# Fixtures

## Introduction

Playwright Test is based on the concept of test fixtures. Test fixtures are used to establish the environment for each test, giving the test everything it needs and nothing else. Test fixtures are isolated between tests. With fixtures, you can group tests based on their meaning, instead of their common setup.

### Built-in fixtures

You have already used test fixtures in your first test.

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

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

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

The `{ page }` argument tells Playwright Test to setup the `page` fixture and provide it to your test function.

Here is a list of the pre-defined fixtures that you are likely to use most of the time:

| Fixture | Type | Description |
|---|---|---|
| page | Page | Isolated page for this test run. |
| context | BrowserContext | Isolated context for this test run. The `page` fixture belongs to this context as well. Learn how to configure context. |
| browser | Browser | Browsers are shared across tests to optimize resources. Learn how to configure browser. |
| browserName | string | The name of the browser currently running the test. Either `chromium`, `firefox` or `webkit`. |
| request | APIRequestContext | Isolated APIRequestContext instance for this test run. |

### Without fixtures

Here is how typical test environment setup differs between traditional test style and the fixture-based one.

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

Click to expand the code for the `TodoPage`
todo.spec.ts
```
const { test } = require('@playwright/test');
const { TodoPage } = require('./todo-page');

test.describe('todo tests', () => {
  let todoPage;

  test.beforeEach(async ({ page }) => {
    todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo('item1');
    await todoPage.addToDo('item2');
  });

  test.afterEach(async () => {
    await todoPage.removeAll();
  });

  test('should add an item', async () => {
    await todoPage.addToDo('my item');
    // ...
  });

  test('should remove an item', async () => {
    await todoPage.remove('item1');
    // ...
  });
});
```

## With fixtures

Fixtures have a number of advantages over before/after hooks:

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

Click to expand the code for the `TodoPage`
example.spec.ts
```
import { test as base } from '@playwright/test';
import { TodoPage } from './todo-page';

// Extend basic test by providing a "todoPage" fixture.
```

```
const test = base.extend<{ todoPage: TodoPage }>({
  todoPage: async ({ page }, use) => {
    const todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo('item1');
    await todoPage.addToDo('item2');
    await use(todoPage);
    await todoPage.removeAll();
  },
});

test('should add an item', async ({ todoPage }) => {
  await todoPage.addToDo('my item');
  // ...
});

test('should remove an item', async ({ todoPage }) => {
  await todoPage.remove('item1');
  // ...
});
```

# Creating a fixture

To create your own fixture, use test.extend() to create a new `test` object that will include it.

Below we create two fixtures `todoPage` and `settingsPage` that follow the Page Object Model pattern.

Click to expand the code for the `TodoPage` and `SettingsPage`
my-test.ts

```
import { test as base } from '@playwright/test';
import { TodoPage } from './todo-page';
import { SettingsPage } from './settings-page';

// Declare the types of your fixtures.
type MyFixtures = {
  todoPage: TodoPage;
  settingsPage: SettingsPage;
};

// Extend base test by providing "todoPage" and "settingsPage".
// This new "test" can be used in multiple test files, and each of them
will get the fixtures.
export const test = base.extend<MyFixtures>({
  todoPage: async ({ page }, use) => {
    // Set up the fixture.
    const todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo('item1');
    await todoPage.addToDo('item2');

    // Use the fixture value in the test.
    await use(todoPage);

    // Clean up the fixture.
    await todoPage.removeAll();
  },
```

```
  settingsPage: async ({ page }, use) => {
    await use(new SettingsPage(page));
  },
});
export { expect } from '@playwright/test';
```

NOTE

Custom fixture names should start with a letter or underscore, and can contain only letters, numbers, underscores.

# Using a fixture

Just mention fixture in your test function argument, and test runner will take care of it. Fixtures are also available in hooks and other fixtures. If you use TypeScript, fixtures will have the right type.

Below we use the `todoPage` and `settingsPage` fixtures defined above.

```
import { test, expect } from './my-test';

test.beforeEach(async ({ settingsPage }) => {
  await settingsPage.switchToDarkMode();
});

test('basic test', async ({ todoPage, page }) => {
  await todoPage.addToDo('something nice');
  await expect(page.getByTestId('todo-title')).toContainText(['something
nice']);
});
```

# Overriding fixtures

In addition to creating your own fixtures, you can also override existing fixtures to fit your needs. Consider the following example which overrides the `page` fixture by automatically navigating to some `baseURL`:

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

export const test = base.extend({
  page: async ({ baseURL, page }, use) => {
    await page.goto(baseURL);
    await use(page);
  },
});
```

Notice that in this example, the `page` fixture is able to depend on other built-in fixtures such as testOptions.baseURL. We can now configure `baseURL` in the configuration file, or locally in the test file with test.use().

example.spec.ts

```
test.use({ baseURL: 'https://playwright.dev' });
```

Fixtures can also be overridden where the base fixture is completely replaced with something different. For example, we could override the testOptions.storageState fixture to provide our own data.

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

export const test = base.extend({
  storageState: async ({}, use) => {
    const cookie = await getAuthCookie();
    await use({ cookies: [cookie] });
  },
});
```

# Worker-scoped fixtures

Playwright Test uses worker processes to run test files. Similarly to how test fixtures are set up for individual test runs, worker fixtures are set up for each worker process. That's where you can set up services, run servers, etc. Playwright Test will reuse the worker process for as many test files as it can, provided their worker fixtures match and hence environments are identical.

Below we'll create an `account` fixture that will be shared by all tests in the same worker, and override the `page` fixture to login into this account for each test. To generate unique accounts, we'll use the workerInfo.workerIndex that is available to any test or fixture. Note the tuple-like syntax for the worker fixture - we have to pass `{scope: 'worker'}` so that test runner sets up this fixture once per worker.

my-test.ts

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

type Account = {
  username: string;
  password: string;
};

// Note that we pass worker fixture types as a second template parameter.
export const test = base.extend<{}, { account: Account }>({
  account: [async ({ browser }, use, workerInfo) => {
    // Unique username.
    const username = 'user' + workerInfo.workerIndex;
    const password = 'verysecure';

    // Create the account with Playwright.
    const page = await browser.newPage();
    await page.goto('/signup');
    await page.getByLabel('User Name').fill(username);
    await page.getByLabel('Password').fill(password);
    await page.getByText('Sign up').click();
    // Make sure everything is ok.
    await expect(page.getByTestId('result')).toHaveText('Success');
    // Do not forget to cleanup.
    await page.close();

    // Use the account value.
    await use({ username, password });
  }, { scope: 'worker' }],
```

```
  page: async ({ page, account }, use) => {
    // Sign in with our account.
    const { username, password } = account;
    await page.goto('/signin');
    await page.getByLabel('User Name').fill(username);
    await page.getByLabel('Password').fill(password);
    await page.getByText('Sign in').click();
    await expect(page.getByTestId('userinfo')).toHaveText(username);

    // Use signed-in page in the test.
    await use(page);
  },
});
export { expect } from '@playwright/test';
```

# Automatic fixtures

Automatic fixtures are set up for each test/worker, even when the test does not list them directly. To create an automatic fixture, use the tuple syntax and pass `{ auto: true }`.

Here is an example fixture that automatically attaches debug logs when the test fails, so we can later review the logs in the reporter. Note how it uses TestInfo object that is available in each test/fixture to retrieve metadata about the test being run.

my-test.ts

```
import * as debug from 'debug';
import * as fs from 'fs';
import { test as base } from '@playwright/test';

export const test = base.extend<{ saveLogs: void }>({
  saveLogs: [async ({}, use, testInfo) => {
    // Collecting logs during the test.
    const logs = [];
    debug.log = (...args) => logs.push(args.map(String).join(''));
    debug.enable('myserver');

    await use();

    // After the test we can check whether the test passed or failed.
    if (testInfo.status !== testInfo.expectedStatus) {
      // outputPath() API guarantees a unique file name.
      const logFile = testInfo.outputPath('logs.txt');
      await fs.promises.writeFile(logFile, logs.join('\n'), 'utf8');
      testInfo.attachments.push({ name: 'logs', contentType: 'text/plain',
path: logFile });
    }
  }, { auto: true }],
});
export { expect } from '@playwright/test';
```

# Fixture timeout

By default, fixture shares timeout with the test. However, for slow fixtures, especially worker-scoped ones, it is convenient to have a separate timeout. This way you can keep the overall test timeout small, and give the slow fixture more time.

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

const test = base.extend<{ slowFixture: string }>({
  slowFixture: [async ({}, use) => {
    // ... perform a slow operation ...
    await use('hello');
  }, { timeout: 60000 }]
});

test('example test', async ({ slowFixture }) => {
  // ...
});
```

# Fixtures-options

Playwright Test supports running multiple test projects that can be separately configured. You can use "option" fixtures to make your configuration options declarative and type-checked. Learn more about parametrizing tests.

Below we'll create a `defaultItem` option in addition to the `todoPage` fixture from other examples. This option will be set in configuration file. Note the tuple syntax and `{ option: true }` argument.

Click to expand the code for the `TodoPage`
my-test.ts

```
import { test as base } from '@playwright/test';
import { TodoPage } from './todo-page';

// Declare your options to type-check your configuration.
export type MyOptions = {
  defaultItem: string;
};
type MyFixtures = {
  todoPage: TodoPage;
};

// Specify both option and fixture types.
export const test = base.extend<MyOptions & MyFixtures>({
  // Define an option and provide a default value.
  // We can later override it in the config.
  defaultItem: ['Something nice', { option: true }],

  // Our "todoPage" fixture depends on the option.
  todoPage: async ({ page, defaultItem }, use) => {
    const todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo(defaultItem);
    await use(todoPage);
    await todoPage.removeAll();
  },
});
export { expect } from '@playwright/test';
```

We can now use `todoPage` fixture as usual, and set the `defaultItem` option in the config file.

playwright.config.ts

```ts
import { defineConfig } from '@playwright/test';
import type { MyOptions } from './my-test';

export default defineConfig<MyOptions>({
  projects: [
    {
      name: 'shopping',
      use: { defaultItem: 'Buy milk' },
    },
    {
      name: 'wellbeing',
      use: { defaultItem: 'Exercise!' },
    },
  ]
});
```

**Array as an option value**

If the value of your option is an array, for example `[{ name: 'Alice' }, { name: 'Bob' }]`, you'll need to wrap it into an extra array when providing the value. This is best illustrated with an example.

```ts
type Person = { name: string };
const test = base.extend<{ persons: Person[] }>({
  // Declare the option, default value is an empty array.
  persons: [[], { option: true }],
});

// Option value is an array of persons.
const actualPersons = [{ name: 'Alice' }, { name: 'Bob' }];
test.use({
  // CORRECT: Wrap the value into an array and pass the scope.
  persons: [actualPersons, { scope: 'test' }],
});

test.use({
  // WRONG: passing an array value directly will not work.
  persons: actualPersons,
});
```

# Execution order

Each fixture has a setup and teardown phase separated by the `await use()` call in the fixture. Setup is executed before the fixture is used by the test/hook, and teardown is executed when the fixture will not be used by the test/hook anymore.

Fixtures follow these rules to determine the execution order:

- When fixture A depends on fixture B: B is always set up before A and torn down after A.
- Non-automatic fixtures are executed lazily, only when the test/hook needs them.
- Test-scoped fixtures are torn down after each test, while worker-scoped fixtures are only torn down when the worker process executing tests is shutdown.

Consider the following example:

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

const test = base.extend<{
  testFixture: string,
  autoTestFixture: string,
  unusedFixture: string,
}, {
  workerFixture: string,
  autoWorkerFixture: string,
}>({
  workerFixture: [async ({ browser }) => {
    // workerFixture setup...
    await use('workerFixture');
    // workerFixture teardown...
  }, { scope: 'worker' }],

  autoWorkerFixture: [async ({ browser }) => {
    // autoWorkerFixture setup...
    await use('autoWorkerFixture');
    // autoWorkerFixture teardown...
  }, { scope: 'worker', auto: true }],

  testFixture: [async ({ page, workerFixture }) => {
    // testFixture setup...
    await use('testFixture');
    // testFixture teardown...
  }, { scope: 'test' }],

  autoTestFixture: [async () => {
    // autoTestFixture setup...
    await use('autoTestFixture');
    // autoTestFixture teardown...
  }, { scope: 'test', auto: true }],

  unusedFixture: [async ({ page }) => {
    // unusedFixture setup...
    await use('unusedFixture');
    // unusedFixture teardown...
  }, { scope: 'test' }],
});

test.beforeAll(async () => { /* ... */ });
test.beforeEach(async ({ page }) => { /* ... */ });
test('first test', async ({ page }) => { /* ... */ });
test('second test', async ({ testFixture }) => { /* ... */ });
test.afterEach(async () => { /* ... */ });
test.afterAll(async () => { /* ... */ });
```

Normally, if all tests pass and no errors are thrown, the order of execution is as following.

- worker setup and `beforeAll` section:
  - `browser` setup because it is required by `autoWorkerFixture`.
  - `autoWorkerFixture` setup because automatic worker fixtures are always set up before anything else.
  - `beforeAll` runs.
- `first test` section:

    - `autoTestFixture` setup because automatic test fixtures are always set up before test and `beforeEach` hooks.
    - `page` setup because it is required in `beforeEach` hook.
    - `beforeEach` runs.
    - `first test` runs.
    - `afterEach` runs.
    - `page` teardown because it is a test-scoped fixture and should be torn down after the test finishes.
    - `autoTestFixture` teardown because it is a test-scoped fixture and should be torn down after the test finishes.
- `second test` section:
    - `autoTestFixture` setup because automatic test fixtures are always set up before test and `beforeEach` hooks.
    - `page` setup because it is required in `beforeEach` hook.
    - `beforeEach` runs.
    - `workerFixture` setup because it is required by `testFixture` that is required by the `second test`.
    - `testFixture` setup because it is required by the `second test`.
    - `second test` runs.
    - `afterEach` runs.
    - `testFixture` teardown because it is a test-scoped fixture and should be torn down after the test finishes.
    - `page` teardown because it is a test-scoped fixture and should be torn down after the test finishes.
    - `autoTestFixture` teardown because it is a test-scoped fixture and should be torn down after the test finishes.
- `afterAll` and worker teardown section:
    - `afterAll` runs.
    - `workerFixture` teardown because it is a workers-scoped fixture and should be torn down once at the end.
    - `autoWorkerFixture` teardown because it is a workers-scoped fixture and should be torn down once at the end.
    - `browser` teardown because it is a workers-scoped fixture and should be torn down once at the end.

A few observations:

- `page` and `autoTestFixture` are set up and torn down for each test, as test-scoped fixtures.
- `unusedFixture` is never set up because it is not used by any tests/hooks.
- `testFixture` depends on `workerFixture` and triggers its setup.
- `workerFixture` is lazily set up before the second test, but torn down once during worker shutdown, as a worker-scoped fixture.
- `autoWorkerFixture` is set up for `beforeAll` hook, but `autoTestFixture` is not.

# Combine custom fixtures from multiple modules

You can merge test fixtures from multiple files or modules:

fixtures.ts

```
import { mergeTests } from '@playwright/test';
import { test as dbTest } from 'database-test-utils';
import { test as a11yTest } from 'a11y-test-utils';

export const test = mergeTests(dbTest, a11yTest);
```

test.spec.ts

```
import { test } from './fixtures';

test('passes', async ({ database, page, a11y }) => {
  // use database and a11y fixtures.
});
```

# Box fixtures

Usually, custom fixtures are reported as separate steps in in the UI mode, Trace Viewer and various test reports. They also appear in error messages from the test runner. For frequently-used fixtures, this can mean lots of noise. You can stop the fixtures steps from being shown in the UI by "boxing" it.

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

export const test = base.extend({
  helperFixture: [async ({}, use, testInfo) => {
    // ...
  }, { box: true }],
});
```

This is useful for non-interesting helper fixtures. For example, an automatic fixture that sets up some common data can be safely hidden from a test report.

# Custom fixture title

Instead of the usual fixture name, you can give fixtures a custom title that will be shown in test reports and error messages.

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

export const test = base.extend({
  innerFixture: [async ({}, use, testInfo) => {
    // ...
  }, { title: 'my fixture' }],
});
```

# Adding global beforeEach/afterEach hooks

test.beforeEach() and test.afterEach() hooks run before/after each test declared in the same file and same test.describe() block (if any). If you want to declare hooks that run before/after each test globally, you can declare them as auto fixtures like this:

fixtures.ts

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

```
export const test = base.extend<{ forEachTest: void }>({
  forEachTest: [async ({ page }, use) => {
    // This code runs before every test.
    await page.goto('http://localhost:8000');
    await use();
    // This code runs after every test.
    console.log('Last URL:', page.url());
  }, { auto: true }],  // automatically starts for every test.
});
```

And then import the fixtures in all your tests:

mytest.spec.ts
```
import { test } from './fixtures';
import { expect } from '@playwright/test';

test('basic', async ({ page }) => {
  expect(page).toHaveURL('http://localhost:8000');
  await page.goto('https://playwright.dev');
});
```

# Adding global beforeAll/afterAll hooks

test.beforeAll() and test.afterAll() hooks run before/after all tests declared in the same file and same test.describe() block (if any), once per worker process. If you want to declare hooks that run before/after all tests in every file, you can declare them as auto fixtures with `scope: 'worker'` as follows:

fixtures.ts
```
import { test as base } from '@playwright/test';

export const test = base.extend<{}, { forEachWorker: void }>({
  forEachWorker: [async ({}, use) => {
    // This code runs before all the tests in the worker process.
    console.log(`Starting test worker ${test.info().workerIndex}`);
    await use();
    // This code runs after all the tests in the worker process.
    console.log(`Stopping test worker ${test.info().workerIndex}`);
  }, { scope: 'worker', auto: true }],  // automatically starts for every
worker.
});
```

And then import the fixtures in all your tests:

mytest.spec.ts
```
import { test } from './fixtures';
import { expect } from '@playwright/test';

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

Note that the fixtures will still run once per worker process, but you don't need to redeclare them in every file.

Previous
Emulation

Next
Global setup and teardown