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

Writing tests

Introduction

Playwright tests are simple, they

- **perform actions**, and
- **assert the state** against expectations.

There is no need to wait for anything prior to performing an action: Playwright automatically waits for the wide range of actionability checks to pass prior to performing each action.

There is also no need to deal with the race conditions when performing the checks - Playwright assertions are designed in a way that they describe the expectations that need to be eventually met.

That's it! These design choices allow Playwright users to forget about flaky timeouts and racy checks in their tests altogether.

You will learn

- How to write the first test
- How to perform actions
- How to use assertions
- How tests run in isolation
- How to use test hooks

First test

Take a look at the following example to see how to write a test.

tests/example.spec.ts

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

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

```
test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');

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

  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation'
})).toBeVisible();
});
```

NOTE

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

Actions

Navigation

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

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

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

Interactions

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

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

// Click it.
await getStarted.click();
```

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

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

Basic actions

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

Action	Description
locator.check()	Check the input checkbox

Action	Description
locator.click()	Click the element
locator.uncheck()	Uncheck the input checkbox
locator.hover()	Hover mouse over the element
locator.fill()	Fill the form field, input text
locator.focus()	Focus the element
locator.press()	Press single key
locator.setInputFiles()	Pick files to upload
locator.selectOption()	Select option in the drop down

Assertions

Playwright includes test assertions in the form of `expect` function. To make an assertion, call `expect(value)` and choose a matcher that reflects the expectation.

There are many generic matchers like `toEqual`, `toContain`, `toBeTruthy` that can be used to assert any conditions.

```
expect(success).toBeTruthy();
```

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

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

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

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

Test Isolation

Playwright Test is based on the concept of test fixtures such as the built in page fixture, which is passed into your test. Pages are isolated between tests due to the Browser Context, which is equivalent to a brand new browser profile, where every test gets a fresh environment, even when multiple tests run in a single Browser.

tests/example.spec.ts

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

test('example test', async ({ page }) => {
  // "page" belongs to an isolated BrowserContext, created for this
specific test.
});

test('another test', async ({ page }) => {
  // "page" in this second test is completely isolated from the first test.
});
```

Using Test Hooks

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

tests/example.spec.ts

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

test.describe('navigation', () => {
  test.beforeEach(async ({ page }) => {
    // Go to the starting url before each test.
    await page.goto('https://playwright.dev/');
  });

  test('main navigation', async ({ page }) => {
    // Assertions use the expect API.
    await expect(page).toHaveURL('https://playwright.dev/');
  });
});
```

What's Next

- Run single test, multiple tests, headed mode
- Generate tests with Codegen
- See a trace of your tests
- Explore UI Mode
- Run tests on CI with GitHub Actions

Previous
Installation

Next
Generating tests

- Introduction
- First test
- Actions

    - Navigation
    - Interactions
    - Basic actions
- Assertions
    - Test Isolation
    - Using Test Hooks
- What's Next

Learn

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

Community

- Stack Overflow
- Discord
- Twitter
- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors