), fill(), type() etc.**

For example:

```
await page.locator('button#submit').click();
```

Before clicking, Playwright will **wait until**:

- The button is **attached** to the DOM
- It is **visible**
- It is **enabled**
- It is **not covered** by another element

This removes the need for adding manual waits like **sleep()** or **waitForTimeout()** in most cases.

Forcing actions

Some actions like **locator.click()** support **force** option that disables non-essential actionability checks, For example:

```
await page.locator('button#submit').click({force:true});
```

Above **click({force:true})** method will not check that the target element actually receives click events.

Why is Auto-Waiting Useful?

- Reduces flaky tests caused by timing issues
- Makes your tests **more stable and reliable**
- Improves **readability and maintainability**

Timeouts

Timeouts are used in Playwright to define how long the framework should wait before failing a test or assertion. Playwright provides flexible options to manage timeouts globally or locally.

Test Timeout (Timeout for Each Test)

- **Default:** 30,000 ms (30 seconds)
- This timeout defines **how long a single test is allowed to run**.

Set Test Timeout in the Config File:

To change the timeout globally for all tests:

// playwright.config.ts

```
export default defineConfig({
     timeout: 60000
})
```

Override/Set Timeout for a Specific Test

To change the timeout for just one test:

```
test('my long-running test', async ({ page }) => {
  test.setTimeout(60000); // 60 seconds
  // your test code here
});
```

Make Test Slower Temporarily

To automatically **triple the default timeout**:

```
test('slow test', async ({ page }) => {
  test.slow(); // Now timeout = 90,000 ms (3x default)
});
```

Expect Timeout (Timeout for Assertions)

- **Default:** 5,000 ms (5 seconds)
- This timeout defines **how long Playwright waits for a condition/assertion** (like visibility, text match, etc.)

Set Expect Timeout in the Config File

To apply a longer wait for all expect conditions:

// playwright.config.ts

```
export default defineConfig({
    expect: { timeout: 10000 }
})
```

Override/set Timeout for a Specific Expect

You can override the timeout for a particular assertion like this:

await expect(locator).toBeVisible(**{ timeout: 10_000 }**);

Summary

Timeout Type	Default	Set Globally	Override in Test
Test Timeout	30,000ms	timeout: 60_000 in config	test.setTimeout(120_000)
Expect Timeout	5,000ms	expect: { timeout: 10_000 }	expect(...).toBeVisible({ timeout })
Triple Timeout	-	-	test.slow()

Playwright Assertions

Playwright provides **built-in assertions** to validate your test conditions, such as checking text, visibility, URLs, etc. These assertions fall into two categories: **Auto-Retrying** and **Non-Retrying**.

Auto-Retrying Assertions:

These assertions are used **on locators or pages** and are **asynchronous**, which means they **return a Promise**, so you must use await.

Key Features:

- Automatically **retry** until the condition passes or times out.
- **Auto-wait** for elements to appear, become visible, etc.
- **Timeout is configurable.**

Syntax:

```
await expect(locator).toHaveText('Some Text');

await expect(locator).toBeVisible();

await expect(page).toHaveURL('https://example.com');
```

Examples:

```
await expect(page).toHaveURL("https://demowebshop.tricentis.com/");

await expect(page.locator('text=Welcome to our store')).toBeVisible();

await expect(page.locator("div[class='product-grid home-page-product-grid'] strong")).toHaveText('Featured products');
```

Non-Retrying Assertions

These assertions are used **on values (like strings, booleans, numbers)** and are **synchronous**, meaning they **do not return a Promise**, so you **don't need await**.

Limitations:

- No retrying — assertion runs immediately.
- No auto-waiting — make sure the value is ready.
- Timeout is **not configurable**.

Syntax:

```
expect(value).toBeTruthy();

expect(text).toContain('Welcome');
```

Examples:

```
const title = await page.title();

expect(title.includes('Demo Web Shop')).toBeTruthy();


const welcometext = await page.locator('text=Welcome to our store').textContent();

expect(welcometext).toContain('Welcome');
```

Negating Matchers

You can **invert** an assertion using .not.

Example:

```
await expect(page.locator('h1')).not.toHaveText('Error');
```

Auto-Retrying Assertions Vs Non-Retrying Assertions

Feature / Criteria	Auto-Retrying Assertions	Non-Retrying Assertions
Used On	page, locator	Primitive values like string, boolean, number, etc.
Returns	Promise (asynchronous)	Synchronous (does not return Promise)
Need await before expect()?	✅ Yes	❌ No
Auto-waiting	✅ Yes (waits for the condition to be met)	❌ No (executes immediately)
Retry Mechanism	✅ Yes (keeps retrying until timeout or success)	❌ No (asserts once)
Timeout Configurable?	✅ Yes (can be set using timeout option)	❌ No
Failure Handling	Retries before failing	Fails immediately if the condition is false

Typical Use Case	Asserting UI elements, page properties	Validating extracted values from elements or functions

Hard vs Soft Assertions

Hard Assertion (Default)

- Fails immediately and **stops the test**.
- Use when a failure should **halt** execution.

Soft Assertion

- Fails, but **allows the test to continue**.
- Use to **record multiple issues** in one run.

Example:

```
test('soft assertion example', async ({ page }) => {
  await expect.soft(page.locator('h1')).toHaveText('Welcome');
  // The test continues even if the above fails
});
```

Note: Use **soft assertions** when you want to verify multiple conditions independently.

Hard Assertions Vs Soft Assertions

Feature / Criteria	Hard Assertion	Soft Assertion
Default Behavior	✅ Yes, it is the default assertion type	❌ No, must be explicitly written using expect.soft()
Test Execution on Failure	❌ Stops test execution immediately after failure	✅ Continues executing remaining steps even if it fails
Use Case	Critical checks where failure should block further steps	Non-critical checks or multiple validations in a single test
Syntax	await expect(locator).toHaveText('Welcome');	await expect.soft(locator).toHaveText('Welcome');
Result Reporting	Reports failure and halts the test	Reports failure but aggregates it at the end of the test