Logika

МОДУЛЬ 5. УРОК 3.

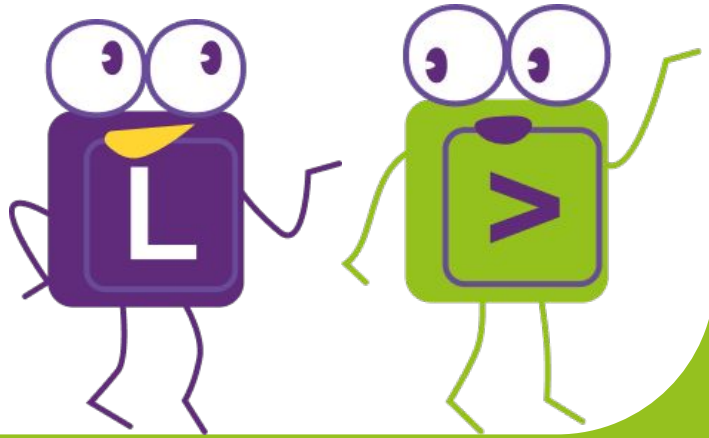
Асинхронність і запити до сервера

logikaschool.com

@logika_it_school

МОДУЛЬ 5. УРОК 3.

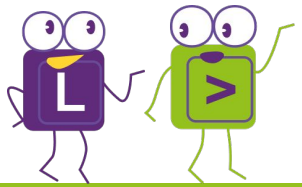
Обговорення. Дані на сайті



Ми з вами вже крута команда! 💪
Ми вміємо верстати (HTML/CSS), писати логіку (JS)

Але... Всі наші програми працюють лише всередині вашого браузера.

10:12

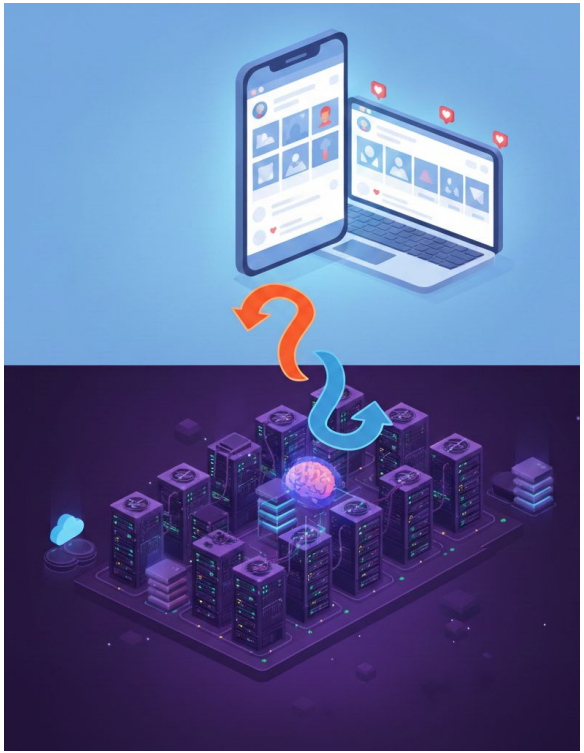


Як працюють вебзастосунки

Будь-який серйозний вебсервіс (Instagram, TikTok, Google) складається з двох частин.

- ➔ **Frontend (це ми):** картинка, кнопки, анімації. Те, що бачить користувач у браузері.
- ➔ **Backend (сервер):** "мозок" і "пам'ять" сайту. Комп'ютер десь в дата-центрі, який зберігає всі фото, лайки, повідомлення.

☝ **Щоб твій сайт побачили інші, фронтенд має “подружитися” з бекендом.’**

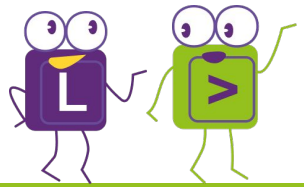


Взаємодія бекенду та фронтенду

Щоб мати доступ до актуальних даних у будь-якій точці світу, вони **зберігаються на сервері**

Тому потрібно навчитися отримувати та надсилати дані **з фронтенду на бекенд і навпаки**.

Сьогодні ми навчимо наш код "дзвонити" серверу і питати: "Ей, яка там погода в Лондоні?"



Але не все так просто

Сервер може знаходитися в США, а ви в Україні. Сигнал проходить по кабелях на дні океану (!!!) дуже швидко, але не миттєво.

➔ Запит може йти 0.5 секунди, 1 секунду або навіть 5 секунд.

➔ **Сайт не може просто чекати відповіді – він “зависне”.** 😕

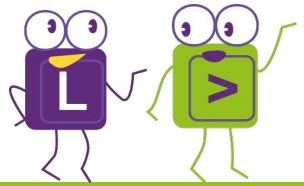
Рішення — асинхронність

Інший приклад, коли ви пишете:

```
button.addEventListener('click', () => {
    alert("Клік!");
});
```

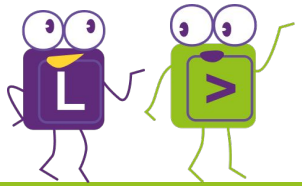
Хіба програма зупиняється на цьому рядку і чекає, поки ви клікнете? **Ні, це спрацьовує “на фоні”.**

В обох випадках використовується асинхронність! Гайда розбиратися! 🧑‍💻



Сьогодні на уроці

1. Вивчимо що таке **асинхронність**.
2. Навчимося робити запити на сервер.
3. Розберемося, що таке **API** (спосіб спілкування з сервером).
4. **Створимо додаток погоди, який показує реальну температуру за вікном!** 🌦️



Повторення: Git

Часто плутають Git та GitHub. В чому головна різниця?

- **Git** — це програма на твоєму комп'ютері. "Машина часу" для коду.
- **GitHub** — це сайт (хмара). "Соціальна мережа" для розробників, де ми зберігаємо проєкти.

Що таке репозиторій ?
Як його створити?

Репозиторій – це папка, що зберігає всі версії та історію змін кожного файлу.

Для перетворення папки на репозиторій, потрібна команда:

```
git init
```

Які команди треба ввести по черзі, щоб зберегти зміни в історії Git та відправити в інтернет?

1. `git add .` — додає всі нові та змінені файли до списку підготовки перед збереженням.
2. `git commit -m "..."` — фіксує підготовлені зміни, створюючи новий коміт (точку збереження) на твоєму комп'ютері.
3. `git push` — відправляє твої коміти на віддалений сервер GitHub.

Навіщо нам створювати гілки (branches) в Git?

Щоб працювати над новою фічею і не зламати основний працюючий код.

Як створити нову гілку dev і як злити її після розробки?

➔ Створити гілку dev:

```
git checkout -b dev
```

➔ Злити гілку dev з поточною:

```
git merge dev
```

МОДУЛЬ 5. УРОК 3.

Нова тема:
Асинхронність

Синхронний код

Подивимося на приклад **синхронного** коду:

```
console.log('I')
console.log('Love')
console.log('Logika')
```

```
>>> I
>>> Love
>>> Logika
```

Усі команди **виконуються по черзі, одна за одною**.

Другий console.log спрацює лише тоді, коли завершив роботу перший.

Синхронний код

Навіть якщо додамо функцію:

```
function print_love() {
    console.log('Love')
}
console.log('I')
print_love()
console.log('Logika')
```

```
>>> I
>>> Love
>>> Logika
```

Код при цьому залишиться синхронним, а результат передбачуваним: як написали — так і буде.

А тепер... подивіться уважно

Розглянемо приклад коду із setTimeout. **Що буде у консолі?**

```
console.log("1.Початок");
setTimeout(() => {
    console.log("2.Всередині таймера");
}, 2000);
console.log("3.Кінець");
```

```
>>> 1.Початок
>>> 3.Кінець
>>> 2.Всередині таймера
```

Чому?

Код же послідовно виконується...

Асинхронний код

Річ у тім, що setTimeout — **асинхронна функція**.

1. JS виконав "**Початок**".
2. Дійшов до setTimeout. Побачив, що це "довга задача".
3. Він **НЕ став чекати**. Він "відклав" код на потім і побіг виконувати код далі ("**Кінець**").
4. Тільки через 2 секунди, він повернувся до команди яка була у таймері.

Асинхронність — це інструмент, який дозволяє виконувати деякі команди “на фоні” і не блокувати основний код.

Ми вже це робили!

addEventListener — це теж асинхронність!

```
console.log("Сайт завантажився");
button.addEventListener('click', () => {
    console.log("Клік по кнопці!");
});
console.log("Сайт працює далі...");
```

```
>>> Сайт завантажився
>>> Сайт працює далі...
>>> Сайт завантажився
```

👉Ми не знаємо, КОЛИ юзер клікне (через секунду чи через годину). Тому JS просто додає “слухач” події і йде працювати далі.

Event Loop

Щоб керувати потоком викликів (Callback Queue) так, щоб все йшло за планом, у JavaScript існує **Event Loop** – **менеджер асинхронних викликів**.

Його завдання — вчасно додавати в потік викликів мікрозавдання (callback), якщо черга при цьому вільна.

Куди потрапляють відкладені задачі поки JS виконує код далі?

- Вони потрапляють у "залу очікування" (**Web API**) у браузері.
- Головний потік JS (**Call Stack**) виконує швидкі задачі.
- Коли "відкладена" задача (таймер/запит) готова — вона стає в чергу, щоб повернутися в головний потік.

Чому це важливо для нас сьогодні?

Запит до сервера — це така сама "довга задача", як і **setTimeout**.

Сервер може відповідати довго. **Ми НЕ можемо зупинити сайт поки чекаємо відповідь.**

Тому запити до сервера завжди асинхронні ☝

МОДУЛЬ 5. УРОК 3.

Нова тема:
Запити до сервера

Як спілкуватися з сервером?

Наш фронтенд написаний на JavaScript. А сервер (бекенд) може бути на Python, Java або C# або іншій мові програмування.

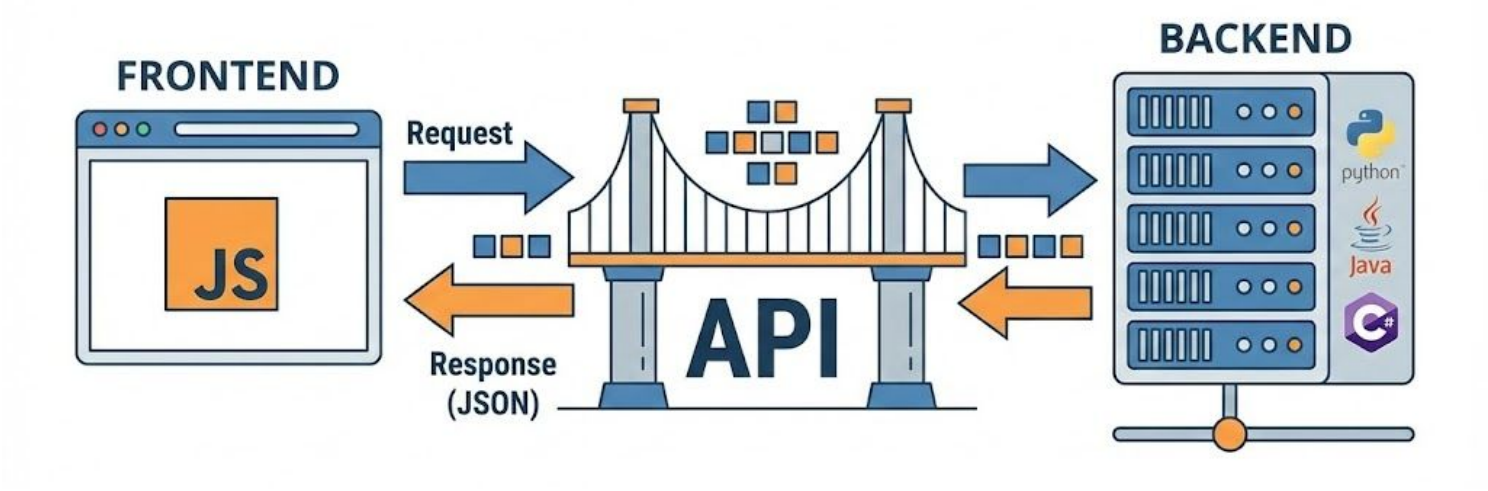
Виникає проблема: як змусити дві різні мови програмування зрозуміти одна одну й обмінятися даними?

- ➔ **Потрібен універсальний стандарт або протокол спілкування.**
- ➔ **Зазвичай для цього використовують API.**

Що таке API?

API (Application Programming Interface) — це програмний інтерфейс застосунку.

Це набір чітких правил, за якими одна програма може звернутися до іншої.



Програми постійно спілкуються з іншими програмами

Наприклад, саме по API можна

- отримувати погоду
- підвантажувати повідомлення
- перекладати текст
- показувати карти
- працювати з ШІ
- оплачувати на сайтах
- входити в профіль через Google тощо..

Фактично, весь сучасний веб побудований на API.

API

По суті, API — це просто набір URL-адрес і правил, за якими:

- ➔ фронтенд надсилає запити у вигляді URL
- ➔ сервер повертає дані
- ➔ дані подані в універсальному зрозумілому і для сервера, і для клієнта форматі (зазвичай **JSON**)

Мова спілкування – JSON

Щоб дані були зрозумілі будь-якій мові (і JS, і Python), використовується універсальний текстовий формат — **JSON** (JavaScript Object Notation).

Структура:

- ➔ Виглядає як звичайний JS-об'єкт (або масив об’єктів).
- ➔ Головна відмінність: **ключі завжди в подвійних лапках** "".
- ➔ Це просто текст (String), який ми передаємо мережею.

JSON

```
[
    {
        "name": "Володимир",
        "age": 14,
        "email": "vova@logikaschool.com"
    },
    {
        "name": "Вікторія",
        "age": 15,
        "email": "viktoria_ua@logikaschool.com"
    }
]
```

Це дозволяє дуже зручно використовувати такі дані на сайті.

API з котиками

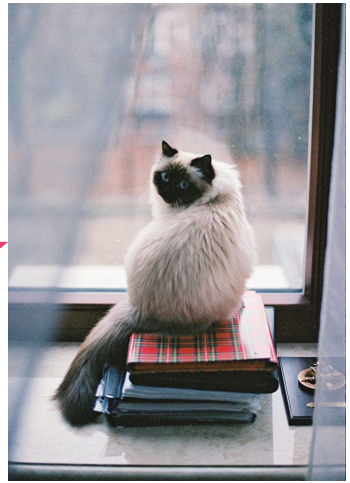
Розглянемо на прикладі простого [The Cat API](), що повертає випадкове фото котика по URL 😺

Відкриємо це посилання у браузері:
https://api.thecatapi.com/v1/images/search

Ми бачимо JSON-масив з об'єктом всередині.

```
[
  {
    "id": "afv",
    "url": "https://cdn2.thecatapi.com/images/afv.jpg",
    "width": 700,
    "height": 990
  }
]
```

Фото котика



Функція fetch()

Для асинхронного обміну даними між браузером і сервером, найчастіше використовують функцію **fetch**().

Для такого запиту використаємо асинхронну функцію

```
fetch(url, options)
```

якій передають аргументи:

`url` — це адреса запиту до сервера,

`options` — додаткові параметри запиту (необов'язково)

Асинхронний запит

Якщо ми просто викличемо функцію отак:

```
const data = fetch('https://api.thecatapi.com/...');

console.log(data);
```

Здається все зрозуміло — в змінну **data** потрапить результат функції fetch, тобто відповідь від API сервера. **Чи не так?**

Асинхронний запит

```
const data = fetch('https://api.thecatapi.com/...');
console.log(data);
```

Ні, не зовсім. ...ми побачимо в консолі `Promise { <pending> }`.

Адже ми не отримуємо дані одразу. Функція **fetch** повертає **Promise** (Обіцянку) — спеціальний об'єкт, який сигналізує: "Результат буде пізніше".

А як тоді отримати відповідь від сервера?

Синтаксис async / await

Для асинхронних запитів до сервера зазвичай використовують дуже типову функцію:

```
async function getData() {
    const response = await fetch('url...');
    const data = await response.json();
    console.log(data);
}
```

Ключове слово async

Щоб використовувати асинхронний код, ми повинні попередити JavaScript. Ми ставимо слово **async** перед оголошенням асинхронної функції.

```
// Звичайна функція (виконується миттєво)
function getData() { ... }

// Асинхронна функція (може чекати)
async function getData() { ... }
```

Ключове слово await

Ось тут вступає в гру **await** (від англ. "**чекати**").

- **Його можна ставити тільки всередині async функцій.**
- **Як це працює:** Коли JS бачить **await**, він **зупиняє** виконання цієї функції на цьому рядку і чекає, поки Promise не виконається (поки не надійде відповідь від сервера).

```
// Звичайна функція (виконується миттєво)
function getData() { ... }

// Асинхронна функція (може чекати)
async function getData() { ... }
```

Чому ми пишемо await двічі?

```
const response = await fetch(url); // Пауза №1

const data = await response.json(); // Пауза №2
```

Пауза №1: fetch() Ми чекаємо, поки сервер скаже "Привіт, я отримав твій запит". Ми отримуємо об'єкт **response** (відповідь)

Пауза №2: .json() Метод **.json()** читає потік даних і перетворює ("парсить") текст у зрозумілий JavaScript-об'єкт.

Тоді в **data** ми отримуємо об’єкт з посилання на фото котика🐱

Завдання:
генератор котиків

Перший і дуже милий виклик:

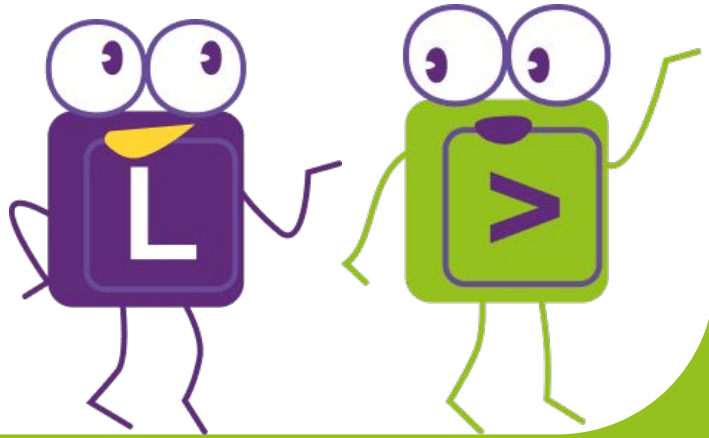
За допомогою [The Cat API](#) створити застосунок для отримання картинок милих котиків. 🐈

Спробуємо?

Робота на платформі

Перерва

Обговорення:
Нове замовлення

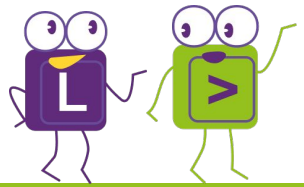


Проєкт "Кишеньковий перекладач"

Ми навчилися просто робити запити на сервер, щоб отримувати інформацію.

Тепер ми навчимося "**вести діалог**": надсилати свої дані й отримувати конкретну відповідь.

➔ **Розробимо вебзастосунок перекладача, що для перекладу використовуватиме сторонній API.**



Який API обрати? 🤔

Нам потрібен провайдер для перекладу тексту (робити власний перекладач точно зайве).

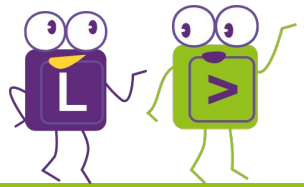
➢ **Чому б не взяти офіційний Google Translate API?**

На жаль, більшість API-сервісів мають обмеження:

➔ **Гроші**: Google Translate API платний.

➔ **Доступ**: Щоб його отримати, треба прив'язати банківську картку (навіть для тестування).

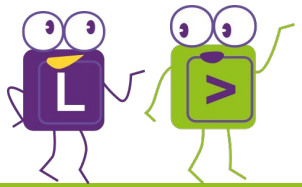
➔ **Складність**: потрібні API-ключі та система авторизації (OAuth).



Наш вибір: MyMemory API

- Це **Open Source** проєкт (відкритий код).
- Він **безплатний** для навчання.
- Він простий: **не вимагає реєстрації** та карток.

MyMemory
by translated LABS



МОДУЛЬ 5. УРОК 3.

Нова тема: Працюємо з API

Примітки для викладача

MyMemory API має ліміт запитів (безкоштовно — близько 5000 символів на день з однієї IP).

Якщо вся група сидить на одному Wi-Fi на локації, їх запити можуть відхилятися після активного тестування (як одну IP-адресу).

В такому разі можна вказати в запиті параметр de=ваш_email@gmail.com, щоб збільшити ліміт до 50 000 символів/день.

Також допускається використання сторонніх API: Google Translate, DeepL API (але потрібна реєстрація і прив’язка картки!)

Як сформувати запит?

Щоб **MyMemory** зробив переклад тексту, ми маємо надіслати йому правильне повідомлення з вхідним текстом користувача.

Зазвичай дані передають прямо в **URL** запиту:

https://api.mymemory.translated.net/get?q=Hello%20World!&langpair=en|uk-UA

Розберемо цей приклад детальніше 🧐

Анатомія URL-адреси

В API-запитах URL (посилання) — це не просто адреса. Це набір інструкцій для сервера.

Схема:
https://api.mymemory.translated.net/get?q=Text&langpair=en|uk-UA

1. **Base URL (https://api.mymemory.translated.net) –** доменне ім'я сервера, до якого ми звертаємося
2. **Endpoint (Кінцева точка)** – шлях до конкретного ресурсу або дії (**/get**).
3. **Query-параметри** – додаткові дані, які ми передаємо серверу для уточнення результату. Починаються після знаку **?.**

Query-параметри

Query-параметри (або "рядок запиту") дозволяють передавати змінні прямо в посиланні.

1. **Початок:** Завжди починаються зі знаку питання ?. Все, що до нього — адреса, все після — дані.
2. **Формат:** Працюють за принципом **key=value** (ключ=значення).
3. **Роздільник:** Якщо параметрів декілька, вони розділяються символом амперсанд &.

Приклад: **?q=Hello** (переклади слово Hello) **& langpair=en|uk** (мовна пара).

Чи потрібен API ключ?

Більшість сервісів вимагає передавати у сервісі ваш **API ключ**. Ключ API – це унікальний рядковий ідентифікатор (токен). Це наче ваш унікальний пароль до API-сервісу.

Для чого він зазвичай потрібен:

1. **Аутентифікація:** Сервер розпізнає, який саме додаток робить запит.
2. **Rate Limiting (Ліміти):** Дозволяє контролювати кількість запитів (наприклад, не більше 1000 на добу), щоб уникнути перевантаження системи.
3. **Доступ:** Відкриває доступ до платних або приватних функцій API.

Специфіка MyMemory API

MyMemory API є публічним, тому не вимагає API ключа.

Однак, для розширення лімітів (квоти) до 50 000 символів/день, вони використовують спрощену авторизацію через **Email**.

Як це працює: Ми передаємо параметр **de** (**destination email**) у рядку запиту. Сервер зчитує цей параметр і прив'язує статистику використання до вашої пошти.

Як це виглядає: `&de=myemail@gmail.com`

Додатковий трюк

URL-адреса має суворі правила. У ній не можна використовувати пробіли та деякі символи (&, ?, /), тому що вони мають службове значення.

Тому нам потрібна функція encodeURIComponent().

Це вбудована функція JavaScript, яка перетворює "небезпечні" символи на спеціальний код (відсоток + число).

```
const safeText = encodeURIComponent(text);
```

Формування фінального запиту

Щоб отримати переклад, нам потрібно об'єднати всі частини в один рядок. **Можемо зберегти емейл і мовні пари у змінні**

```
const LANGS = 'en|uk';
const EMAIL = 'f@example.com';
```

Об’єднуємо результат в один URL за допомогою шаблонних рядків:

```
const url =
`https://api.mymemory.translated.net/get?q=${safeText}&langpair=en|uk&de=${E
MAIL}`;
```

Підіб’ємо підсумки

Час виставляти логіки

До зустрічі наступного заняття!