

API тестування з Playwright

API (Application Programming Interface) – це спосіб взаємодії програм між собою. У веб-контексті це набір HTTP-запитів, за допомогою яких клієнт (браузер, мобільний додаток, інша система) отримує або надсилає дані на сервер. Кожного разу, коли ви завантажуєте сторінку, натискаєте кнопку чи відправляєте форму – за лаштунками виконуються десятки API-запитів.

API-тестування – це перевірка роботи цих запитів без графічного інтерфейсу: ми відправляємо запити напряму, перевіряємо статус-код і тіло відповіді. Такий підхід значно швидший і стабільніший за E2E-тести, оскільки не потрібно керувати браузером, чекати на рендеринг сторінки чи обробляти динамічні елементи.

Навіщо Playwright для API?



Один інструмент

E2E та API-тести в одному фреймворку – єдиний стек технологій



Спільна інфраструктура

Fixtures, конфігурація та логіка використовуються повторно



Setup через API

API-запити для підготовки даних у E2E-тестах – швидко та надійно

APIRequestContext: основні ідеї

APIRequestContext – це основний об'єкт Playwright для виконання HTTP-запитів. Його інтерфейс нагадує стандартний `fetch`, але з додатковими можливостями, які спрощують роботу з API в контексті тестування. Цей об'єкт інкапсулює всю логіку відправки запитів, обробки відповідей та управління сесією.

Головна перевага `APIRequestContext` полягає в тому, що він дозволяє налаштовувати спільні параметри для всіх запитів один раз – і далі використовувати їх без повторення. Це значно зменшує дублювання коду та робить тести чистішими і зрозумілішими.



Спільний baseURL

Один базовий URL для всіх запитів контексту – не потрібно повторювати домен у кожному виклику



Зручні методи

Готові методи `get`, `post`, `put`, `patch`, `delete` – інтуїтивний API



Вбудовані заголовки

Заголовки авторизації, Content-Type та інші встановлюються один раз для всього контексту



Інтеграція з авторизацією

Підтримка cookie, storageState та httpCredentials для різних схем автентифікації

Створення APIRequestContext

Контекст створюється через метод `apiRequest.newContext()` (або `playwright.request.newContext()` у fixtures). Ви можете створити його без параметрів для простих запитів або передати об'єкт з опціями для більш тонкого налаштування. Правильне налаштування контексту – ключ до чистих і підтримуваних тестів.

Основні опції

01

`baseURL`

Базовий URL, до якого додаються відносні шляхи запитів

02

`extraHTTPHeaders`

Заголовки для всіх запитів, наприклад `Authorization`

03

`httpCredentials`

Логін та пароль для Basic Auth

04

`timeout`

Таймаут очікування на запит у мілісекундах

05

`storageState`

Cookies та `localStorage` для авторизації

Приклад коду

```
// Простий контекст без налаштувань  
const client = await apiRequest.newContext();
```

```
// Контекст з повними налаштуваннями  
const client = await apiRequest.newContext({  
  baseURL: 'https://api.example.com',  
  extraHTTPHeaders: {  
    'Authorization': 'Bearer TOKEN',  
    'Content-Type': 'application/json',  
  },  
  timeout: 10000,  
});
```

Повний список опцій – у [офіційній документації Playwright](#).

HTTP-методи і параметри

Playwright підтримує всі основні HTTP-методи, кожен з яких має своє призначення у REST API. Розуміння різниці між ними та правильне використання параметрів `params` і `data` – фундамент ефективного API-тестування.

Метод	Коли використовувати	Приклад
GET	Отримання даних (список, один елемент)	Список користувачів, деталі замовлення
POST	Створення нового ресурсу	Нова книга, реєстрація користувача
PUT	Повна заміна ресурсу	Оновлення всього профілю
PATCH	Часткове оновлення ресурсу	Зміна email або імені
DELETE	Видалення ресурсу	Видалення запису, скасування підписки

params

Параметри запиту, які додаються до URL як query string `?key=value`. Використовується переважно для **GET**-запитів – фільтрація, пагінація, пошук.

data

Тіло запиту у форматі JSON або form-data. Використовується для **POST**, **PUT**, **PATCH** – передача даних для створення або оновлення.

GET-запити

GET – найпоширеніший HTTP-метод, який використовується для отримання даних з сервера. У Playwright виконати GET-запит максимально просто – достатньо викликати метод `client.get()` з URL та опціональними параметрами. Параметри автоматично серіалізуються у query string.

Приклад використання

```
await client.get(  
  'https://example.com/api/getText',  
  {  
    params: {  
      isbn: '1234',  
      page: 23,  
    }  
  }  
);
```

□ **Результат:** Playwright сформує запит до URL:

`https://example.com/api/getText?isbn=1234&page=23`

Як це працює

Об'єкт `params` автоматично перетворюється на query string параметри, які додаються до URL після символу `?`.

Кожна пара ключ-значення з'єднується символом `=`, а між параметрами ставиться `&`.

Це повністю аналогічно тому, як формуються URL у браузері при заповненні форм або натисканні посилань з параметрами.

POST-запити

POST-метод використовується для створення нових ресурсів на сервері. У Playwright метод `client.post()` автоматично серіалізує об'єкт `data` у JSON і встановлює заголовок `Content-Type: application/json`, якщо ви не задали інший тип вмісту явно. Це значно спрощує код тестів і позбавляє необхідності робити `JSON.stringify()` вручну.

Приклад коду

```
await client.post(  
  'https://example.com/api/createBook',  
  {  
    data: {  
      title: 'Book Title',  
      author: 'John Doe'  
    }  
  }  
);
```

Що відбувається під капотом

- Об'єкт `data` серіалізується в JSON-рядок
- Автоматично встановлюється `Content-Type: application/json`
- Запит відправляється методом POST на вказаний URL
- Відповідь повертається як об'єкт `APIResponse`

POST з файлом (multipart)

Multipart – це спеціальний формат для відправки файлів разом із полями форми в одному HTTP-запиті. Playwright підтримує як потокову передачу файлів з диску (stream), так і відправку вмісту з пам'яті (buffer). Це особливо корисно для тестування ендпоїнтів завантаження файлів – аватарів, документів, CSV-імпорту тощо.

Варіант 1: файл з диску

Використовуйте `fs.createReadStream()` для потокового читання файлу. Ідеально для великих файлів, оскільки не завантажує весь вміст у пам'ять.

```
1 const stream = fs.createReadStream('team.csv');
  await client.post(
    'https://example.com/api/uploadTeamList',
    {
      multipart: {
        fileField: stream
      }
    }
  );
```

Варіант 2: вміст у пам'яті

Створіть файл програмно через `Buffer.from()`. Зручно для тестів, де вміст файла генерується динамічно.

```
2 await client.post(
  'https://example.com/api/uploadScript',
  {
    multipart: {
      fileField: {
        name: 'script.js',
        mimeType: 'text/javascript',
        buffer: Buffer.from('console.log("hello");')
      }
    }
  }
);
```

PUT і PATCH

Методи PUT і PATCH обидва використовуються для оновлення ресурсів, але мають принципову різницю в семантиці. Розуміння цієї різниці важливе для правильного тестування API та для перевірки того, що сервер коректно обробляє обидва типи запитів.

PUT – повна заміна

Очікується, що тіло запиту містить **повний** об'єкт ресурсу. Всі поля, яких немає в запиті, можуть бути скинуті до значень за замовчуванням.

```
await client.put(  
  'https://example.com/api/update/1',  
  {  
    data: {  
      title: 'New Title',  
      author: 'Jane Doe',  
    }  
  }  
);
```

PATCH – часткове оновлення

Оновлюються **лише** зазначені поля. Всі інші поля ресурсу залишаються без змін – це безпечніший підхід для точкових правок.

```
await client.patch(  
  'https://example.com/api/update/1',  
  {  
    data: {  
      author: 'Jane Doe',  
    }  
  }  
);
```

DELETE з додатковими опціями

DELETE-запити видаляють ресурси з сервера. Часто вони потребують додаткових опцій – авторизаційних заголовків, таймаутів або специфічних параметрів. Playwright дозволяє передати об'єкт опцій як другий аргумент методу `client.delete()`, що дає повний контроль над запитом.

Приклад з опціями

```
const options = {
  headers: {
    Authorization: 'Bearer YOUR_TOKEN',
    'Content-Type': 'application/json',
  },
  timeout: 5000,
};

await client.delete(
  'https://example.com/deleteEndpoint',
  options
);
```

Ключові моменти

Заголовки, передані в опціях конкретного запиту, **перезаписують** заголовки контексту (`extraHTTPHeaders`) для цього запиту.

Таймаут у запиті також має пріоритет над таймаутом контексту.

Це корисно, коли для DELETE потрібен інший токен або коротший таймаут, ніж для решти запитів.

Універсальний метод fetch

Для повного контролю над запитом Playwright надає метод `client.fetch()` – універсальний аналог стандартного `fetch` з довільними параметрами. Цей метод дозволяє вказати будь-який HTTP-метод, заголовки, тіло запиту та інші параметри в одному виклику. Він особливо корисний, коли вам потрібно відправити запит з нестандартним методом або комбінацією параметрів.

```
await client.fetch(  
  'https://example.com/api/createBook',  
  {  
    method: 'post',  
    headers: {  
      'Accept-Encoding': 'Infinity',  
    },  
    data: {  
      title: 'Book Title',  
      author: 'John Doe',  
    }  
  }  
);
```

Коли обирати `fetch`

Нестандартні HTTP-методи, потреба у повному контролі над параметрами запиту, або перехоплення маршрутів у E2E-тестах

Коли обирати `get/post/put`

Стандартні CRUD-операції з типовими параметрами – код буде коротшим і зрозумілішим



Використання fixture request у тестах

Playwright надає готовий **fixture** `request` – це вже створений `APIRequestContext` з налаштуваннями з `playwright.config`. Вам не потрібно вручну створювати контекст – достатньо вказати `baseURL` та заголовки в конфігурації, і `fixture` буде готовий до використання в кожному тесті.

Конфігурація

```
// playwright.config.ts
export default defineConfig({
  use: {
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      'Accept': 'application/vnd.github.v3+json',
    },
  },
});
```

Приклад тесту

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

const REPO = 'test-repo-1';
const USER = 'github-username';

test('should create a bug report', async ({ request }) => {
  const newIssue = await request.post(
    `/repos/${USER}/${REPO}/issues`,
  {
    data: {
      title: '[Bug] report 1',
      body: 'Bug description',
    }
  });
  expect(newIssue.ok()).toBeTruthy();

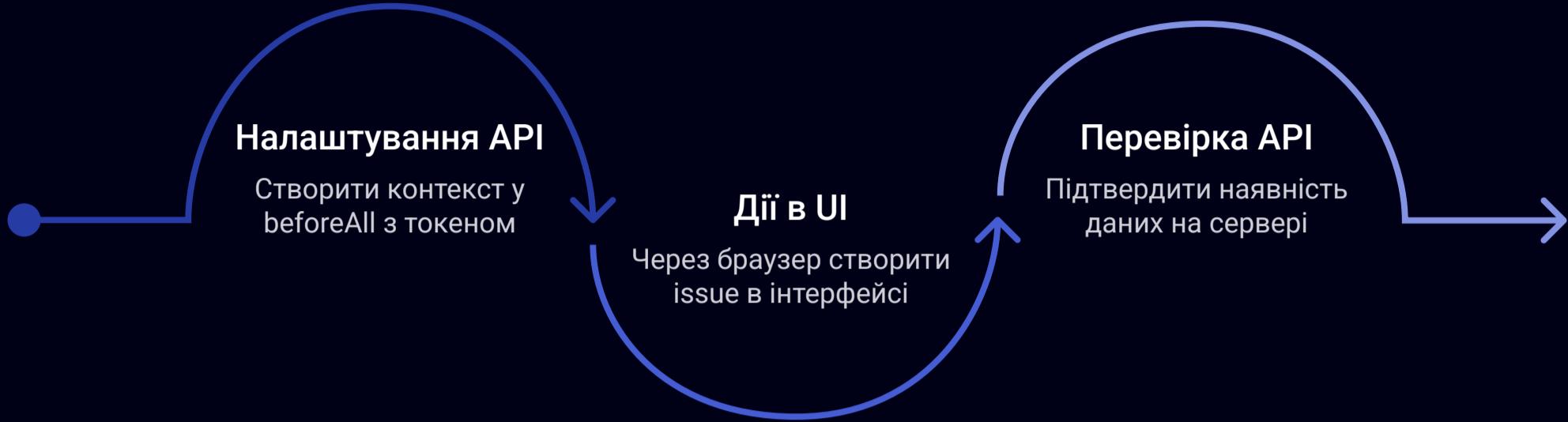
  const issues = await request.get(
    `/repos/${USER}/${REPO}/issues`
  );
  expect(issues.ok()).toBeTruthy();

  const json = await issues.json();
  expect(json).toContainEqual(
    expect.objectContaining({
      title: '[Bug] report 1',
      body: 'Bug description'
    })
  );
});
```

- **Важливо:** `newIssue.ok()` повертає `true` для статусів 200–299. Для перевірки тіла відповіді використовуйте `response.json()` або `response.text()`.

API-запити в E2E: setup через API

Замість виконання довгих дій у UI – реєстрації, логіну, створення тестових даних – можна зробити це через API у `beforeAll`. Такий підхід значно пришвидшує і стабілізує тести, оскільки API-запити виконуються в десятки разів швидше за UI-дії та не залежать від рендерингу сторінки.



Нижче – повний приклад, де API використовується для верифікації результатів UI-дій:

```
import { test, expect } from '@playwright/test';
const REPO = 'test-repo-1';
const USER = 'github-username';
let apiContext;

test.beforeAll(async ({ playwright }) => {
  apiContext = await playwright.request.newContext({
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      'Accept': 'application/vnd.github.v3+json',
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  });
}

test.afterAll(async () => {
  await apiContext.dispose();
});

test('last created issue should be on the server',
  async ({ page }) => {
    // Крок 1: Створити issue через UI
    await page.goto(`https://github.com/${USER}/${REPO}/issues`);
    await page.getByText('New Issue').click();
    await page.getRole('textbox', { name: 'Title' })
      .fill('Bug report 1');
    await page.getRole('textbox', { name: 'Comment body' })
      .fill('Bug description');
    await pageByText('Submit new issue').click();

    const issueId = page.url().substr(
      page.url().lastIndexOf('/') + 1
    );

    // Крок 2: Перевірити через API
    const newIssue = await apiContext.get(
      `/repos/${USER}/${REPO}/issues/${issueId}`
    );
    expect(newIssue.ok()).toBeTruthy();

    const body = await newIssue.json();
    expect(body).toEqual(
      expect.objectContaining({ title: 'Bug report 1' })
    );
  });
}
```

Context request vs Global request: огляд

Playwright підтримує два варіанти `APIRequestContext`, і вибір між ними залежить від конкретного сценарію тестування. Ключове питання: **чи потрібні спільні cookies з браузером?** Відповідь на нього визначає, який контекст використовувати.

Контекст, прив'язаний до браузера (`context.request` або `page.request`), ділить cookies з браузерним контекстом. Це означає, що якщо ви залогінились через UI, API-запити автоматично будуть авторизованими. І навпаки – cookies з API-відповідей потраплять у браузер.

Глобальний контекст (`playwright.request.newContext()`) повністю ізольований. Він має власне cookie-сховище і не впливає на браузер. Це ідеальний вибір для чистих API-тестів або коли потрібна повна незалежність від UI-сесії.



Context request

Спільні cookies з браузером

Який обрати?

Залежить від сценарію

Global request

Ізольоване cookie-сховище

Контекст, прив'язаний до браузера

Коли ви використовуєте `context.request` або `page.request`, API-запити діляться cookie-сховищем з `BrowserContext`. Це створює єдину сесію між браузером і API-клієнтом – потужний інструмент для комбінованих E2E+API тестів.



Спільний cookie storage

Cookies зберігаються разом з `BrowserContext` – одна сесія для UI та API



Автоматична синхронізація

Запити додають cookies браузера, а cookies з API-відповідей оновлюють браузерний контекст



Єдина авторизація

Залогінились через UI – API-запити вже авторизовані автоматично

Коли використовувати: коли API і UI використовують одну й ту ж сесію (одні й ті самі cookies), наприклад – логін через UI і подальші API-виклики для перевірки даних у тому ж контексті.



Глобальний (ізольований) контекст

Глобальний контекст створюється через `playwright.request.newContext()` і є повністю незалежним від браузера. Він має власне cookie-сховище, не впливає на жоден `BrowserContext` і не отримує від них cookies. Це робить його ідеальним для чистих API-тестів.

Власний cookie storage

Створюється через `playwright.request.newContext()` – повністю ізольований від браузерних контекстів та їхніх cookies

Підходить для чистих API-тестів

Коли cookies не потрібні або коли потрібна повна ізоляція від браузерного контексту – це правильний вибір

Потребує явного dispose

Не забувайте викликати `apiContext.dispose()` після завершення тестів, щоб звільнити ресурси

Порівняння контекстів

Щоб обрати правильний контекст, потрібно чітко розуміти різницю між ними. Нижче – детальне порівняння двох підходів за ключовими аспектами. Це допоможе вам швидко прийняти рішення для конкретного сценарію тестування.

Аспект	Context request	Global request
Cookie storage	Спільний з браузером	Ізольований
Автоматичні cookies	Так – двостороння синхронізація	Ні – повна ізоляція
Створення	<code>context.request / page.request</code>	<code>playwright.request.newContext()</code>
Типове використання	E2E з API-перевірками	API-only тести
Dispose	Автоматично з контекстом	Потрібно вручну

Context request

Обираєте, коли тест поєднує UI-дії з API-перевірками і потрібна єдина авторизована сесія

Global request

Обираєте для чистих API-тестів, setup/teardown або коли потрібна гарантована ізоляція

Приклад: context.request (спільні cookies)

У цьому прикладі демонструється, як context.request автоматично синхронізує cookies з браузерним контекстом. Ми переходжуємо маршрут, виконуємо API-запит через context.request.fetch() і перевіряємо, що cookies з відповіді автоматично потрапили в браузерний контекст.

```
test('context request shares cookies with browser',
  async ({ page, context }) => {

    await context.route('https://www.github.com/',
      async route => {
        const response = await context.request.fetch(
          route.request()
        );
        const responseHeaders = response.headers();
        const setCookie = responseHeaders['set-cookie'];

        // Парсимо cookies з відповіді
        const responseCookies = new Map(
          setCookie.split('\n').map(c => {
            const [name, value] =
              c.split(';', 2)[0].split('=');
            return [name.trim(), value];
          })
        );
        expect(responseCookies.size).toBe(3);

        // Перевіряємо: cookies браузера == cookies відповіді
        const contextCookies = await context.cookies();
        const contextMap = new Map(
          contextCookies.map(({ name, value }) =>
            [name, value]
          )
        );
        expect(contextMap).toEqual(responseCookies);

        route.fulfill({
          response,
          headers: { ...responseHeaders, foo: 'bar' },
        });
      });

    await page.goto('https://www.github.com/');
  });
}
```

- ☐ **Ключовий момент:** cookies з API-відповіді автоматично з'являються у context.cookies() – саме це і є спільний cookie storage.

Приклад: global request (ізольовані cookies)

Цей приклад показує протилежний підхід – використання глобального контексту, де cookies **не** синхронізуються з браузером автоматично. Щоб перенести cookies з ізольованого контексту в браузер, потрібно явно використати `storageState`.

```
test('global request has isolated cookie storage',
  async ({ page, context, browser, playwright }) => {

    const request = await playwright.request.newContext();

    await context.route('https://www.github.com/',
      async route => {
        const response = await request.fetch(
          route.request()
        );
        const responseHeaders = response.headers();
        const setCookie = responseHeaders['set-cookie'];

        const responseCookies = new Map(
          setCookie.split('\n').map(c =>
            c.split(';', 2)[0].split('=')
          )
        );
        expect(responseCookies.size).toBe(3);

        // Браузер НЕ отримав cookies!
        const contextCookies = await context.cookies();
        expect(contextCookies.length).toBe(0);

        // Але можна перенести через storageState
        const storageState = await request.storageState();
        const browserContext2 = await browser.newContext({
          storageState
        });
        const contextCookies2 =
          await browserContext2.cookies();
        expect(new Map(
          contextCookies2.map(({ name, value }) =>
            [name, value]
          )
        )).toEqual(responseCookies);

        route.fulfill({
          response,
          headers: { ...responseHeaders, foo: 'bar' },
        });
      });
    });

    await page.goto('https://www.github.com/');
    await request.dispose();
  });
}
```

Ізоляція

`contextCookies.length === 0` – браузер не отримав cookies від глобального контексту

Перенесення

Через `request.storageState()` можна явно передати cookies в новий браузерний контекст

Поширені помилки

Навіть досвідчені розробники припускаються типових помилок при API-тестуванні з Playwright. Знання цих пасток заздалегідь збереже вам години налагодження. Розглянемо чотири найпоширеніші проблеми та способи їх уникнення.

1 Плутанина params і data

Для query string параметрів у URL використовуйте `params`, а для тіла запиту – `data`. Якщо ви передасте `data` у GET-запиті, сервер, швидше за все, його проігнорує. А `params` у POST додасть параметри в URL, а не в тіло.

2 `response.json()` – лише один раз

Виклик `response.json()` витрачає тіло відповіді. Якщо вам потрібно використати дані кілька разів – **зберіжіть результат у змінну**: `const data = await response.json();` і працюйте з `data`.

3 Забули `dispose context`

Якщо ви створюєте власний контекст у `beforeAll`, обов'язково викличте `apiContext.dispose()` у `afterAll`. Без цього ресурси не звільняються, що може привести до витоків пам'яті та нестабільних тестів.

4 `baseURL` і повний URL

При налаштованому `baseURL` відносні шляхи (`/api/users`) доповнюються базовим URL автоматично. Але повний URL (`https://...`) використовується як є – `baseURL` ігнорується. Будьте послідовні у вашому підході.

Підсумок

Ви пройшли повний шлях від базових концепцій API-тестування до просунутих сценаріїв із контекстами та cookies. Тепер у вас є всі інструменти для ефективного тестування API за допомогою Playwright. Головне – обрати правильний підхід для кожного сценарію.

Задача	Рішення
Простий API-тест	Fixture request + baseURL у config
Авторизація	extraHTTPHeaders, storageState або httpCredentials
Підготовка даних для E2E	playwright.request.newContext() у beforeEach
Спільні cookies з браузером	context.request / page.request
Ізольовані API-тести	playwright.request.newContext()



Швидкість

API-тести виконуються в десятки разів швидше за E2E



Стабільність

Немає залежності від UI-рендерингу та динамічних елементів



Гнучкість

Один інструмент для E2E, API та комбінованих сценаріїв

Починайте з простих тестів, використовуючи fixture request, і поступово переходьте до складніших сценаріїв з кастомними контекстами. Успіхів у тестуванні!