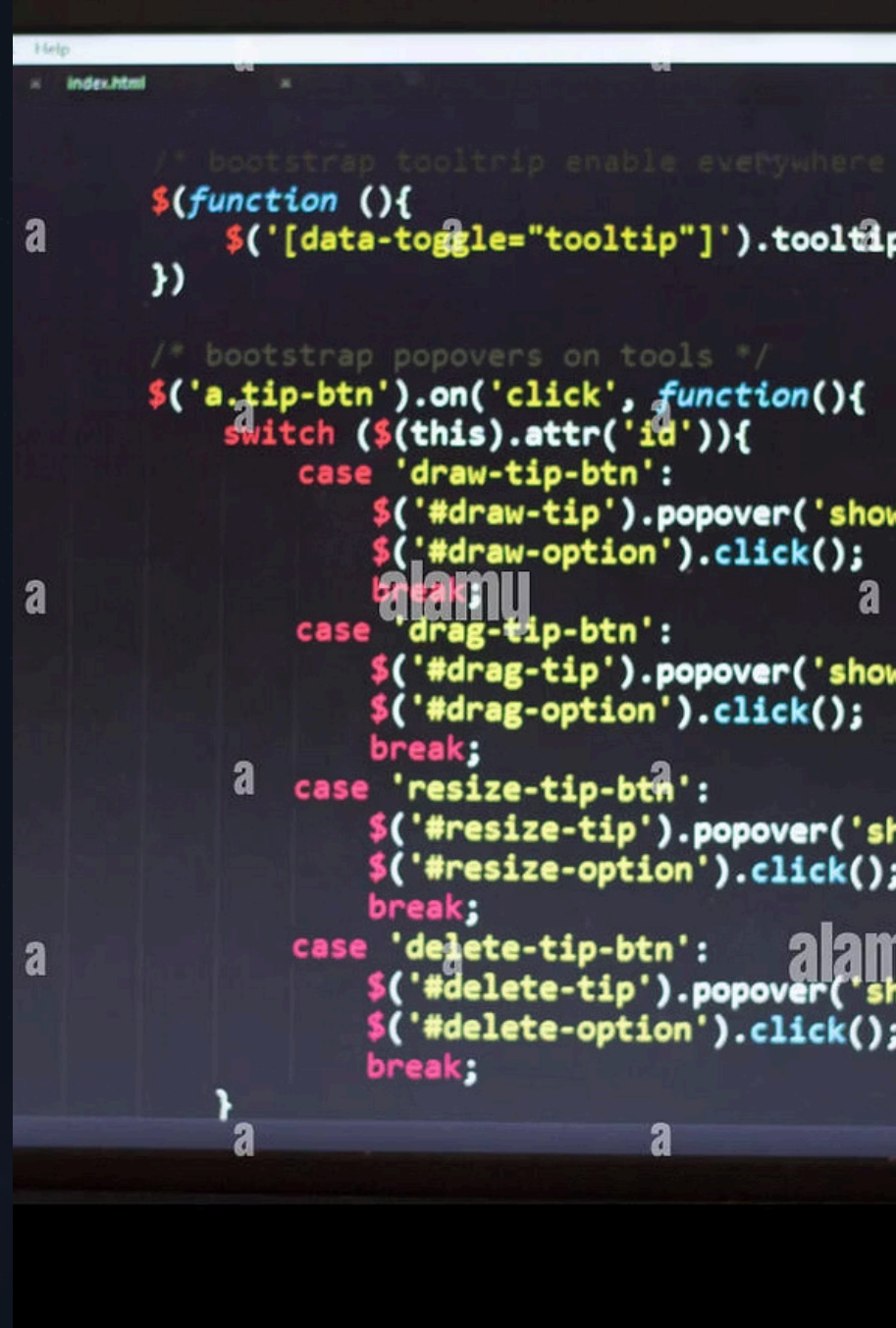
Fetch API: Сучасний підхід до мережевих запитів

Fetch API - це сучасний стандарт для виконання HTTP-запитів у браузері, який змінив підхід до роботи з мережевими операціями. На відміну від застарілого XMLHttpRequest, Fetch надає елегантний інтерфейс на основі промісів, що робить код більш читабельним та зручним у підтримці.

Цей API підтримує всі основні HTTP-методи: GET для отримання даних, POST для створення нових ресурсів, PUT для оновлення існуючих та DELETE для видалення. Fetch API став стандартом де-факто для сучасної веб-розробки завдяки своїй простоті та потужності.



Основи роботи з Fetch API

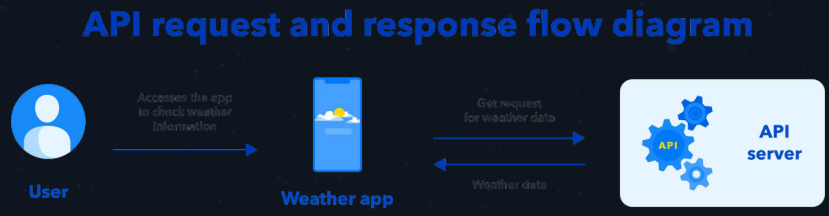
Базова структура

Функція `fetch()` є серцем API. Вона приймає два параметри: URL ресурсу та необов'язковий об'єкт конфігурації.

Ключові компоненти конфігурації:

- **method** - HTTP-метод запиту
- **headers** - заголовки запиту
- **body** - тіло запиту для POST/PUT

Після виконання `fetch()` повертає проміс з об'єктом Response, який містить статус відповіді, заголовки та методи для обробки даних.



Важливо: Fetch повертає проміс, який вирішується навіть при статусі 404 або 500. Потрібно перевіряти `response.ok` для визначення успішності.

Робота з HTTP-методами через проміси

GET запит

Отримання даних з сервера

```
fetch('https://api.example.com/users/1')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

POST запит

Створення нового ресурсу

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {'Content-Type': 'application/json'},
  body: JSON.stringify(newUser)
})
```

PUT запит

Оновлення існуючого ресурсу повністю

```
fetch('https://api.example.com/users/1', {
  method: 'PUT',
  headers: {'Content-Type': 'application/json'},
  body: JSON.stringify(updatedUser)
})
```

DELETE запит

Видалення ресурсу з сервера

```
fetch('https://api.example.com/users/1', {
  method: 'DELETE'
}).then(response => {
  if (response.ok) console.log('Видалено');
})
```

Всі запити використовують ланцюжок промісів з методами `.then()` для обробки успішних відповідей та `.catch()` для перехоплення помилок. Такий підхід забезпечує чіткість коду та легкість обробки асинхронних операцій.

Сучасний підхід: async/await

Синтаксис async/await робить асинхронний код схожим на синхронний, підвищуючи читабельність. Замість ланцюжків `.then()` використовуються звичайні конструкції з `await`.

Базова функція-обгортка

```
async function fetchData(url, options) {
  try {
    const response = await fetch(url, options);
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}
```

Приклад використання

```
async function createUser(newUser) {
  const options = {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(newUser)
  };
  const created = await fetchData(
    'https://api.example.com/users',
    options
  );
  console.log('Створено:', created);
}
```

01

Створіть async функцію

Додайте ключове слово `async` перед оголошенням функції

02

Використовуйте await

Застосуйте `await` перед викликом `fetch()` та методами Response

03

Обробіть помилки

Використовуйте блок `try/catch` для перехоплення помилок

04

Переваги async/await: Більш інтуїтивний синтаксис, легше відлагоджувати, природна обробка помилок через try/catch, можливість використання циклів з асинхронними операціями.