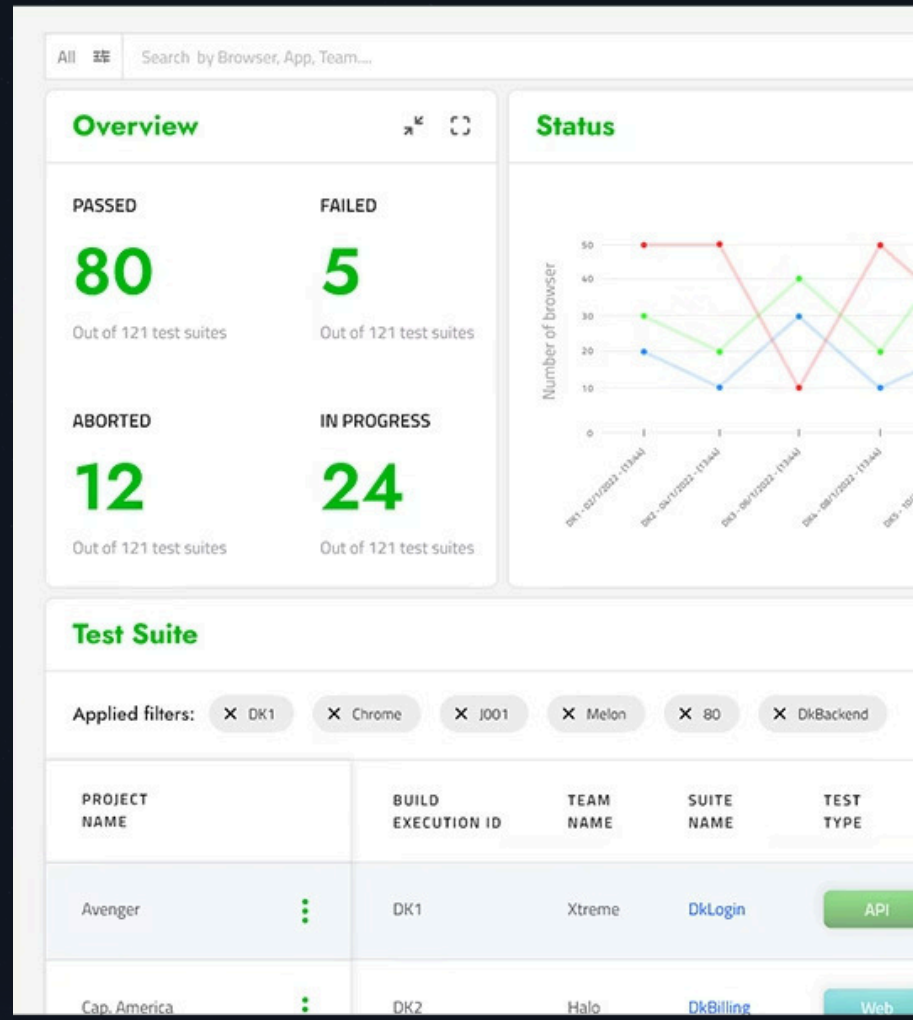
Розширена конфігурація API тестів

Професійний підхід до налаштування та оптимізації автоматизованих тестів API з використанням Axios та Jest



Автоматичне перетворення даних у JSON

Базові можливості Axios

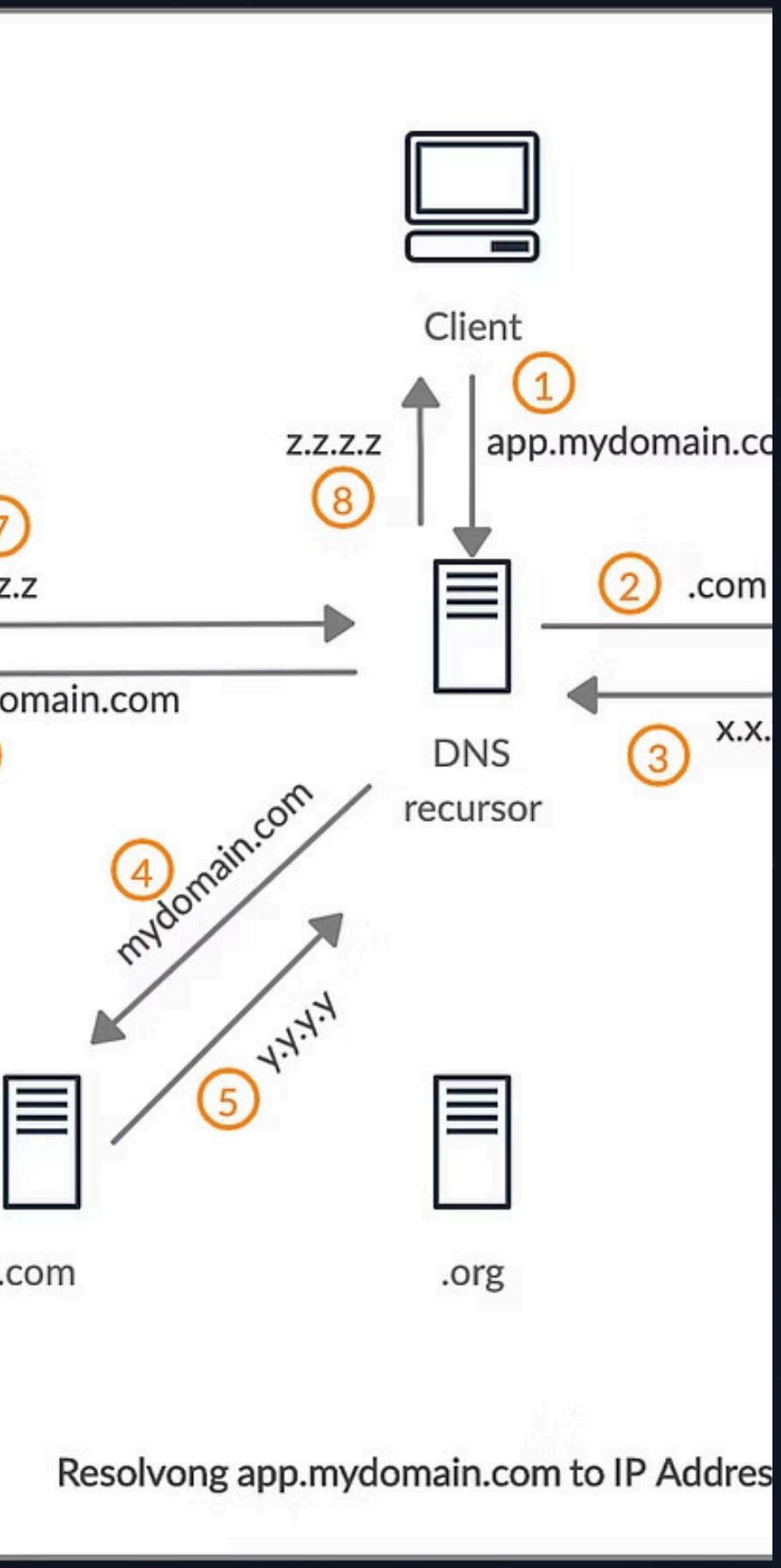
За замовчуванням Axios автоматично налаштовує Content-Type на application/json при надсиланні запитів до сервера. Це суттєво спрощує роботу з API та зменшує кількість рутинного коду.

Якщо сервер не відповідає стандарту JSON, Axios надає метод `transformResponse` для кастомізації обробки відповіді від сервера.

Переваги підходу

- Автоматична серіалізація даних
- Зменшення boilerplate коду
- Вбудована обробка помилок
- Підтримка Promise API
- Гнучкість конфігурації

```
const axios = require('axios');

async function fetchData() {
 try {
 const res = await axios.get('https://httpbin.org/get', {
 params: { answer: 42 }
 });

 console.log(res.constructor.name); // 'Object'
 console.log(res.data); // Отримані дані
 console.log(res.data instanceof Object); // true
 } catch (error) {
 console.error('Error fetching data:', error);
 }
}

fetchData();
```



Автоматичний парсинг POST і PUT запитів

Axios автоматично перетворює JavaScript об'єкти в JSON при відправці POST або PUT запитів, додаючи відповідні заголовки та форматуючи тіло запиту.

POST запити

Автоматична серіалізація об'єктів у JSON формат з налаштуванням Content-Type заголовка

PUT запити

Оновлення даних з автоматичною обробкою та перетворенням payload

Приклад POST

```
// Автоматична серіалізація { answer: 42 }
const res = await axios.post(
  'https://httpbin.org/post',
  { answer: 42 }
);

res.data.data;
// '{"answer":42}'

res.data.headers['Content-Type'];
// 'application/json;charset=utf-8'
```

Приклад PUT

```
const res = await axios.put(
  'https://httpbin.org/put',
  { hello: 'world' }
);

res.data.json;
// { hello: 'world' }
```

Параметри запиту GET із Axios

При роботі з GET запитами параметри можна передавати двома способами: безпосередньо в URL або через об'єкт конфігурації params.

1

Прямий URL

Параметри вказуються безпосередньо в рядку запиту після знака питання

```
axios.get('https://jsonplaceholder.typicode.com/comments?postId=1')
```

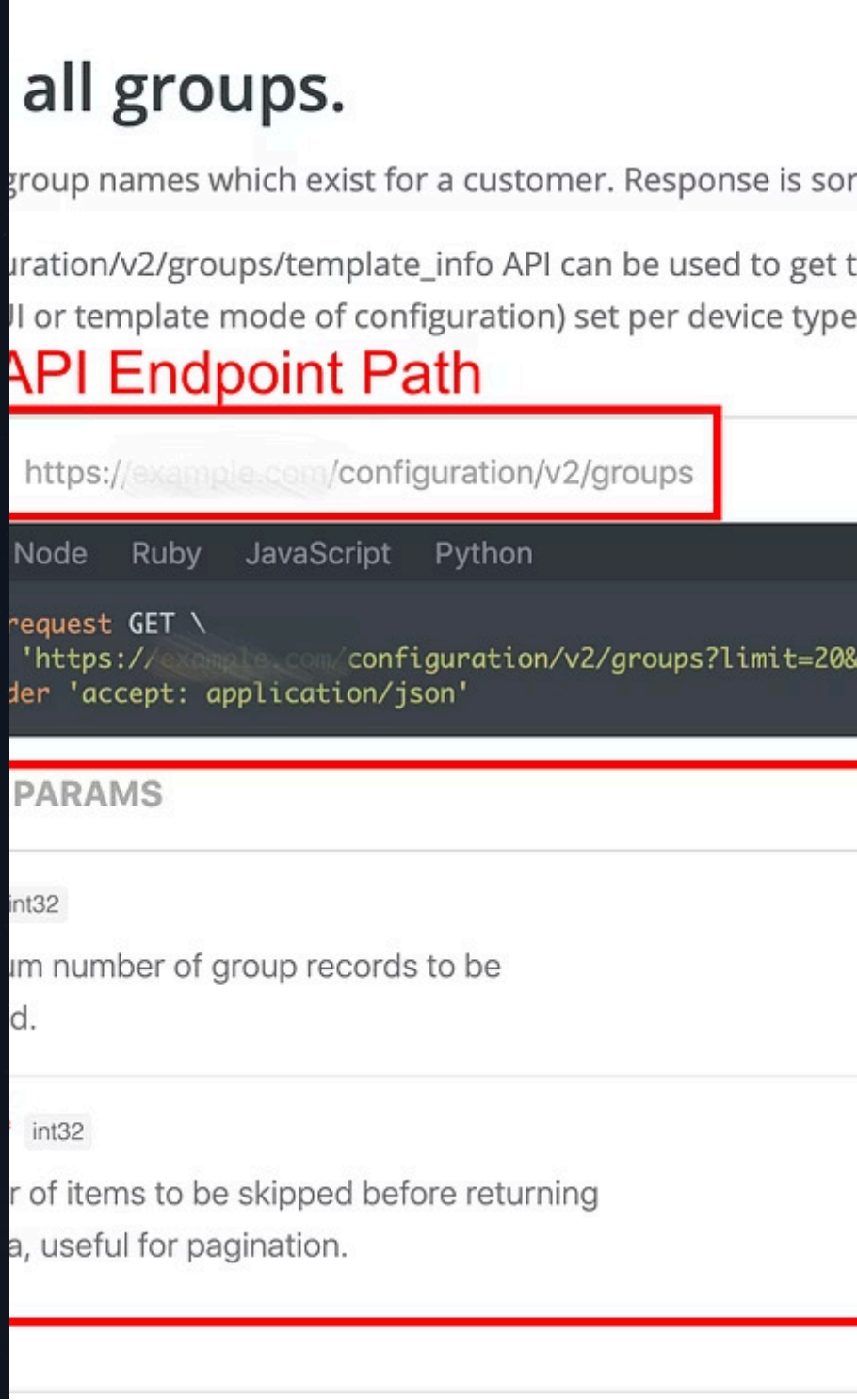
2

Об'єкт params

Використання options.params для чистішого та читабельнішого коду

```
const res = await axios.get(
  'https://jsonplaceholder.typicode.com/comments',
  { params: { postId: 1 } }
);
```

Рекомендація: Використовуйте об'єкт params для динамічних параметрів та складних запитів. Це забезпечує кращу читабельність коду та спрощує підтримку.



URLSearchParams для параметрів запиту

Альтернативний підхід

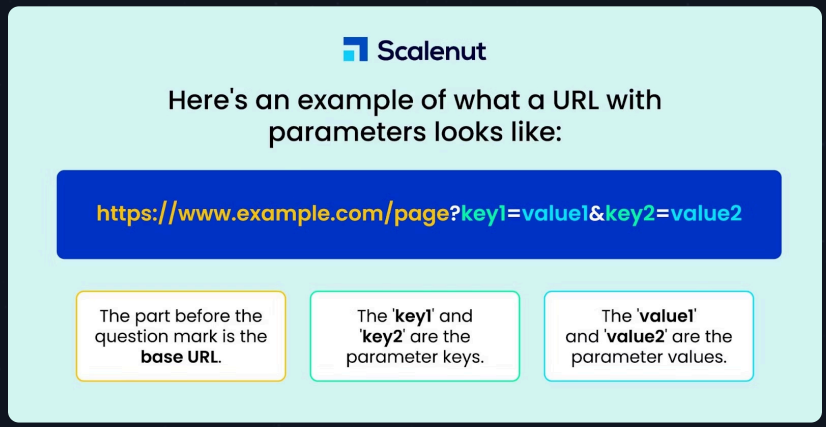
Для більш складних сценаріїв можна використовувати вбудований клас `URLSearchParams`, який надає додаткову гнучкість при роботі з параметрами запиту.

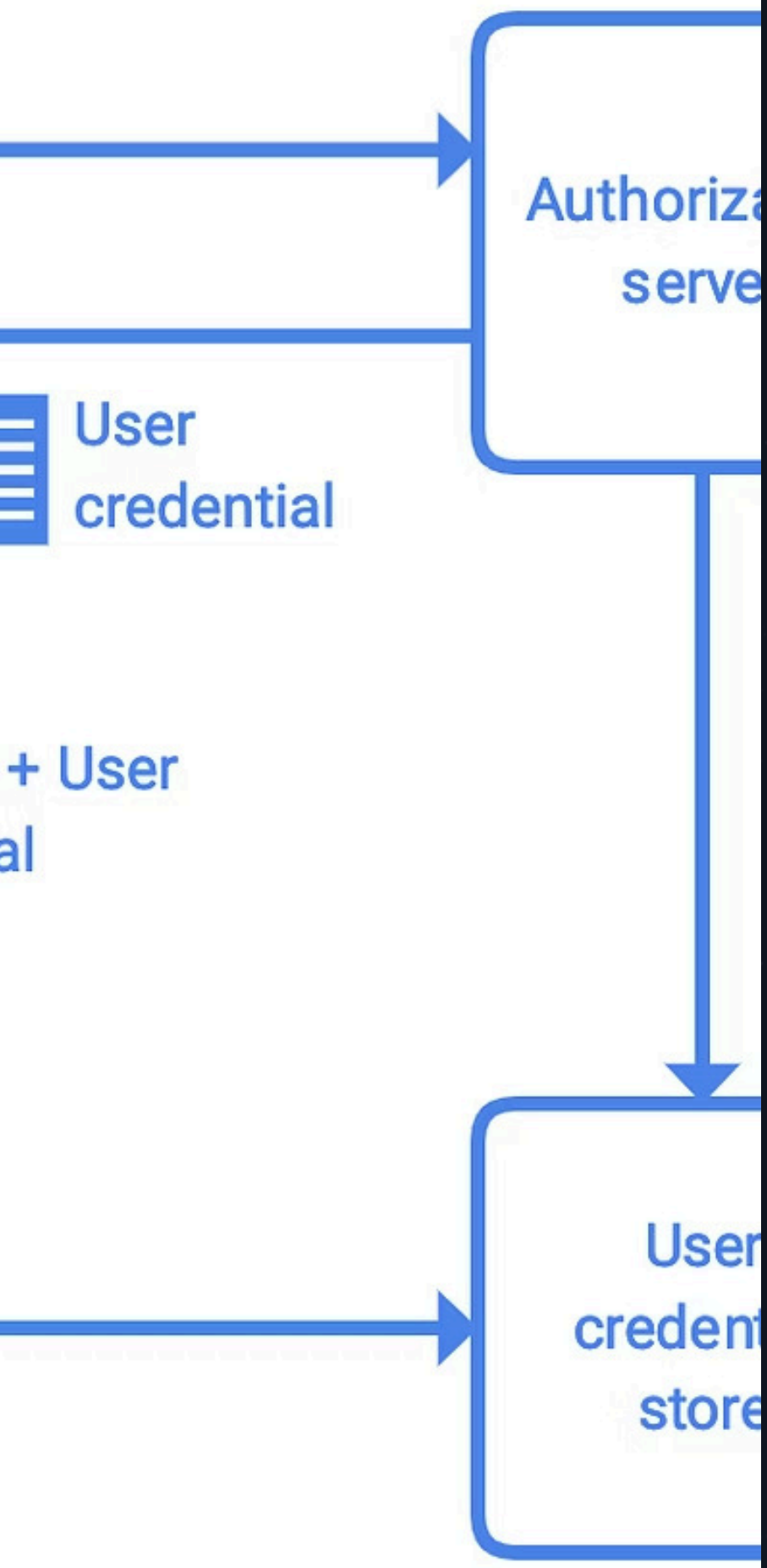
Цей підхід особливо корисний при динамічному формуванні параметрів або при роботі з масивами значень.

```
const params = new URLSearchParams([
  ['postId', 1]
]);

const res = await axios.get(
  'https://jsonplaceholder.typicode.com/comments',
  { params }
);

res.data.args; // { postId: 1 }
```

Переваги

- Стандартний Web API
- Підтримка множинних значень
- Автоматичне кодування
- Гнучкість у побудові





Оптимізація: винесення логіну в окремий файл

Для значного скорочення часу виконання тестів критично важливо оптимізувати процес автентифікації. Замість генерації токену в кожному тестовому файлі, створюємо централізований механізм отримання та збереження токену.

01

Створення login.test.js

Окремий файл для процесу автентифікації та отримання токену

02

Збереження токену

Токен зберігається в env.json для подальшого використання

03

Запуск перед тестами

Файл login.test.js виконується перед усіма іншими тестами

```
const axios = require('axios');
const jsonData = require('./env.json');
const fs = require('fs');

test("login and getting token", async () => {
 var response = await axios.post(
 `${jsonData.baseUrl}/user/login`,
 {
 "email": "salman@roadtocareer.net",
 "password": "1234"
 },
 {
 headers: {
 "Content-Type": "application/json",
 }
 }
 );

 console.log(response.data);
 expect(response.data.message).toContain("Login successfully");

 let token_value = response.data.token;
 jsonData.token = token_value;
 fs.writeFileSync('env.json', JSON.stringify(jsonData));
});
```

Використання токену в тестах

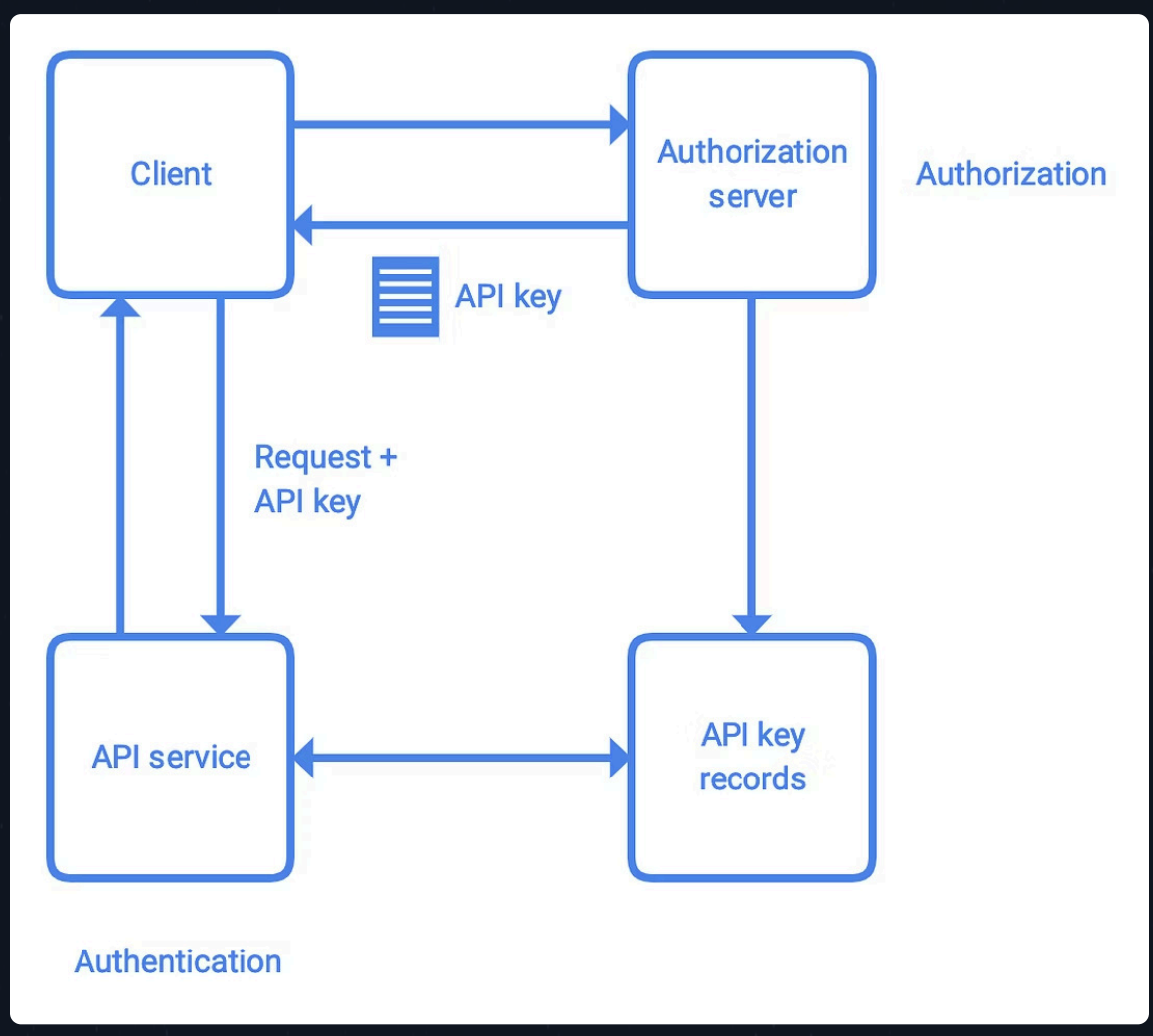
Після створення централізованого механізму автентифікації, імпортуємо env.json файл у тестові файли для використання отриманого токену в авторизованих запитах.

Структура файлу

Імпортуємо необхідні залежності та env.json конфігурацію:

- axios для HTTP запитів
- jsonData з токеном
- fs для роботи з файлами

Оголошуємо змінні для тестових даних, які будуть використовуватись протягом тестування.




```
const axios = require('axios');
const jsonData = require('./env.json');
const fs = require('fs');

let userId;
let userName;
let userLName;
let userPwd;
let token;


test('create product', async () => {
  const createProduct = await axios.post(
    `${jsonData.baseUrl}/products/add`,
    { 'title': 'MyOwnProduct' },
    {
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${jsonData.token}`
      }
    }
  );

  console.log(createProduct.data);
  expect(createProduct.status).toEqual(200);
});
```

Важливо: Токен додається в заголовок Authorization з префіксом Bearer для коректної автентифікації на сервері.

Послідовний запуск тестів

Генерація токену

login.test.js виконується першим

1

Збереження даних

Токен записується в env.json

2

Виконання тестів

actions.test.js використовує токен

3

Команда для запуску

Для об'єднання та послідовного виконання кількох команд використовується оператор `&&`, який гарантує, що наступна команда виконається тільки після успішного завершення попередньої.

```
npx jest api_requests/login.test.js && npx jest api_requests/actions.test.js
```

Переваги підходу

- Скорочення часу виконання тестів
- Централізована автентифікація
- Простіше управління токенами
- Зменшення навантаження на сервер

Оператор &&

- Послідовне виконання команд
- Зупинка при помилці
- Гарантія порядку виконання
- Чіткий контроль flow