ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ І ЗВ'ЯЗКУ

Кафедра інформаційних та комп'ютерних систем

Северин М.В.

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни «Веб-технології та веб-дизайн»

для здобувачів спеціальності 122(F3) «Комп'ютерні науки»

ОДЕСА
ДУІТЗ
2025

УДК 004.774(07)

Укладач

Северин М.В. старший викладач кафедри інформаційних та комп’ютерних систем Державного університету інтелектуальних технологій і зв’язку

Рецензенти:

Бондар Л. В. – кандидат педагогічних наук, доцент кафедри інноваційних технологій та методики викладання природничих дисциплін, Південноукраїнський національний педагогічний університет імені К. Д. Ушинського (ПНПУ), Одеса, Україна.

Панченко Б.Є., д.ф.-м.н., професор кафедри Інженерії програмного забезпечення Державного університету інтелектуальних технологій і зв'язку

Рекомендовано до друку Навчально-методичною Радою
Державного університету інтелектуальних технологій і зв’язку
(протокол №11 від 25 грудня 2025 р.)

Веб-технології та веб-дизайн : конспект лекцій [для здобувачів першого (бакалаврський) рівня вищої освіти спеціальності 122(F3) «Комп'ютерні науки»] / уклад. М. В. Северин. Одеса : ДУІТЗ, (Електр. вид. https://metod.suitt.edu.ua), 2025. 129 с.

Конспект лекцій призначений для здобувачів освіти спеціальності 122(F3) Комп’ютерні науки і може використовуватися при вивченні дисципліни «Веб-технології та веб-дизайн». Конспект лекцій спрямований на вивчення сучасних методів, мов та інструментів створення веб-орієнтованих інформаційних систем. Охоплює повний цикл розробки (Full-stack): від проєктування інтерфейсів користувача та візуального дизайну до створення серверних рішень. Особлива увага приділяється професійним стандартам верстки, типізації коду, компонентній архітектурі та безпеці веб-застосунків. Студенти опанують сучасний технологічний стек (TypeScript, React, Node.js), що дозволить їм розробляти конкурентоспроможні веб-продукти.

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій та глобальна цифровізація всіх сфер суспільного життя зумовлюють високий попит на кваліфікованих фахівців у галузі веб-розробки. Сучасний веб-застосунок перестав бути просто набором статичних сторінок, сьогодні це складна екосистема, що поєднує в собі високу інтерактивність, надійність серверної частини, безпеку даних та бездоганний користувацький досвід.

Навчальна дисципліна «Веб-технології та веб-дизайн» інтегрує фундаментальні принципи комп'ютерних наук із практичними навичками використання передових інструментів розробки. Особливістю даної програми є орієнтація на концепцію Full-stack розробки, що дозволяє здобувачам освіти отримати цілісне розуміння архітектури веб-проєктів: від дизайнерського макету до розгортання масштабованого сервера в хмарній інфраструктурі.

Актуальність курсу визначається динамічністю веб-індустрії. Використання статичних підходів минулого десятиліття вже не відповідає вимогам ринку. Тому програма базується на технологіях сучасної індустрії – мові TypeScript, бібліотеці React та серверному середовищі Node.js. Вивчення цих технологій забезпечує здобувачам освіти швидку адаптацію до реальних виробничих процесів у IT-компаніях.

Навчання побудоване на поєднанні теоретичного базису та інтенсивної практичної підготовки. Велика увага приділяється чистоті коду, типізації, тестуванню та автоматизації процесів. Здобувачі освіти вивчають веб-дизайн не як абстрактну графіку, а як функціональне проєктування інтерфейсів (UI/UX), що безпосередньо втілюється в коді.

Курс «Веб-технології та веб-дизайн» є логічним продовженням циклу дисциплін з програмування, алгоритмізації та баз даних. Він формує у здобувачів освіти професійні компетенції, необхідні для успішної професійної діяльності у веброзробці.

ТЕМА 1. ОСНОВИ UI/UX ДИЗАЙНУ: КОНЦЕПЦІЇ, ПРИНЦИПИ ВЗАЄМОДІЇ ТА ЇХ РОЛЬ У ЖИТТЄВОМУ ЦИКЛІ РОЗРОБКИ ВЕБ-ДОДАТКІВ.

Мета: Сформувати у студентів глибоке розуміння концепцій User Experience (UX) та User Interface (UI) у контексті веб-розробки. Ознайомити з базовими евристиками юзабіліті, процесом проєктування інтерфейсів та навчити аналізувати веб-продукти з точки зору зручності для користувача та ефективності для бізнесу.

1. Місце UI/UX у життєвому циклі розробки ПЗ (SDLC)

Для інженера-програміста важливо розуміти, що дизайн – це не просто «малювання картинок». У сучасній веб-розробці етап проєктування інтерфейсу (UI/UX) передує написанню коду (Frontend) і безпосередньо впливає на структуру бази даних та логіку сервера (Backend). Від якості UX залежить, наскільки складним буде флоу (user flow) і які дані потрібно збирати від користувача.

2. User Experience (UX): Досвід користувача

UX-дизайн – це аналітичний процес створення продуктів, які забезпечують релевантний, зручний та значущий досвід для користувача. У веб-розробці UX відповідає на запитання: *«Як це працює і чи вирішує це проблему користувача?»*

Основні складові UX у веб-дизайні:

- ***Інформаційна архітектура (Information Architecture, IA):*** Логічна структура сайту. Як згруповані сторінки, як побудована навігація (меню, "хлібні крихти" - breadcrumbs).
- ***Сценарії користувача (User Flows / Journey Maps):*** Покроковий шлях, який проходить користувач для досягнення мети (наприклад, від заходу на сайт до успішної оплати в кошику).
- ***Вайрфрейми (Wireframing):*** Створення чорнових, низькодеталізованих схем розташування елементів на сторінці без дизайну (лише сірі блоки та текст).
- ***Прототипування (Prototyping):*** Створення інтерактивної моделі сайту, де можна клікати на кнопки і переходити між сторінками для тестування логіки до початку програмування.

3. User Interface (UI): Інтерфейс користувача

UI-дизайн – це візуальне оформлення цифрового продукту. Це все, що користувач бачить на екрані і з чим взаємодіє. У веб-розробці UI тісно пов'язаний з HTML/CSS та Frontend-фреймворками. UI відповідає на запитання: *«Як це виглядає?»*

Основні складові UI у веб-дизайні:

- ***Візуальна ієрархія:*** Використання розміру, кольору та відступів (whitespace) для керування увагою користувача (наприклад, кнопка цільової дії H1 завжди більша за звичайний текст).
- ***Типографіка:*** Підбір шрифтів, їх розмірів (font-size), міжрядкових інтервалів (line-height), що забезпечують читабельність (Readability).
- ***Теорія кольору та контрастність:*** Кольори не лише створюють настрій, але й несуть функцію. За стандартами доступності вебу (WCAG), текст повинен мати достатній контраст відносно фону, щоб його могли читати люди з вадами зору.
- ***Дизайн-системи та UI-кіти:*** Набір стандартизованих компонентів (кнопки, інпути, модальні вікна), які повторно використовуються розробниками по всьому проєкту (як у бібліотеках Bootstrap, Material UI, Tailwind).

4. Фундаментальні евристики юзабіліті Якоба Нільсена

Для створення якісних вебінтерфейсів існують загальноприйняті правила (евристики). Ось 4 найважливіші для веб-розробників:

1. ***Видимість стану системи:*** Веб-додаток має завжди інформувати користувача про те, що відбувається (наприклад, показ анімації завантаження (spinner) під час очікування відповіді від API).
2. ***Контроль і свобода користувача:*** Користувачі часто роблять помилки. Повинен бути чіткий «аварійний вихід» (кнопка "Скасувати", "Закрити вікно", посилання на головну сторінку).
3. ***Послідовність і стандарти:*** Не варто винаходити велосипед. Якщо на 99% сайтів логотип у лівому верхньому куті веде на головну сторінку – зробіть так само. Використовуйте стандартні іконки (лупа для пошуку, кошик для покупок).
4. ***Запобігання помилкам:*** Краще попередити помилку, ніж показувати повідомлення про неї. Наприклад, якщо пароль має містити цифри, покажіть цю вимогу *до* того, як користувач натисне «Зареєструватися».

5. Практичні приклади з поясненням

Приклад 1: Пагінація проти Нескінченної прокрутки (Infinite Scroll)

- ***Проблема:*** У базі даних є 10 000 товарів/статей. Як показати їх користувачу?
- ***Рішення 1 (Пагінація - сторінки 1, 2, 3...):***
  - ***UX:*** Дає користувачу відчуття контролю. Він знає, що на 3-й сторінці бачив потрібний товар і може легко туди повернутись. Підходить для інтернет-магазинів (E-commerce).
- ***Рішення 2 (Нескінченна прокрутка):***
  - ***UX:*** Утримує користувача в стані потоку (flow), максимізує час перебування на сайті. Важко знайти конкретний елемент вдруге. Підходить для соціальних мереж (Instagram, Twitter), де контент споживається лінійно і швидко застаріває.
  - ***Висновки для розробника:*** Різний UX вимагає різної реалізації запитів до бази даних на бекенді (offset/limit проти cursor-based pagination).

Приклад 2: Валідація веб-форм (Реєстрація)

- ***Поганий UX (Синхронна перевірка):*** Користувач заповнює 5 полів форми, натискає "Відправити". Сторінка перезавантажується, і зверху з'являється червоний текст: "Пароль занадто короткий". Всі інші поля при цьому очистилися. Така логіка поведінки дратує користувачів і вони можуть покинути ресурс особливо при наявності альтернативи.
- ***Хороший UX/UI (Асинхронна Inline-валідація):***
  - *UI:* Поле введення підсвічується червоним, з'являється іконка.
  - *UX:* Користувач заповнює одне із полів. Повідомлення про помилку з'являється *одразу* (onBlur або onChange подіях у JavaScript), як тільки користувач перейшов до наступного поля, не чекаючи відправки форми на сервер. Дані не втрачаються.

Приклад 3: Візуальна ієрархія та CTA (Call to Action)

- ***Завдання:*** На лендінговій сторінці (landing page) компанії є дві дії: "Купити зараз" (основна ціль бізнесу) і "Читати документацію" (другорядна).

- ***UI/UX Рішення:*** Кнопка "Купити зараз" робиться яскравим, акцентним кольором (наприклад, синім або помаранчевим) з тінню (Primary Button). Кнопка "Читати документацію" робиться у вигляді простого тексту з підкресленням або як прозора кнопка з рамкою (Secondary/Ghost Button). Такий підхід забезпечує звернення уваги користувача на кнопку, яка є пріоритетною.

6. Контрольні запитання

1. Яке місце займають процеси UI та UX дизайну у загальному життєвому циклі розробки веб-додатку?
2. Поясніть різницю між Інформаційною архітектурою (IA) та візуальною ієрархією. Наведіть приклади.
3. Чому принцип "Видимість стану системи" (за Я. Нільсеном) є критично важливим при роботі з асинхронними запитами у веб?
4. Що таке Wireframe (вайрфрейм) і для чого він створюється до того, як дизайнер почне підбирати кольори та шрифти?
5. Як контрастність кольорів (UI) впливає на доступність веб-сайту (Web Accessibility)?
6. Практичне завдання: Згадайте будь-який веб-сайт, яким вам було вкрай незручно користуватися. Опишіть його проблему, розділивши її на проблему UX (порушена логіка, незрозуміла структура) та проблему UI (дрібний шрифт, невидимі кнопки тощо). Запропонуйте свій варіант вирішення.

ТЕМА 2. HTML5: СЕМАНТИЧНА СТРУКТУРА ДОКУМЕНТА, РОБОТА З ТЕКСТОМ, ПОСИЛАННЯМИ ТА ЗОБРАЖЕННЯМИ

Мета: Ознайомитися з мовою розмітки HTML5. Вивчити семантичні теги для побудови каркаса веб-сторінки. Навчитися правильно розмічати текстовий контент, створювати гіперпосилання та інтегрувати мультимедіа.

1. Структура HTML

1.1. Структура HTML5 документа

Кожна веб-сторінка починається з оголошення типу документа <!DOCTYPE html>, що вказує браузеру на використання стандарту HTML5.

Базовий шаблон:

```
<!DOCTYPE html>
<html lang="uk">
     <head>
          <meta charset="UTF-8">
          <title>Заголовок сторінки</title>
     </head>
     <body>
           <!-- Контент сторінки -->
     </body>
</html>
```

1.2. Семантична структура

Семантика в HTML5 означає використання тегів, які описують їхній зміст (для браузерів, пошукових роботів та екранних читачів), а не просто зовнішній вигляд.

Основні семантичні теги:

- <header> – "шапка" сайту або секції (логотип, навігація).
- <nav> – блок навігаційних посилань.
- <main> – основний, унікальний контент сторінки (використовується один раз).
- <section> – логічний розділ документа.
- <article> – самостійний, незалежний блок контенту (напр. стаття блогу).
- <aside> – бічна панель (другорядний контент).
- <footer> – "підвал" сторінки (контакти, копірайт).

1.3. Робота з текстом

Для структурування тексту використовують:

– ***Заголовки:*** від <h1> (найголовніший) до <h6>.

– ***Абзаци:*** <p>.

– ***Списки:***

  – (маркований) з елементами <li>.

  – (нумерований).

– ***Виділення:*** (важливий текст, жирний), (акцент, курсив).

1.4. Посилання та зображення

Гіперпосилання (<a>): Атрибут href вказує шлях, а target="_blank" дозволяє відкрити посилання в новій вкладці.

Перейти до Google

Зображення (<img>): Тег не має закриваючої пари. Обов'язкові атрибути:

- src – шлях до файлу.
- alt – альтернативний текст (опис зображення).

<img src="path/to/image.jpg" alt="Опис зображення для доступності">

2. Практичні приклади

Приклад 1: Семантична розмітка статті

```
<article>
    <header>
        <h2>Основи веб-дизайну</h2>
        <p>Опубліковано: 20 серпня 2025</p>
    </header>
    <section>
        <p>Веб-дизайн – це не лише візуалізація контенту, а й зручність користування...</p>
        <img src="web-design.png" alt="Схема UX дизайну">
    </section>
    <footer>
        <p>Теги: #web #design #html</p>
    </footer>
</article>
```

Приклад 2: Створення навігаційного меню

```
<nav>
    <ul>
        <li><a href="index.html">Головна</a></li>
        <li><a href="about.html">Про нас</a></li>
        <li><a href="contact.html">Контакти</a></li>
    </ul>
</nav>
```

3. Контрольні запитання

1. Чим семантичні теги (напр. <main>) відрізняються від універсальних контейнерів ( )?
2. Які атрибути є обов'язковими для тегу <img> і чому?
3. У чому різниця між тегами <ul> та ?
4. Який атрибут тегу <a> використовується для відкриття сторінки в новому вікні?
5. Чому заголовок <h1> рекомендується використовувати лише один раз на сторінці?

ТЕМА 3. ОСНОВИ CSS3: СЕЛЕКТОРИ, КАСКАДНІСТЬ, СПЕЦИФІЧНІСТЬ ТА УСПАДКУВАННЯ

Мета: Вивчити способи підключення стилів та типи селекторів. Розібратися в алгоритмі каскадності та пріоритетності стилів. Навчитися розраховувати специфічність селекторів. Зрозуміти механізм успадкування властивостей від батьківських елементів.

1. CSS

1.1. Способи підключення CSS

Існує три основні методи додавання стилів до HTML:

1. ***Зовнішні стилі (External):*** через тег <link> у head. Найкращий метод для великих проєктів.
2. ***Внутрішні стилі (Internal):*** через тег <style> всередині HTML-документа.
3. ***Вбудовані стилі (Inline):*** через атрибут style безпосередньо в тегу. Мають найвищий пріоритет, але складні в підтримці.

1.2. Базові селектори

Селектори визначають, до яких елементів будуть застосовані правила:

- ***Універсальний селектор (*):*** обирає всі елементи на сторінці.
- ***Селектор тегу (h1, p, div):*** обирає всі елементи вказаного типу.
- ***Селектор класу (.classname):*** обирає елементи з відповідним атрибутом class. Може застосовуватися до багатьох елементів.
- ***Селектор ідентифікатора (#idname):*** обирає унікальний елемент. Має бути один на сторінці.

1.3. Каскадність та специфічність

Каскадність – це механізм, який вирішує конфлікти, коли декілька правил CSS застосовуються до одного елемента.

Якщо пріоритет однаковий, перемагає те правило, що написане нижче у коді. Проте, пріоритет зазвичай визначається специфічністю:

Тип селектора	*Вага (умовні бали)*
Inline-стиль	1000
ID (#id)	100
Класи, псевдокласи, атрибути	10
Елементи (теги), псевдоелементи	1
Універсальний селектор	0

Приклад: Селектор div.container #title має специфічність 111 (1+10+100).

Слід пам’ятати, що у CSS модифікатор !important забезпечує найвищий пріоритет для конкретної властивості, перевизначаючи всі інші стандартні правила, включаючи вищу специфічність або інлайн-стилі. Він додається безпосередньо перед крапкою з комою (наприклад, color: red !important;), порушуючи природний каскад стилів, тому його використання рекомендується виключно тимчасово, наприклад для перевірки якоїсь властивості.

1.4. Успадкування (Inheritance)

Деякі властивості автоматично передаються від батьків до дітей (наприклад, color, font-family, line-height). Властивості позиціонування, фону або рамок (border, margin, padding) не успадковуються.

2. Практичні приклади

Приклад 1: Розрахунок специфічності

Розглянемо конфлікт стилів:

```
/* Специфічність: 1 */
p { color: blue; }

/* Специфічність: 10 */
.text-red { color: red; }

/* Специфічність: 110 */
#main .text-red { color: green; }
```

Елемент Текст буде зеленим, оскільки третє правило має найбільшу вагу.

Приклад 2: Використання комбінаторів

```
/* Контекстний селектор: всі p всередині div */
div p { margin-bottom: 10px; }

/* Дочірній селектор: тільки p, що є прямими нащадками div */
div > p { font-weight: bold; }
```

3. Контрольні запитання

1. Який спосіб підключення CSS є найбільш рекомендованим для сучасних веб-сайтів і чому?
2. Яка специфічність у селектора body header.top-nav ul li a?
3. Що станеться, якщо до одного елемента застосувати два однакові селектори з різними значеннями кольору (наприклад, синій і червоний)?
4. Які властивості CSS зазвичай успадковуються, а які – ні?
5. Для чого використовується ключове слово !important і чому його використання вважається поганою практикою?

ТЕМА 4. БЛОКОВА МОДЕЛЬ (BOX MODEL): MARGIN, PADDING, BORDER. ПОЗИЦІЮВАННЯ ЕЛЕМЕНТІВ

Мета: Детально вивчити складові блокової моделі CSS. Навчитися керувати внутрішніми та зовнішніми відступами, межами та розмірами елементів. Ознайомитися з різними методами позиціювання (position) для створення складних макетів. Зрозуміти властивість box-sizing та її вплив на розрахунок ширини блоку.

1. Box Model

1.1. Складові блокової моделі (Box Model)

Кожен елемент у HTML розглядається як прямокутний блок, що складається з чотирьох частин:

1. ***Content (Вміст):*** текст, зображення або інший контент всередині елемента.
2. ***Padding (Внутрішні відступи):*** простір між вмістом і рамкою. Має колір фону елемента.
3. ***Border (Рамка):*** лінія, що оточує внутрішній відступ і вміст.
4. ***Margin (Зовнішні відступи):*** простір навколо елемента (зовні за рамкою), який відділяє його від сусідніх елементів. Завжди прозорий.

1.2. Властивість box-sizing

За замовчуванням (content-box) padding та border додаються до заданої ширини width.

- ***content-box:*** Ширина = width + padding + border
- ***border-box:*** Ширина = width (падінги та рамки входять в загальний розмір).

Рекомендується використовувати box-sizing: border-box; для всіх елементів для легшого розрахунку макетів.

1.3. Позиціювання (position)

Властивість position визначає, як елемент розташовується на сторінці:

- ***static:*** (за замовчуванням) елемент знаходиться у звичайному потоці документа. Властивості top, right, bottom, left та z-index не діють.

- ***relative:*** елемент зсувається відносно свого початкового місця. Місце, яке він займав у потоці, зберігається за ним (пустота), а сам елемент може накладатись на інші.
- ***absolute:*** елемент вивертається з нормального потоку документа. Він позиціонується відносно найближчого батьківського елемента, що має будь-яке позиціювання крім static. Якщо такого немає – відносно body.
- ***f*ixed:** елемент вилучається з потоку і фіксується відносно вікна браузера (viewport). Він залишається на місці навіть при прокручуванні сторінки.
- **sticky:** елемент поводиться як relative, доки не досягне певної межі (наприклад, верхнього краю екрана), після чого «прилипає» як fixed в межах свого батьківського контейнера.

2. Практичні приклади

Приклад 1: Налаштування відступів та рамок

```
.card {
    width: 300px;
    padding: 20px;       /* Внутрішній відступ від контенту до рамки */
    border: 2px solid #333; /* Рамка товщиною 2px */
    margin: 10px auto;    /* 10px зверху/знизу, центровано по горизонталі */
    box-sizing: border-box; /* Ширина 300px включає в себе padding та border */
}
```

Приклад 2: Створення «липкої» шапки та модального вікна

```
header {
    position: sticky;
    top: 0;
    background: white;
    z-index: 100; /* Розташовується поверх іншого контенту */
}

.modal-overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
```

```
        background: rgba(0,0,0,0.5);
    }

    .modal-content {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%); /* Точне центрування по
центру екрана */
        background: #fff;
        padding: 20px;
    }
```

3. Контрольні запитання

1. Яка різниця між margin та padding? У яких випадках краще використовувати кожен з них?
2. Як зміниться реальна ширина блоку, якщо до width: 200px додати padding: 20px при значенні box-sizing: content-box?
3. Що таке «схлопування марджинів» (margin collapsing) і для яких сторін блоку воно характерне?
4. Відносно чого позиціонується елемент із значенням position: absolute?
5. Навіщо потрібна властивість z-index і з якими типами позиціювання вона працює?

ТЕМА 5. СУЧАСНА ВЕРСТКА: FLEXBOX

Мета: Опанувати концепцію гнучкої верстки (Flexible Box Layout). Вивчити термінологію: головна та поперечна осі, флекс-контейнер та флекс-елементи. Детально розібрати властивості вирівнювання по обох осях. Навчитися створювати адаптивні розмітку з Flexbox. Зрозуміти принципи керування розмірами елементів через flex-grow, flex-shrink та flex-basis.

1. Flexbox

1.1. Вступ до Flexbox

Flexbox (Flexible Box Layout Module) – це одномірна модель розкладки, яка дозволяє легко розподіляти простір між елементами в інтерфейсі та вирівнювати їх, навіть якщо їхній розмір невідомий або динамічний.

На відміну від звичайної блокової моделі, де елементи йдуть один за одним зверху вниз, Flexbox дозволяє керувати напрямком, порядком та пропорціями елементів у межах одного рядка або стовпця.

1.2. Основні поняття та осі

Для роботи з Flexbox необхідно розуміти два рівні взаємодії:

- ***Flex Container (Батько):*** елемент, до якого застосовано display: flex або display: inline-flex.
- ***Flex Items (Діти):*** всі прямі нащадки контейнера.

Система координат Flexbox:

- ***Main Axis (Головна вісь):*** вісь, вздовж якої розташовуються елементи. За замовчуванням вона горизонтальна (зліва направо).
- ***Cross Axis (Поперечна вісь):*** вісь, перпендикулярна головній. За замовчуванням вона вертикальна (згори вниз).
- ***Main Start / Main End:*** початкова та кінцева точки головної осі.
- ***Cross Start / Cross End:*** початкова та кінцева точки поперечної осі.

1.3. Властивості флекс-контейнера (Батьківські властивості)

1.3.1. display

Встановлює елемент як флекс-контейнер.

- display: flex; – блок розтягується на всю ширину батька.
- display: inline-flex; – блок займає лише необхідну ширину контенту.

1.3.2. flex-direction

Визначає напрямок головної осі:

- row (за замовчуванням): зліва направо.
- row-reverse: справа наліво.
- column: зверху вниз (головною віссю стає вертикаль).
- column-reverse: знизу вгору.

1.3.3. flex-wrap

Керує перенесенням елементів на новий рядок:

- nowrap (за замовчуванням): всі елементи намагаються втиснутися в один рядок.
- wrap: елементи переносяться на наступні рядки, якщо не вистачає місця.
- wrap-reverse: перенесення відбувається вгору.

1.3.4. justify-content

Визначає вирівнювання вздовж *головної осі*:

- flex-start: притиснуті до початку.
- flex-end: притиснуті до кінця.
- center: по центру.
- space-between: перший елемент на початку, останній у кінці, рівні проміжки між ними.
- space-around: рівні проміжки навколо кожного елемента (крайні відступи вдвічі менші за внутрішні).
- space-evenly: абсолютно однакові проміжки між усіма елементами та краями.

1.3.5. align-items

Визначає вирівнювання вздовж *поперечної осі* (для поточного рядка):

- stretch (за замовчуванням): розтягуються на всю висоту/ширину контейнера.
- flex-start: по верхньому краю.
- flex-end: по нижньому краю.
- center: по центру.
- baseline: вирівнювання за базовою лінією тексту всередині елементів.

1.3.6. align-content

Визначає відстань між рядками всередині контейнера (працює лише при наявності flex-wrap: wrap та декількох рядків):

- Аналогічні значення: flex-start, flex-end, center, space-between, space-around, stretch.

1.4. Властивості флекс-елементів (Дочірні властивості)

1.4.1. flex-grow (Коефіцієнт росту)

Визначає, яку частку вільного простору забере елемент.

- Значення 0 (дефолт): не росте.
- Якщо всі мають flex-grow: 1, вони поділять вільне місце порівну.

1.4.2. flex-shrink (Коефіцієнт стискання)

Визначає здатність елемента стискатися, коли місця в контейнері недостатньо.

- Значення 1 (дефолт): стискається порівну.
- Значення 0: елемент заборонено стискати (може вийти за межі батька).

1.4.3. flex-basis

Визначає початковий розмір елемента до розподілу вільного простору. Можна задавати в px, %, em або auto.

1.4.4. Скорочений запис flex

Рекомендується використовувати властивість flex, яка об’єднує три попередні: flex: [flex-grow] [flex-shrink] [flex-basis]; Приклад: flex: 1 1 200px;

1.4.5. order

Змінює візуальний порядок елементів без зміни HTML-коду. Початкове значення 0. Чим менше число, тим раніше стоїть елемент.

1.4.6. align-self

Дозволяє перевизначити властивість align-items для окремого конкретного елемента.

2. Практичні приклади

Приклад 1: Центрування

Раніше центрування блоку по вертикалі та горизонталі вимагало складних маніпуляцій. З Flexbox це робиться за 3 рядки:

```
.parent {
    display: flex;
    justify-content: center; /* по горизонталі */
    align-items: center;     /* по вертикалі */
    height: 100vh;           /* на весь екран */
}
```

Приклад 2: Адаптивна навігаційна панель

Створимо меню, де логотип зліва, а посилання – справа:

```
<nav class="navbar">
    <div class="logo">MyBrand</div>
    <ul class="nav-links">
        <li>Home</li>
        <li>About</li>
        <li>Contact</li>
    </ul>
</nav>

.navbar {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px 20px;
    background: #2c3e50;
    color: white;
}

.nav-links {
    display: flex;
    list-style: none;
    gap: 20px; /* відступи між флекс-елементами */
}
```

Приклад 3: Гнучка сітка карток (Grid-like Flex)

```
.container {
    display: flex;
```

```
    flex-wrap: wrap;
    gap: 15px;
}

.item {
    flex: 1 1 200px; /* рости, стискатися, основа 200px */
    background: #f4f4f4;
    padding: 20px;
    border: 1px solid #ddd;
}
```

Ця конструкція автоматично адаптується: якщо на екрані мало місця, картки переходять на нові рядки.

3. Аналіз розрахунку розмірів

Розглянемо приклад, коли використовується flex: 1 1 100px для трьох елементів у контейнері шириною 600px:

1. Сумарний flex-basis = 300px.
2. Вільний простір = 600px - 300px = 300px.
3. Оскільки у всіх flex-grow: 1, вільні 300px діляться на 3 частини по 100px.
4. Фінальний розмір кожного елемента = 100px (basis) + 100px (growth) = 200px.

4. Контрольні запитання

1. Поясніть різницю між justify-content та align-items. На які осі вони впливають за замовчуванням?
2. Як змінити напрямок осей у контейнері? Яка властивість за це відповідає?
3. Що станеться з елементами, якщо задати flex-wrap: nowrap, а їх сумарна ширина перевищить ширину контейнера?
4. Опишіть призначення властивості flex-basis. Чим вона відрізняється від width?
5. Навіщо використовувати gap замість margin для створення проміжків між флекс-елементами?
6. Які переваги дає Flexbox у порівнянні з використанням float для побудови макетів?

ТЕМА 6. CSS GRID LAYOUT ТА СТВОРЕННЯ БАГАТОКОЛОНКОВИХ МАКЕТІВ

Мета: Ознайомитися з концепцією двомірного позиціювання (CSS Grid). Вивчити термінологію: Grid-контейнер, лінії, смуги, комірки та області. Навчитися створювати складні багатоколонкові макети за допомогою grid-template-areas. Опанувати гнучкі одиниці вимірювання (fr) та функції автоматизації (repeat, minmax). Навчитися комбінувати Flexbox та Grid для професійної верстки.

1. CSS Grid

1.1. Вступ до CSS Grid Layout

CSS Grid Layout – це система розкладки, яка, на відміну від Flexbox (що є одномірним), працює у двох вимірах одночасно: по горизонталі (рядки) та вертикалі (стовпці). Це дозволяє розробникам створювати складні макети без використання вкладених контейнерів.

1.2. Основні поняття Grid

1. ***Grid Container:*** елемент, до якого застосовано display: grid або display: inline-grid.
2. ***Grid Item:*** прямий нащадок контейнера.
3. ***Grid Line:*** розділові лінії, що утворюють сітку. Вони можуть бути вертикальними (column lines) та горизонтальними (row lines).
4. ***Grid Track (Смуга):*** простір між двома сусідніми лініями (це або весь стовпець, або весь рядок).
5. ***Grid Cell (Комірка):*** найменша одиниця сітки, простір між чотирма перехресними лініями.
6. ***Grid Area (Область):*** прямокутник, що складається з будь-якої кількості комірок.

1.3. Властивості Grid-контейнера

1.3.1. Визначення структури

- grid-template-columns: задає кількість та розмір стовпців.
- grid-template-rows: задає кількість та розмір рядків.
- Одиниця fr (Fractional unit): представляє частку вільного простору в контейнері.

1.3.2. Функції та ключові слова

- repeat(count, size): дозволяє уникнути дублювання значень. Наприклад, repeat(3, 1fr) замінює 1fr 1fr 1fr.
- minmax(min, max): задає діапазон розміру – від мінімально необхідного до максимально можливого.
- auto-fill / auto-fit: автоматично заповнює рядок максимальною кількістю стовпців залежно від ширини екрана.

1.3.3. Відступи

- column-gap: відстань між стовпцями.
- row-gap: відстань між рядками.
- gap (або grid-gap): скорочення для обох типів відступів.

1.3.4. Вирівнювання в Grid

- justify-items: вирівнює елементи всередині їхніх комірок вздовж осі рядків (горизонтально).
- align-items: вирівнює елементи вздовж осі стовпців (вертикально).
- place-items: скорочення для обох вищезгаданих властивостей.
- justify-content / align-content: вирівнює саму сітку (всі треки) відносно контейнера.

1.4. Властивості Grid-елементів

1.4.1. Позиціювання за лініями

- grid-column-start / grid-column-end: вказують, з якої по яку вертикальну лінію розтягується елемент.
- grid-row-start / grid-row-end: те саме для горизонтальних ліній.
- Скорочення span: дозволяє вказати кількість комірок, на які треба розтягнути елемент (наприклад, grid-column: span 2).

1.4.2. Робота з іменованими областями

Властивість grid-template-areas дозволяє візуально «намалювати» макет сторінки прямо в CSS:

```
.container {
  display: grid;
  grid-template-areas:
```

```
    "header header header"
    "sidebar main main"
    "footer footer footer";
}
```

Елементи прив'язуються до цих областей за допомогою grid-area: header;.

2. Практичні приклади

Приклад 1: Триколонкова сітка

```
.grid-container {
    display: grid;
    grid-template-columns: 200px 1fr 150px;
    gap: 20px;
}
```

У представленому макеті перша та третя колонки мають фіксовану ширину, а середня займає весь простір, що залишився.

Приклад 2: Складний макет через іменовані області

```
.layout {
    display: grid;
    grid-template-columns: 250px 1fr;
    grid-template-rows: auto 1fr auto;
    grid-template-areas:
        "nav nav"
        "aside article"
        "footer footer";
    min-height: 100vh;
}

header { grid-area: nav; }
aside { grid-area: aside; }
main { grid-area: article; }
footer { grid-area: footer; }
```

Приклад 3: Адаптивна сітка без медіа-запитів

Використання auto-fit дозволяє автоматично переносити картки на новий рядок при зменшенні ширини вікна:

```
.cards-grid {
    display: grid;
```

```
        grid-template-columns:    repeat(auto-fit,    minmax(250px,
1fr));
        gap: 1rem;
    }
```

3. Порівняння Flexbox та Grid Layout

Характеристика	Flexbox	CSS Grid
Вимірність	Одномірний (рядок АБО стовпець)	Двомірний (рядок ТА стовпець одночасно)
Методологія	Контент керує формою (Content-first)	Макет керує контентом (Layout-first)
Вирівнювання	Потужне вздовж головної осі	Потужне в межах комірок та всього контейнера
Накладання	Через від'ємні margin або position: absolute	Природне (елементи можуть займати однакові комірки)

4. Контрольні запитання

1. Поясніть різницю між auto-fill та auto-fit. В яких ситуаціях краще застосовувати кожне з них?
2. Як працює одиниця вимірювання fr? Як вона поєднується з фіксованими значеннями (наприклад, px)?
3. Що станеться, якщо елементу задати grid-column: 1 / 4, але в контейнері визначено лише 2 стовпці?
4. Навіщо використовувати grid-template-areas замість позиціювання за номерами ліній? Наведіть приклади переваг для підтримки коду.
5. Як за допомогою Grid створити ефект накладання одного елемента на інший без використання position: absolute?
6. Які властивості Grid допомагають зробити макет адаптивним без використання великої кількості @media запитів?
7. Чи можна використовувати Flexbox всередині Grid-комірки? Обґрунтуйте доцільність такої комбінації.

ТЕМА 7. ВСТУП ДО JAVASCRIPT: ЗМІННІ, ТИПИ ДАНИХ ТА ОПЕРАТОРИ

Мета: Ознайомитися з призначенням та роллю JavaScript у сучасній веб-розробці. Вивчити способи оголошення змінних та різницю між let і const. Розібрати систему типів даних у JS (примітиви та об'єкти). Опанувати використання арифметичних, логічних операторів та операторів порівняння. Навчитися писати перші скрипти та виводити дані в консоль браузера.

1. Теоретичні відомості

JavaScript (JS) – це високорівнева мова програмування, яка додає інтерактивність веб-сторінкам. Якщо HTML відповідає за структуру, а CSS – за зовнішній вигляд, то JavaScript керує поведінкою елементів.

Ключові особливості:

- ***Мультиплатформенність:*** працює у всіх сучасних браузерах.
- ***Динамічна типізація:*** типи даних визначаються автоматично під час виконання коду.
- ***Інтерпретованість:*** код виконується рядок за рядком без попередньої компіляції.

2. Змінні та константи

Змінна – це "контейнер" для зберігання даних. У сучасному стандарті використовуються два основні способи оголошення let і const.

2.1. let – змінні, що можуть змінюватися

Використовується, коли значення змінної планується перезаписувати пізніше.

```
let score = 10;      // Оголошення та ініціалізація змінної
console.log(score); // Виведе 10

score = 15;          // Зміна значення
console.log(score); // Виведе 15
```

2.2. const – константи

Використовується для значень, які не повинні змінюватися протягом виконання програми. Спроба змінити значення const призведе до помилки.

```
const pi = 3.14159;
```

```
const birthYear = 1995;

// birthYear = 1996; // Помилка: Assignment to constant variable.
```

2.3. Чому не варто використовувати var?

Ключове слово var використовувалося в старих версіях JS. Воно має "проблемну" область видимості (function scope замість block scope) та підтримує "підняття" (hoisting), що часто призводить до помилок, які важко відстежити.

3. Типи даних у JavaScript

JavaScript має 8 основних типів даних, які поділяються на примітивні та складні (об'єкти).

3.1. Number (Число)

Представляє як цілі числа, так і числа з плаваючою крапкою. Також включає спеціальні значення: Infinity (математичну нескінченність), -Infinity та NaN (Not a Number).

let age = 25;
let price = 19.99;
console.log(10 / 0); // Infinity
console.log("текст" / 2); // NaN

3.2. String (Рядок)

Текстові дані. Можуть бути взяті в одинарні ('), подвійні (") або зворотні (`) лапки. Зворотні лапки дозволяють вставляти змінні через ${}.

let userName = "Микола";
let greeting = `Привіт, ${userName}!`; // *Шаблонний рядок*
console.log(greeting); // *"Привіт, Микола!"*

3.3. Boolean (Логічний тип)

Має лише два значення: true (істина) або false (хибне значення).

let isLogged = true;
let hasPermission = false;

3.4. Null

Спеціальне значення, яке вказує на "порожнечу" або "ніщо". Воно повинно бути присвоєно явно.

let currentTask = null; // *Поки що завдання немає*

3.5. Undefined

Тип, який автоматично отримує змінна, якій не було присвоєно жодного значення.

let x;
console.log(x); // *undefined*

3.6. BigInt та Symbol

- ***BigInt*** використовується для роботи з дуже великими числами (більше ніж $2^{53} - 1$).
- ***Symbol*** використовується для створення унікальних ідентифікаторів в об'єктах.

3.7. Object (Об'єкт)

Складний тип даних, що дозволяє зберігати колекції даних у форматі "ключ: значення".

```
const user = {
    name: "Марія",
    age: 21,
    isAdmin: true
};
console.log(user.name); // Мар ія
```

4. Оператори в JavaScript

4.1. Арифметичні оператори

- + – додавання (також конкатенація рядків).
- - – віднімання.
- * – множення.
- / – ділення.
- % – залишок від ділення.
- ** – піднесення до степеня.

```
let a = 10;
let b = 3;
```

```
console.log(a % b); // 1 (залишок від 10/3)
console.log(a ** 2); // 100
```

4.2. Оператори присвоєння

- = – базове присвоєння.
- +=, -=, *=, /= – скорочені оператори.

```
let x = 5;
x += 10; // те саме, що x = x + 10
```

4.3. Оператори порівняння

- > , < , >= , <= – базові порівняння.
- == – нестрога рівність (призводить типи до одного виду).
- === – строга рівність (порівнює і значення, і тип). *Рекомендовано до використання.*
- != та !== – нерівність.

```
console.log(5 == "5");  // true (нестрога рівність)
console.log(5 === "5"); // false (різні типи: number і string)
```

4.4. Логічні оператори

- && – ЛОГІЧНЕ І (повертає true, якщо обидві умови істинні).
- || – ЛОГІЧНЕ АБО (повертає true, якщо хоча б одна умова істинна).
- ! – ЛОГІЧНЕ НЕ (інвертує значення).

5. Практичне застосування

Взаємодія з користувачем

Для виводу та введення даних на початкових етапах використовуються такі методи:

1. console.log() – вивід у консоль розробника.
2. alert() – вивід повідомлення у модальне вікно.
3. prompt() – отримання текстових даних від користувача.
4. confirm() – отримання логічної відповіді (Так/Ні).

Приклад програми:

```
const userName = prompt("Як вас звати?");
const currentYear = 2024;
let birthYear = prompt("Якого року ви народилися?");
```

```
// Перетворюємо рядок у число
birthYear = Number(birthYear);

let age = currentYear - birthYear;
alert(`Вітаю, ${userName}! Вам приблизно ${age} років.`);
```

6. Контрольні запитання

1. У чому принципова різниця між let та const?
2. Що станеться, якщо спробувати додати число до рядка (наприклад, 5 + "5")?
3. Які типи даних у JavaScript вважаються примітивними?
4. Чому варто використовувати оператор === замість ==?
5. Як працює оператор залишку від ділення %? Наведіть приклад його використання.
6. Що таке NaN і за яких умов воно з'являється?
7. Для чого потрібні зворотні лапки (Backticks) при створенні рядків?
8. Як перетворити рядок "100" на число за допомогою JavaScript?
9. Яке значення матиме змінна, якщо її оголосити, але не присвоїти їй значення?
10. Поясніть пріоритет операторів: що виконається першим у виразі 2 + 2 * 2?

ТЕМА 8. КЕРУЮЧІ КОНСТРУКЦІЇ: УМОВНІ ПЕРЕХОДИ ТА ЦИКЛИ

Мета: Ознайомитися з принципами розгалуження алгоритмів за допомогою if, else та switch. Вивчити синтаксис та особливості роботи циклів for та while. Опанувати сучасний метод перебору масивів forEach. Навчитися керувати ітераціями за допомогою операторів break та continue. Створювати складні логічні конструкції для обробки даних.

1. Умовні переходи

Умовні конструкції дозволяють програмі приймати рішення та виконувати різні блоки коду залежно від певних обставин.

1.1. Оператор if...else

Це базова конструкція для перевірки умови. Умова всередині if завжди приводиться до логічного типу (Boolean).

```
let temp = 25;

if (temp > 30) {
    console.log("На вулиці спекотно");
} else if (temp > 15) {
    console.log("На вулиці комфортно");
} else {
    console.log("На вулиці холодно");
}
```

1.2. Тернарний оператор

Коротка форма запису if...else, яка використовується для присвоєння значення залежно від умови. *Синтаксис:* умова ? значення_якщо_true : значення_якщо_false

```
let age = 18;
let access = (age >= 18) ? "Дозволено" : "Заборонено";
console.log(access);
```

1.3. Конструкція switch

Використовується, коли потрібно порівняти одну змінну з багатьма конкретними значеннями. Важливо використовувати break, щоб зупинити виконання наступних варіантів.

```
let day = 3;
```

```
switch (day) {
    case 1:
        console.log("Понеділок");
        break;
    case 2:
        console.log("Вівторок");
        break;
    case 3:
        console.log("Середа");
        break;
    default:
        console.log("Інший день");
}
```

2. Цикли: Повторення дій

Цикли дозволяють виконувати один і той самий блок коду багато разів, поки виконується певна умова.

2.1. Цикл while

Виконує код, доки умова істинна. Кількість ітерацій може бути невідомою заздалегідь.

```
let count = 1;

while (count <= 5) {
    console.log(`Ітерація №${count}`);
    count++;  // Важливо змінювати умову, щоб не створити нескінченний цикл
}
```

2.2. Цикл for

Найбільш вживаний цикл. Він об'єднує ініціалізацію, умову та крок в одному рядку.

```
for (let i = 0; i < 5; i++) {
    console.log(`Крок: ${i}`);
}
```

Розбір структури:

1. let i = 0 – початок (виконується один раз).
2. i < 5 – умова (перевіряється на початку кожної ітерації циклу).
3. i++ – крок (виконується після кожного тіла циклу).

3. Робота з масивами та метод forEach

У сучасному JavaScript для перебору елементів масиву часто використовують вбудовані методи замість класичного циклу for.

3.1. Синтаксис forEach

Цей метод викликає функцію для кожного елемента масиву.

```
const fruits = ["Яблуко", "Банан", "Апельсин"];

fruits.forEach(function(item, index) {
    console.log(`${index + 1}: ${item}`);
});
```

Переваги forEach:

- Код виглядає більш лаконічним та читабельним.
- Менше ризиків помилитися з індексами або виходом за межі масиву.
- Підтримує анонімні та стрілкові функції.

4. Керування циклами: break та continue

Іноді виникає потреба перервати цикл раніше або пропустити певну ітерацію.

1. ***break*** – повністю зупиняє цикл і виходить з нього.
2. ***continue*** – зупиняє поточну ітерацію і переходить до наступної.

```
for (let i = 1; i <= 10; i++) {
    if (i === 5) continue; // Пропустити число 5
    if (i === 8) break;    // Зупинити цикл на числі 8
    console.log(i);
}
// Виведе: 1, 2, 3, 4, 6, 7
```

5. Практичні приклади

Приклад 1: Перевірка паролю (цикл while)

```
const correctPassword = "admin";
let userPass = "";

while (userPass !== correctPassword) {
    userPass = prompt("Введіть пароль:");
    if (userPass !== correctPassword) {
        alert("Невірно, спробуйте ще раз!");
```

```
    }
}
alert("Доступ надано!");
```

Приклад 2: Пошук парних чисел (цикл for)

```
let limit = 20;
console.log("Парні числа:");

for (let i = 2; i <= limit; i += 2) {
    console.log(i);
}
```

Приклад 3: Підрахунок суми елементів (метод forEach)

```
const prices = [100, 250, 50, 400];
let total = 0;

prices.forEach(price => {
    total += price;
});

console.log(`Загальна сума: ${total} грн`);
```

6. Контрольні запитання

1. Поясніть різницю між циклами while та do...while.
2. Коли доцільніше використовувати switch замість ланцюжка if...else if?
3. Яка роль ключового слова break у конструкції switch? Що станеться, якщо його не вказати?
4. Опишіть три складові частини заголовка циклу for.
5. У чому перевага методу forEach над класичним циклом for при роботі з масивами?
6. Що робить оператор continue? Наведіть приклад ситуації, де він корисний.
7. Як створити нескінченний цикл і чому це може бути небезпечно?
8. Як працює тернарний оператор? Перепишіть простий if...else у тернарну форму.

ТЕМА 9. ФУНКЦІЇ: FUNCTION DECLARATION, EXPRESSION TA ARROW FUNCTIONS. ОБЛАСТЬ ВИДИМОСТІ

Мета: Розгляд функції як фундаментального елементу програми. Вивчити три основні способи оголошення функцій та їхні відмінності. Ознайомитися з механізмом передачі параметрів та повернення значень. Зрозуміти рівні області видимості (глобальну, функціональну, блочну). Вивчити особливості стрілкових функцій та поняття контексту this.

1. Основи функцій

Функція – це підпрограма, блок коду, який можна викликати багаторазово для виконання певної дії. Функції дозволяють уникати дублювання коду та структурувати логіку.

1.1. Параметри та аргументи

- ***Параметри*** – це змінні, перелічені в круглих дужках при оголошенні функції.
- ***Аргументи*** – це реальні значення, які передаються функції під час її виклику.

```
function sayHi(name) { // name – параметр
    console.log(`Привіт, ${name}!`);
}

sayHi("Іван"); // "Іван" – аргумент
```

1.2. Повернення значення (return)

Функція може повертати результат своєї роботи назад у код, що її викликав. Після виконання return функція негайно припиняє роботу.

2. Способи оголошення функцій

У JavaScript існує кілька способів створення функцій, кожен з яких має свої особливості.

2.1. Function Declaration (Оголошення функції)

Класичний спосіб. Головна особливість – Hoisting (підняття). Hoisting (підняття/спливання) в JavaScript – це механізм, при якому оголошення змінних та функцій переміщуються (вспливають) у початок їхньої області

видимості перед виконанням коду. Це дозволяє використовувати функції або змінні до того, як вони фізично визначені у коді. Розглянемо приклад виклику функції до моменту її опису у коді.

```
printHello(); // Працює завдяки Hoisting

function printHello() {
    console.log("Hello World");
}
```

2.2. Function Expression (Функціональний вираз)

Функція створюється як частина виразу (наприклад, присвоюється змінній). Вона не "піднімається", тому її можна викликати лише після оголошення.

```
const sum = function(a, b) {
    return a + b;
};

console.log(sum(5, 10)); // 15
```

2.3. Arrow Functions (Стрілкові функції)

З'явилися в ES6. Мають дуже лаконічний синтаксис.

```
// Повна форма
const multiply = (a, b) => {
    return a * b;
};

// Скорочена форма (якщо лише один вираз)
const square = n => n * n;
```

Особливості стрілкових функцій:

- Не мають власного this (беруть його із зовнішнього середовища).
- Не мають об'єкта arguments.
- Не можуть бути використані як конструктори.

3. Область видимості (Scope)

Область видимості визначає, де саме змінні доступні для використання.

3.1. Глобальна область видимості

Змінні, оголошені поза будь-якими функціями чи блоками, доступні всюди в коді. Використання великої кількості глобальних змінних вважається поганою практикою.

3.2. Функціональна область видимості

Змінні, оголошені всередині функції (навіть через var), доступні лише всередині цієї функції.

3.3. Блочна область видимості (ES6)

Змінні let та const обмежені фігурними дужками { }, у яких вони оголошені (цикли, умовні оператори).

```
if (true) {
    let blockVar = "Я всередині блоку";
    console.log(blockVar); // Працює
}
// console.log(blockVar); // Помилка: blockVar is not defined
```

4. Параметри за замовчуванням та замикання

4.1. Значення за замовчуванням

Ми можемо вказати значення параметра на випадок, якщо аргумент не буде передано.

```
function showMessage(from, text = "текст не додано") {
    console.log(`${from}: ${text}`);
}
showMessage("Микола"); // Микола: текст не додано
```

4.2. Замикання (Closures)

Це здатність функції запам'ятовувати своє зовнішнє оточення навіть після того, як зовнішня функція завершила виконання.

Тобто внутрішня функція має доступ до змінних батьківської функції.

```
function createCounter() {
    let count = 0; // змінна з зовнішньої області
    return function () {
        count++;
        return count;
    };
}
```

```
const counter = createCounter();

console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

Розглянемо роботу функції `createCounter`:

- createCounter() виконується.
- Вона повертає внутрішню функцію.
- Змінна count не зникає, бо внутрішня функція має до неї доступ.
- Кожен виклик counter() змінює ту саму змінну count.

Замикання часто використовуються для: інкапсуляції (приховування даних), створення лічильників, фабрик функцій, роботи з асинхронним кодом.

5. Практичні приклади

Приклад 1: Калькулятор площі (Function Declaration)

```
function getRectArea(width, height) {
    if (isNaN(width) || isNaN(height)) {
        return "Помилка: введіть числа";
    }
    return width * height;
}
console.log(getRectArea(10, 20)); // 200
```

Приклад 2: Фільтрація масиву (Arrow Function)

```
const numbers = [1, 5, 8, 12, 15, 20];
const bigNumbers = numbers.filter(n => n > 10);

console.log(bigNumbers); // [12, 15, 20]
```

Приклад 3: Перевірка на повноліття (Тернарний оператор + Функція)

```
const checkAge = age => age >= 18 ? true : confirm('Батьки дозволили?');

if (checkAge(16)) {
    console.log("Доступ дозволено");
}
```

6. Контрольні запитання

1. Поясніть різницю між Function Declaration та Function Expression.
2. Що таке "Hoisting" (підняття) стосовно функцій?
3. Які обмеження мають стрілкові функції порівняно зі звичайними?
4. Що станеться, якщо функція не містить оператора return, але ми намагаємося отримати її результат?
5. Чим відрізняється глобальна область видимості від локальної?
6. Як працюють параметри за замовчуванням?
7. Чи можна змінити глобальну змінну всередині функції? Як це вплине на код?
8. Що таке анонімна функція? Наведіть приклад її використання.
9. У яких випадках у стрілковій функції можна не писати фігурні дужки та слово return?
10. Чому змінні let та const вважаються безпечнішими за var з точки зору області видимості?

ТЕМА 10. РОБОТА З DOM: ПОШУК ЕЛЕМЕНТІВ, ЗМІНА СТИЛІВ ТА АТРИБУТІВ, МАНІПУЛЯЦІЯ СТРУКТУРОЮ

Мета: Зрозуміти концепцію DOM (Document Object Model) як об'єктного представлення HTML-документа. Вивчити сучасні методи пошуку елементів на сторінці. Навчитися динамічно змінювати вміст, атрибути та CSS-стилі елементів. Опанувати методи створення, видалення та переміщення вузлів у структурі документа. Розібрати практичні сценарії живої взаємодії з користувачем через інтерфейс.

1. DOM

DOM (Document Object Model) – це програмний інтерфейс (API) для HTML та XML документів. Коли браузер завантажує сторінку, він створює деревоподібну модель, де кожен тег, атрибут або текст є окремим об'єктом, доступним для маніпуляцій через JavaScript.

- window – глобальний об'єкт браузера.
- document – головна точка входу в DOM, що представляє саму сторінку.

2. Пошук елементів у DOM

Для того, щоб щось змінити, спочатку потрібно знайти відповідний елемент.

2.1. Старі методи (класичні)

- document.getElementById('id') – пошук за унікальним ідентифікатором.
- document.getElementsByTagName('tag') – повертає "живу" колекцію всіх елементів (тегів) з вказаним іменем тегу.
- document.getElementsByClassName('class') – повертає "живу" колекцію всіх елементів з вказаним класом.

Live Collections ("живі" колекції) – це DOM-колекції, які автоматично оновлюються, якщо структура документа змінюється. Тобто якщо ви додасте або видалите елемент у DOM, така колекція зміниться без повторного запиту. Live Collections повертаються методами getElementsBy.

2.2. Сучасні методи (рекомендовані)

Використовують синтаксис CSS-селекторів, що робить їх універсальними.

– document.querySelector('.my-class') – повертає перший знайдений елемент.
– document.querySelectorAll('ul > li') – повертає статичну колекцію всіх знайдених елементів (NodeList).
– const mainTitle = document.querySelector('#main-title');
– const buttons = document.querySelectorAll('.btn-submit');

3. Зміна вмісту та атрибутів

3.1. Текст та HTML

- textContent –дозволяє отримати або змінити суто текстовий вміст (безпечно).
- innerHTML –дозволяє працювати з HTML-розміткою всередині елемента (використовувати обережно через ризики XSS).

```
const box = document.querySelector('.info-box');
box.textContent = "Просто текст";
box.innerHTML = "<strong>Жирний текст</strong>";
```

3.2. Атрибути

JavaScript дозволяє керувати будь-якими атрибутами тегів:

- element.getAttribute('name') – отримати значення.
- element.setAttribute('name', 'value') – встановити значення.
- element.removeAttribute('name') – видалити.
- Прямий доступ: element.id, element.src, element.href, element.value.

4. Керування стилями та класами

4.1. Властивість style (Inline-стилі)

Дозволяє змінювати стилі безпосередньо в атрибуті style елемента. Назви властивостей пишуться в стилі *camelCase*.

```
const el = document.querySelector('.card');
el.style.backgroundColor = 'blue';
el.style.marginTop = '20px';
```

4.2. Робота з класами (classList) – найкраща практика

Замість прямої зміни стилів краще додавати або видаляти заздалегідь підготовлені CSS-класи.

- element.classList.add('active') – додати клас.
- element.classList.remove('hidden') – видалити клас.

- element.classList.toggle('dark-mode') – перемкнути (якщо є – видалити, якщо немає – додати).
- element.classList.contains('active') – перевірити наявність.

5. Маніпуляція структурою документа

За допомогою JavaScript можна створювати нові елементи та інтегрувати їх у структуру DOM під час виконання програми.

5.1. Створення елементів

```
const newDiv = document.createElement('div');
newDiv.className = 'alert alert-success';
newDiv.textContent = 'Дані успішно збережено!';
```

5.2. Додавання та видалення

- parent.appendChild(node) – додає елемент в кінець батьківського блоку.
- parent.prepend(node) – додає на початок.
- element.before(node) / element.after(node) – вставляє до або після елемента.
- element.remove() – повністю видаляє елемент з DOM.

6. Практичні приклади

Приклад 1: Динамічний список завдань

```
function addTask(text) {
    const list = document.querySelector('#task-list');
    const li = document.createElement('li');

    li.textContent = text;
    li.style.cursor = 'pointer';

    // Видалення при кліку (приклад на майбутнє)
    li.onclick = () => li.remove();

    list.appendChild(li);
}
```

Приклад 2: Перемикач нічної теми

```
function toggleTheme() {
```

```
    const body = document.body;
    body.classList.toggle('dark-theme');

    const btn = document.querySelector('#theme-btn');
    if (body.classList.contains('dark-theme')) {
        btn.textContent = 'Увімкнути світлу тему';
    } else {
        btn.textContent = 'Увімкнути темну тему';
    }
}
```

7. Контрольні запитання

1. Що таке DOM-дерево і як воно пов'язане з HTML-кодом?
2. Чим відрізняється querySelector від getElementById?
3. Чому властивість textContent безпечніша за innerHTML?
4. Як змінити колір фону елемента за допомогою властивості style?
5. Які переваги використання classList.add() порівняно з прямим редагуванням стилів?
6. Опишіть процес створення нового елемента та додавання його на сторінку.
7. Як отримати значення, яке користувач ввів у текстове поле input?
8. Що таке "живі" колекції (Live Collections) і які методи їх повертають?
9. Як видалити певний HTML-елемент зі сторінки?
10. Для чого використовується об'єкт document в коді JavaScript?

ТЕМА 11. ОБРОБКА ПОДІЙ: CLICK, INPUT, SUBMIT, BUBBLING & CAPTURING

Мета: Ознайомитися з концепцією подій як основного механізму інтерактивності. Вивчити методи призначення обробників подій (addEventListener). Розібрати роботу з найпоширенішими типами подій: click, input, submit. Зрозуміти фази проходження події: занурення (capturing) та спливання (bubbling). Навчитися використовувати об'єкт події (event) для отримання додаткової інформації та скасування стандартної поведінки.

1. Основи подій

Подія – це сигнал від браузера про те, що на сторінці щось відбулося (користувач натиснув кнопку, сторінка завантажилася, в поле введення було внесено текст тощо).

1.1. Способи призначення обробників

1. ***Атрибут HTML (не рекомендовано):*** <button onclick="alert('Hi')">.
2. ***Властивість DOM-об'єкта:*** element.onclick = function() { ... }.
3. ***Метод addEventListener (рекомендовано):*** дозволяє призначати кілька обробників на одну подію та гнучко ними керувати.

```
const btn = document.querySelector('#myBtn');

btn.addEventListener('click', () => {
    console.log('Кнопку натиснуто!');
});
```

2. Поширені типи подій

2.1. Подія click

Виникає при натисканні лівою кнопкою миші на елементі.

```
const box = document.querySelector('.box');
box.addEventListener('click', function() {
    this.style.backgroundColor = 'red';
});
```

2.2. Подія input

Спрацьовує щоразу, коли змінюється значення в текстовому полі (<input>, <textarea>). Це краще за change, бо працює миттєво при кожному натисканні клавіші.

```
const searchInput = document.querySelector('#search');
searchInput.addEventListener('input', (event) => {
    console.log(`Поточний текст: ${event.target.value}`);
});
```

2.3. Подія submit

Виникає при спробі відправки форми. Це ключова подія для валідації даних перед відправкою на сервер.

```
const loginForm = document.querySelector('#login-form');
loginForm.addEventListener('submit', (event) => {
    event.preventDefault();   // Зупиняє перезавантаження сторінки
    console.log('Форма оброблена скриптом');
});
```

3. Об'єкт події (event)

Коли спрацьовує подія, браузер передає в обробник об'єкт події, який містить корисні дані:

- event.type –тип події (наприклад, "click").
- event.target –елемент, на якому безпосередньо виникла подія.
- event.currentTarget –елемент, на якому спрацював обробник (враховуючи спливання).
- event.preventDefault() –скасовує дію браузера за замовчуванням (наприклад, перехід за посиланням).
- event.stopPropagation() –зупиняє подальше поширення події по дереву DOM.

4. Спливання та занурення (Bubbling & Capturing)

Події в DOM проходять через три фази:

1. ***Capturing (Занурення):*** подія йде від window вниз до цільового елемента.
2. ***Target (Ціль):*** подія досягла цільового елемента.
3. ***Bubbling (Спливання):*** подія піднімається вгору від цільового елемента до window.

За замовчуванням addEventListener працює на фазі спливання.

```
// Якщо натиснути на вкладений елемент, спочатку спрацює він,
// а потім подія "спливе" до батьківського елемента.
document.querySelector('div').addEventListener('click', () => {
    console.log('Клік на DIV (батько)');
});

document.querySelector('button').addEventListener('click', (e) => {
    console.log('Клік на Кнопку (дитина)');
    //    e.stopPropagation();    //    Якщо    розкоментувати, батьківський елемент не дізнається про клік
});
```

5. Практичні приклади

Приклад 1: Делегування подій

Замість того, щоб ставити обробник на кожен елемент списку, ми ставимо один обробник на батька.

```
const list = document.querySelector('#myList');

list.addEventListener('click', (event) => {
    if (event.target.tagName === 'LI') {
        event.target.classList.toggle('completed');
    }
});
```

Приклад 2: Відстеження координат миші

```
document.addEventListener('mousemove', (e) => {
    const info = document.querySelector('#coords');
    if (info) {
        info.textContent = `X: ${e.clientX}, Y: ${e.clientY}`;
    }
});
```

6. Контрольні запитання

1. Які переваги використання addEventListener порівняно з атрибутом onclick?
2. Що таке event.preventDefault() і в яких випадках він необхідний?
3. Поясніть різницю між event.target та event.currentTarget.

4. Опишіть механізм спливання подій (bubbling).
5. Як зупинити спливання події до батьківських елементів?
6. Чому для валідації форми краще використовувати подію submit на тегу <form>, а не click на кнопці?
7. Що таке делегування подій і в чому його вигода для продуктивності?
8. Як працює фаза занурення (capturing) і як її активувати в addEventListener?
9. Яка подія спрацьовує при кожній зміні вмісту текстового поля в реальному часі?
10. Як видалити раніше призначений обробник події?

ТЕМА 12. АСИНХРОННІСТЬ У JS: CALLBACKS, PROMISES, ASYNC/AWAIT. РОБОТА З FETCH API

Мета: Вивчити механізми обробки асинхронних операцій у JavaScript; зрозуміти принципи роботи Event Loop; навчитися використовувати проміси та синтаксис async/await для написання чистого коду; опанувати виконання мережевих запитів за допомогою Fetch API.

1. Асинхронність

1.1. Природа асинхронності та Event Loop

JavaScript є однопотоковою мовою програмування. Це означає, що він має один стек викликів (Call Stack) і може виконувати лише одну операцію одночасно. Однак веб-застосунки часто потребують виконання тривалих операцій (запити до сервера, читання файлів, таймери), які не повинні блокувати інтерфейс.

Механізм роботи:

- ***Call Stack (Стек викликів):*** Сюди потрапляють функції для виконання.
- ***Web APIs:*** Браузер надає додаткові інструменти (таймери, HTTP-запити), які працюють поза основним потоком JS.
- ***Callback Queue (Черга зворотних викликів):*** Сюди потрапляють результати асинхронних операцій після завершення.
- ***Event Loop (Цикл подій):*** Постійно перевіряє: якщо стек порожній, він бере першу задачу з черги та переміщує її в стек.

1.2. Callbacks (Зворотні виклики) та Callback Hell

Найперший спосіб обробки асинхронності – передача однієї функції в іншу як аргументу.

Приклад:

```
function fetchData(callback) {
    setTimeout(() => {
        console.log("Дані отримано");
        callback({ id: 1, name: "Admin" });
    }, 2000);
}

fetchData((user) => {
    console.log("Користувач:", user);
});
```

Проблема (Callback Hell): Коли операції залежать одна від одної, виникає вкладеність, яку важко читати та підтримувати:

```
getData(a => {
    getMoreData(a, b => {
        getEvenMoreData(b, c => {
            // І так далі...
        });
    });
});
```

1.3. Promises (Проміси)

Проміс – це об'єкт, що представляє результат асинхронної операції, який може бути доступний зараз, пізніше або ніколи.

Стани промісу:

1. ***Pending (Очікування):*** Початковий стан.
2. ***Fulfilled (Виконано успішно):*** Операція завершена, є результат (resolve).
3. ***Rejected (Відхилено):*** Помилка (reject).

Створення та використання:

```
const myPromise = new Promise((resolve, reject) => {
    const success = true;
    setTimeout(() => {
        if (success) resolve("Успіх!");
        else reject("Помилка мережі");
    }, 1000);
});

myPromise
    .then(result => console.log(result)) // Обробка успіху
    .catch(error => console.error(error)) // Обробка помилки
    .finally(() => console.log("Завершено")); // Виконується завжди
```

1.4. Async / Await

Це «синтаксичний цукор» над промісами, що дозволяє писати асинхронний код так, ніби він синхронний.

- async – робить функцію асинхронною (вона завжди повертає проміс).
- await – зупиняє виконання функції до моменту розв'язання промісу.

Приклад:

```
async function getUserData() {
```

```
        try {
            const user = await myPromise; // Чекаємо на результат
            console.log(user);
        } catch (err) {
            console.error("Сталася помилка:", err);
        }
    }
```

1.5. Робота з Fetch API

fetch() – це сучасний інтерфейс для виконання мережевих запитів. Він базується на промісах.

Базовий GET-запит:

```
    fetch('[https://jsonplaceholder.typicode.com/posts/1](https://jsonplaceholder.typicode.com/posts/1)')
        .then(response => {
            if (!response.ok) throw new Error("Помилка статусу: " + response.status);
            return response.json(); // Парсинг JSON у об'єкт JS
        })
        .then(data => console.log(data))
        .catch(error => console.error('Помилка запиту:', error));
```

2. Практичні завдання

Приклад 1: Комбінування декількох запитів

Потрібно отримати дані про користувача, а потім, використовуючи його ID, отримати список його постів.

Рішення (Async/Await):

```
    async function getUserPosts(userId) {
        const API_URL = '[https://jsonplaceholder.typicode.com](https://jsonplaceholder.typicode.com)';

        try {
            // Отримуємо користувача
            const userRes = await fetch(`${API_URL}/users/${userId}`);
            const user = await userRes.json();
            console.log(`Користувач: ${user.name}`);
```

```
            // Отримуємо пости користувача
            const postsRes = await
fetch(`${API_URL}/posts?userId=${userId}`);
            const posts = await postsRes.json();

            console.log(`Знайдено постів: ${posts.length}`);
            posts.slice(0, 3).forEach(p => console.log(`-
${p.title}`));
        } catch (error) {
            console.error("Невдача при завантаженні:", error);
        }
    }

    getUserPosts(1);
```

Приклад 2: Обробка масиву промісів (Promise.all)

Замість того, щоб чекати запити по черзі, ми можемо запустити їх паралельно.

Приклад паралельних запитів:

```
    async function fetchMultipleResources() {
        const urls = [

'[https://jsonplaceholder.typicode.com/todos/1](https://jsonplaceho
lder.typicode.com/todos/1)',

'[https://jsonplaceholder.typicode.com/todos/2](https://jsonplaceho
lder.typicode.com/todos/2)'
        ];

        try {
            const promises = urls.map(url => fetch(url).then(res =>
res.json()));
            const results = await Promise.all(promises); // Чекаємо
всі разом

            console.log("Всі дані отримано:", results);
        } catch (error) {
            console.error("Один із запитів не вдався", error);
        }
    }
```

3. Детальний розгляд: Створення власного HTTP-клієнта

Розглянемо створення об'єкта, який спрощує роботу з різними методами (GET, POST).

```
const HttpClient = {
    async request(url, options = {}) {
        const response = await fetch(url, {
            ...options,
            headers: {
                'Content-type':             'application/json; charset=UTF-8',
                ...options.headers
            }
        });

        if (!response.ok) {
            throw new Error(`HTTP Error: ${response.status}`);
        }
        return await response.json();
    },

    get(url) { return this.request(url); },

    post(url, body) {
        return this.request(url, {
            method: 'POST',
            body: JSON.stringify(body)
        });
    }
};
// Використання:
HttpClient.post('[https://jsonplaceholder.typicode.com/posts](https://jsonplaceholder.typicode.com/posts)', { title: 'JS лекція', body: 'Асинхронність' })
    .then(data => console.log("Створено пост:", data))
    .catch(err => console.error(err));
```

4. Порівняльна таблиця методів асинхронності

Характеристика	Callbacks	Promises	Async/Await
Читабельність	Низька (вкладеність)	Середня (ланцюжки)	Висока (як лінійний код)

Обробка помилок	Вручну в кожній ф-ції	.catch()	try...catch
Паралельне виконання	Складно реалізувати	Promise.all()	await Promise.all()
Сучасний стандарт	Застарілий	Актуальний	Рекомендований

5. Контрольні запитання

1. Чому JavaScript називають однопотоковим і як він справляється з асинхронністю?
2. Опишіть роль Event Loop та Callback Queue.
3. Що таке "Callback Hell" і якими способами його можна уникнути?
4. Які три основні стани може мати об'єкт Promise?
5. Чим відрізняється обробка помилок у ланцюжках .then().catch() від блоку try...catch в async/await?
6. Поясніть, що повертає функція, перед якою стоїть ключове слово async.
7. Яка різниця між методами Promise.all() та Promise.race()?
8. Чому перший .then() у Fetch-запиті зазвичай повертає response.json(), а не самі дані?
9. Як перевірити, чи був запит fetch успішним (наприклад, чи не повернув він 404 помилку)?
10. Що станеться, якщо викликати await не всередині функції з позначкою async?

6. Завдання для самостійної роботи

1. ***Реалізація затримки:*** Напишіть функцію delay(ms), яка повертає проміс, що резолвиться через ms мілісекунд. Використайте її в async функції.
2. ***Робота з публічним API:*** Використовуючи fetch, отримайте список користувачів з https://jsonplaceholder.typicode.com/users та відобразіть їхні імена в консолі.
3. ***Симуляція завантаження:*** Створіть кнопку на HTML-сторінці, при натисканні на яку імітується запит до сервера (затримка 2 сек), під час якого на кнопці з'являється напис "Завантаження...", а після – дані з'являються в документі.

ТЕМА 13. ЗБЕРІГАННЯ ДАНИХ НА СТОРОНІ КЛІЄНТА: COOKIES, LOCALSTORAGE, SESSIONSTORAGE

Мета: Вивчити основні механізми веб-сховищ, зрозуміти принципи роботи з Cookies та Web Storage API (LocalStorage, SessionStorage), навчитися обирати оптимальний спосіб зберігання даних залежно від завдань проєкту, опанувати безпечні методи роботи з клієнтськими даними.

1. Клієнтські дані

1.1. Еволюція та необхідність клієнтського зберігання

У сучасній веб-розробці часто виникає потреба зберігати дані безпосередньо у браузері користувача. Це дозволяє:

- Зберігати стан автентифікації (логін користувача).
- Персоналізувати інтерфейс (темна/світла теми).
- Зберігати кошик товарів в інтернет-магазинах.
- Підвищувати продуктивність (кешування налаштувань).

До появи HTML5 основним методом були лише Cookies, проте зараз можна використовувати потужний інструментарій Web Storage API.

1.2. Cookies (Куки)

Cookies – це невеликі фрагменти текстових даних (до 4 КБ), які сервер надсилає браузеру. Браузер зберігає їх і автоматично додає до кожного наступного запиту до того самого сервера.

Основні характеристики:

- ***Обмеження розміру:*** ~4 КБ на один файл.
- ***Термін дії:*** Може бути обмежений сесією або конкретною датою (Expires/Max-Age).
- ***Передача даних:*** Автоматично надсилаються на сервер у заголовку Cookie.
- ***Безпека:*** Підтримують прапорці HttpOnly (захист від XSS) та Secure (передача тільки через HTTPS).

Робота з Cookies через JavaScript:

```
// Встановлення куки
document.cookie = "username=Ivan; max-age=3600; path=/";

// Читання (повертає рядок усіх кук)
console.log(document.cookie);
```

```
// Видалення (встановлення минулої дати)
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
```

1.3. Web Storage API: LocalStorage та SessionStorage

Web Storage API надає механізми зберігання за принципом "ключ-значення". Це набагато зручніше та швидше за куки, оскільки дані не передаються на сервер з кожним запитом.

LocalStorage (Локальне сховище)

Дані в localStorage зберігаються безстроково. Вони не видаляються при закритті вкладки або перезапуску браузера.

- ***Об'єм:*** ~5-10 МБ.
- ***Область видимості:*** Доступно для всіх вкладок з однаковим походженням (Origin: протокол + домен + порт).

SessionStorage (Сесійне сховище)

Дані існують лише протягом поточної сесії вкладки.

- ***Термін дії:*** Видаляються при закритті конкретної вкладки.
- ***Область видимості:*** Тільки в межах однієї вкладки (навіть два вікна з одним сайтом мають різні sessionStorage).

1.4. Методи роботи з Web Storage

Методи ідентичні для обох об'єктів:

1. setItem(key, value) – зберегти дані.
2. getItem(key) – отримати дані за ключем.
3. removeItem(key) – видалити конкретний запис.
4. clear() – повністю очистити сховище.
5. length – кількість записів.

2. Практичні приклади

2.1. Робота з об'єктами (JSON серіалізація)

Сховища localStorage та sessionStorage можуть зберігати лише рядки. Для збереження об'єктів або масивів використовується JSON.stringify(), а для читання – JSON.parse().

```
const userSettings = {
    theme: 'dark',
    fontSize: '16px',
```

```
        notifications: true
    };

    // Збереження об'єкта
    localStorage.setItem('settings', SON.stringify(userSettings));

    // Отримання об'єкта
    const savedSettings =
JSON.parse(localStorage.getItem('settings'));
    console.log(savedSettings.theme); // 'dark'
```

2.2. Реалізація перемикача теми (Theme Switcher)

Приклад використання localStorage для збереження вибору користувача після оновлення сторінки.

```
    const themeBtn = document.querySelector('#theme-toggle');

    // Функція застосування теми
    function applyTheme(theme) {
        document.body.className = theme;
        localStorage.setItem('app-theme', theme);
    }

    // Перевірка при завантаженні
    const  currentTheme  =  localStorage.getItem('app-theme')  ||
'light';
    applyTheme(currentTheme);

    themeBtn.addEventListener('click', () => {
        const newTheme = document.body.classList.contains('light')
? 'dark' : 'light';
        applyTheme(newTheme);
    });
```

2.3. Використання SessionStorage для форм

Корисно для збереження введених даних, якщо користувач випадково оновив сторінку, але не закрив вкладку.

```
    const input = document.querySelector('#comment-field');

    // Відновлення тексту
    input.value = sessionStorage.getItem('draft-comment') || '';
```

```
// Збереження при введенні
input.addEventListener('input', (e) => {
    sessionStorage.setItem('draft-comment', e.target.value);
});

// Очищення після відправки
document.querySelector('#form').onsubmit = () => {
    sessionStorage.removeItem('draft-comment');
};
```

3. Порівняльна таблиця сховищ

Критерій	Cookies	LocalStorage	SessionStorage
Місткість	4 КБ	5-10 МБ	5-10 МБ
Термін життя	Встановлюється (Expires)	Назавжди	До закриття вкладки
Доступність	Будь-яке вікно (той же домен)	Будь-яке вікно (той же домен)	Лише одна вкладка
Зв'язок з сервером	Передаються з запитами	Тільки клієнтська сторона	Тільки клієнтська сторона
Складність API	Складно (текстовий рядок)	Просто (методи)	Просто (методи)

4. Безпека та обмеження

1. ***Конфіденційні дані:*** Ніколи не зберігайте паролі, номери карт або персональні дані у відкритому вигляді в localStorage. До них легко отримати доступ через будь-який JS-скрипт (XSS атака).
2. ***Обмеження домену:*** Сховище прив'язане до протоколу та домену. Дані з http://site.com не будуть доступні на https://site.com.
3. ***Режим інкогніто:*** У багатьох браузерах у режимі інкогніто localStorage очищується після закриття вікна або має нульовий ліміт.
4. ***Квоти:*** При переповненні сховища браузер викине помилку QuotaExceededError. Завжди варто використовувати try...catch при записі великих об'ємів даних.

5. Контрольні запитання

1. Яке головне обмеження розміру даних для одного файлу Cookie?

2. Чим відрізняється автоматична поведінка Cookies від LocalStorage при взаємодії з сервером?
3. Що станеться з даними в sessionStorage, якщо користувач відкриє ту саму сторінку в новій вкладці?
4. Чому для збереження масивів у localStorage необхідно використовувати JSON.stringify()?
5. Як видалити всі дані з localStorage однією командою?
6. Який параметр Cookie робить їх недоступними для JavaScript (захищає від крадіжки через скрипти)?
7. У яких випадках краще використовувати Cookies замість Web Storage?
8. Чи мають різні піддомени (наприклад, blog.example.com та shop.example.com) спільне localStorage?
9. Як перевірити кількість елементів, збережених у сховищі?
10. Яка подія (event) дозволяє відстежувати зміни в localStorage з інших вкладок того самого сайту?

6. Завдання для самостійної роботи

1. ***"Кошик товарів":*** Створіть масив об'єктів (товари) і збережіть його в localStorage. Напишіть функцію, яка додає новий товар до масиву в сховищі, не видаляючи попередні.
2. ***"Таймер сесії":*** Використовуючи sessionStorage, реалізуйте таймер, який показує користувачеві, скільки секунд він перебуває на сторінці. При оновленні сторінки час має продовжуватися, а при відкритті в новій вкладці – починатися з нуля.
3. ***"Аналізатор Cookies":*** Напишіть функцію, яка перетворює рядок document.cookie на зручний JavaScript-об'єкт.
4. ***"Валідатор квоти":*** Напишіть скрипт, який у циклі записує дані в localStorage, поки не виникне помилка переповнення, і виведіть у консоль приблизний максимальний розмір сховища у вашому браузері.

ТЕМА 14. ВСТУП ДО TYPESCRIPT: ТИПІЗАЦІЯ, ІНТЕРФЕЙСИ, GENERIC-ТИПИ У ВЕБ-РОЗРОБЦІ

Мета: Ознайомитися з перевагами TypeScript над JavaScript, вивчити систему типів, механізми створення інтерфейсів та використання узагальнень (Generics), навчитися писати безпечний та масштабований код для веб-додатків.

1. TypeScript

1.1. Введення в TypeScript

TypeScript – це надбудова (superset) над JavaScript, розроблена компанією Microsoft. Він додає в мову сувору типізацію та сучасні можливості ООП, які потім компілюються (транспілюються) у звичайний JavaScript.

Основні переваги:

- ***Виявлення помилок на етапі розробки:*** Більшість помилок "undefined is not a function" виловлюються під час написання коду.
- ***Автодоповнення (Intellisense):*** IDE точно знає структуру об'єктів.
- ***Документування коду:*** Типи слугують живою документацією.
- ***Підтримка великих проєктів:*** Полегшує рефакторинг та роботу в команді.

1.2. Базові типи даних

У TypeScript явно вказуються типи змінних за допомогою двокрапки :.

- ***Примітиви:*** string, number, boolean, null, undefined, symbol.
- ***Масиви:*** number[] або Array<number>.
- ***Any:*** Вимикає перевірку типів (використовувати вкрай не рекомендується).
- ***Unknown:*** Більш безпечна альтернатива any.
- ***Void:*** Використовується для функцій, що не повертають значення.
- ***Never:*** Для функцій, які ніколи не закінчуються (наприклад, викидають помилку).

1.3. Складні типи: Union та Type Aliases

Union Types (Об'єднання): дозволяють змінній мати кілька типів.

```
let id: string | number;
id = 101;     // OK
id = "A101";  // OK
```

Type Aliases (Псевдоніми типів): дозволяють створювати власні імена для складних типів.

type ID = string | number;
type UserRole = "admin" | "editor" | "viewer";

let myId: ID = 5;
let role: UserRole = "admin";

1.4. Інтерфейси (Interfaces)

Interfaces описують форму об'єкта. Вони є основою для типізації сутностей у веб-додатках (користувачі, товари, відповіді від API).

Можливості інтерфейсів:

- ? – необов'язкові поля.
- readonly – поля тільки для читання.
- extends – успадкування структури.

1.5. Generic-типи (Узагальнення)

Generics дозволяють створювати компоненти (функції або класи), які працюють з різними типами, зберігаючи при цьому цілісність типів. Це «типи як параметри».

```
// Функція, яка повертає переданий аргумент, незалежно від його типу
function identity<T>(value: T): T {
    return value;
}

// Використання функції з різними типами
const num = identity<number>(42); // num буде типу number
const str = identity<string>("Привіт"); // str буде типу string
const bool = identity<boolean>(true); // bool буде типу Boolean
```

В розглянутому прикладі *T* – це Generic-параметр типу, який дозволяє функції працювати з будь-яким типом, зберігаючи правильність типів під час компіляції. Слід звернути увагу, що ім’я Generic-параметр типу *T* може бути довільне в межах допустимих імен.

2. Практичні приклади

Приклад 1: Типізація функцій та об'єктів

Уявімо роботу з даними користувача в реальному проєкті:

```
interface User {
    readonly id: number;
    name: string;
    email: string;
    age?: number; // Необов'язкове поле
}

function greetUser(user: User): string {
    return `Вітаємо, ${user.name}! Ваш email: ${user.email}`;
}

const newUser: User = {
    id: 1,
    name: "Олексій",
    email: "alex@example.com"
};

console.log(greetUser(newUser));
```

Приклад 2: Робота з Generics у функціях

Розглянемо приклад функції для отримання випадкового елемента з будь-якого масиву:

```
function getRandomElement<T>(items: T[]): T {
    const randomIndex = Math.floor(Math.random() * items.length);

    return items[randomIndex];
}

const numbers = [1, 5, 10, 25];
const randomNum = getRandomElement<number>(numbers); // Поверне number

const strings = ["Apple", "Orange", "Banana"];
const randomStr = getRandomElement(strings); // Автоматично виведе тип string
```

Приклад 3: Generics в інтерфейсах (API Response)

Типізація стандартної відповіді від сервера є ідеальним місцем для Generics:

```
interface ApiResponse<T> {
    data: T;
    status: number;
    error: string | null;
}

interface Product {
    title: string;
    price: number;
}

// Відповідь для товарів
const productResponse: ApiResponse<Product[]> = {
    data: [{ title: "Laptop", price: 1000 }],
    status: 200,
    error: null
};

// Відповідь для одного користувача (опис User в приклад 1)
const userResponse: ApiResponse<User> = {
    data: newUser,
    status: 200,
    error: null
};
```

3. Концепції

3.1. Перерахування (Enum)

Дозволяють визначати набір іменованих констант.

```
enum OrderStatus {
    Pending = "PENDING",
    Shipped = "SHIPPED",
    Delivered = "DELIVERED"
}

let currentStatus = OrderStatus.Pending;
```

3.2. Tuple (Кортежі)

Масиви з фіксованою кількістю та типом елементів.

```
let contact: [string, number];
contact = ["Kyiv", 44000]; // OK
// contact = [44000, "Kyiv"]; // Помилка: неправильний порядок типів
```

3.3. Type Assertion (Приведення типів)

Коли розробник знає тип краще за TypeScript (наприклад, при роботі з DOM).

```
const myInput = document.getElementById("search-input") as HTMLInputElement;
console.log(myInput.value); // Без 'as' TypeScript видасть помилку, бо не знає про 'value' у HTMLElement
```

4. Порівняння: Interface з Type

Interface та Type схожі, але є відмінності:

1. *Interfaces* можуть бути розширені через декларативне злиття (якщо оголосити два інтерфейси з однаковим ім'ям, вони об'єднаються).
2. *Type* більш гнучкий для об'єднань (|) та перетинів (&).
3. Для опису об'єктів частіше використовують *Interface*, для складних логічних конструкцій – *Type*.

5. Контрольні запитання

1. У чому основна різниця між статичною та динамічною типізацією?
2. Які переваги дає TypeScript під час розробки великих веб-застосунків?
3. Чим тип unknown відрізняється від типу any?
4. Як зробити поле в інтерфейсі необов'язковим для заповнення?
5. Що таке "декларативне злиття" інтерфейсів?
6. Поясніть концепцію Generics. Навіщо використовувати параметр <T>?
7. Як типізувати функцію, яка нічого не повертає?
8. Для чого використовується ключове слово readonly?
9. Як працює успадкування інтерфейсів через ключове слово extends?
10. Яким чином TypeScript допомагає уникати помилок при зверненні до властивостей об'єкта, що прийшов з API?

6. Завдання для самостійної роботи

1. ***"Бібліотека":*** Створіть інтерфейс Book з полями title, author, year та isBorrowed. Створіть масив таких об'єктів та напишіть функцію, яка фільтрує лише доступні книги.
2. ***"Generic Storage":*** Реалізуйте клас DataStorage<T>, який має приватний масив для зберігання даних, метод addItem(item: T) та getItems(): T[]. Протестуйте його з числами та рядками.
3. ***"API Mocking":*** Опишіть тип для відповіді від API погоди, де є координати (об'єкт), температура (число) та опис (union тип: "sunny" | "cloudy" | "rainy").
4. ***"Кортежі в React":*** Спробуйте типізувати функцію, що імітує хук useState, де вона повертає кортеж зі значенням та функцією для його зміни (використовуйте Generics).

ТЕМА 15. ОСНОВИ REACT.JS. JSX/TSX СИНТАКСИС. КОМПОНЕНТНИЙ ПІДХІД

Мета: Вивчити основні концепції бібліотеки React, зрозуміти різницю між імперативним та декларативним підходами, опанувати синтаксис JSX/TSX та навчитися створювати функціональні компоненти з використанням типізації.

1. React

1.1. Введе в React.js

React – це JavaScript-бібліотека з відкритим кодом для створення інтерфейсів користувача (UI), розроблена компанією Meta (Facebook).

Ключові особливості:

- ***Декларативність:*** Ми описуємо, *що* хочемо бачити на екрані, а не *як* це зробити крок за кроком.
- ***Компонентна архітектура:*** Інтерфейс розбивається на незалежні блоки (компоненти), які можна використовувати повторно.
- ***Virtual DOM:*** React створює віртуальну копію реального дерева DOM, порівнює зміни та оновлює лише ті частини сторінки, які дійсно змінилися. Це значно підвищує продуктивність.

1.2. Синтаксис JSX та TSX

JSX (JavaScript XML) – це розширення синтаксису, яке дозволяє писати HTML-подібний код безпосередньо в JavaScript. TSX – це те саме, що і JSX, але з підтримкою типізації TypeScript.

Правила JSX/TSX:

1. ***Повернення одного кореневого елемента:*** Компонент має повертати лише один батьківський елемент. Якщо елементів кілька, їх огортають у фрагмент <> ... </> або .
2. ***Закриття тегів:*** Усі теги повинні бути закриті (навіть
 або <img />).
3. ***camelCase для атрибутів:*** Замість onclick пишемо onClick, замість class – className.
4. ***Вбудовування виразів:*** Для використання JavaScript всередині розмітки використовуються фігурні дужки { }.

1.3. Компонентний підхід

Весь інтерфейс у React – це дерево компонентів.

- ***Компонент*** – це функція (або клас), яка приймає вхідні дані (Props) та повертає опис інтерфейсу.
- ***Reusability (Повторне використання):*** Створивши кнопку або картку товару один раз, ми можемо використовувати її сотні разів з різними даними.

2. Практичні приклади

Приклад 1: Створення першого компонента (TSX)

Опишемо простий компонент вітання з використанням TypeScript для типізації пропсів.

```
import React from 'react';

// Описуємо тип вхідних даних (props)
interface GreetingProps {
  name: string;
  role?: string; // Необов'язкове поле
}

// Функціональний компонент
const Greeting: React.FC<GreetingProps> = ({ name, role = "Користувач" }) => {
  return (
    <div className="greeting-card">
      <h1>Привіт, {name}!</h1>
      <p>Ваша роль: <strong>{role}</strong></p>
    </div>
  );
};

export default Greeting;
```

Приклад 2: Вкладеність компонентів

Компоненти можуть містити інші компоненти, утворюючи ієрархію.

```
const Header = () => (
  <header>
    <nav>Головна | Про нас | Контакти</nav>
  </header>
```

```
);

const App = () => {
  return (
    <div className="app-container">
      <Header />
      <main>
        <Greeting name="Андрій" role="Адміністратор" />
        <p>Ласкаво просимо до нашої панелі керування.</p>
      </main>
    </div>
  );
};
```

Приклад 3: Робота зі списками та атрибутом key

Для відображення масивів даних використовується метод .map(). Важливо вказувати унікальний атрибут key для кожного елемента.

```
interface User {
  id: number;
  userName: string;
}

const UserList = () => {
  const users: User[] = [
    { id: 1, userName: "Олена" },
    { id: 2, userName: "Іван" },
    { id: 3, userName: "Марія" }
  ];

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.userName}</li>
      ))}
    </ul>
  );
};
```

3. Props та State

3.1. Передача даних (Props)

Пропси (properties) – це дані, які передаються від батьківського компонента до дочірнього. Вони є immutable (незмінними) для дочірнього компонента.

3.2. Внутрішній стан (State)

Стан (State) – це дані всередині компонента, які можуть змінюватися з часом (наприклад, введення тексту в поле, клік по кнопці). Для цього використовується хук useState.

```
import React, { useState } from 'react';

const Counter = () => {
  // Ініціалізація стану: [значення, функція_зміни]
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Ви натиснули {count} разів</p>
      <button onClick={() => setCount(count + 1)}>
        Збільшити
      </button>
    </div>
  );
};
```

4. Файлова структура React-проєкту

Типовий проєкт, створений через Vite або CRA, має такий вигляд:

- src/ – основна папка з кодом.
- src/components/ – папка для багаторазових компонентів.
- src/App.tsx – головний компонент додатку.
- src/main.tsx (або index.tsx) – точка входу, де React монтується в DOM.

5. Контрольні запитання

1. У чому полягає різниця між бібліотекою та фреймворком на прикладі React?
2. Що таке Virtual DOM і як він оптимізує роботу браузера?
3. Чому JSX не є валідним JavaScript і як браузер його розуміє?
4. Які обов'язкові правила написання JSX коду ви знаєте?

5. Чим відрізняються Props від State?
6. Навіщо React потрібен атрибут key при рендерингу списків? Що станеться, якщо його не вказати?
7. Яку роль відіграє TypeScript (TSX) у розробці на React?
8. Як реалізувати умовний рендеринг у React (відображення елемента лише за певної умови)?
9. Чи можна змінювати пропси безпосередньо всередині дочірнього компонента? Чому?
10. Яким чином дані передаються від дочірнього компонента до батьківського?

6. Завдання для самостійної роботи

1. ***"Картка товару":*** Створіть компонент ProductCard, який приймає назву товару, ціну та URL зображення через пропси. Додайте стилізацію (через CSS-класи або inline-styles).
2. ***"Перемикач теми":*** Реалізуйте компонент, який за допомогою useState змінює свій колір фону (світлий/темний) при натисканні на кнопку.
3. ***"Динамічний пошук":*** Створіть список рядків. Додайте текстове поле (input), і при введенні тексту в поле список має фільтруватися (відображати лише ті елементи, що містять введені символи).
4. ***"Компонент-обгортка":*** Створіть компонент CardWrapper, який використовує пропс children для того, щоб огорнути будь-який переданий йому контент у рамку з тінню та відступами.

ТЕМА 16. СТАН (STATE) ТА ПРОПСИ (PROPS). ХУКИ (USESTATE, USEEFFECT, USEMEMO)

Мета: Опанувати механізми передачі даних (Props) та керування внутрішнім станом (State). Вивчити принципи роботи основних хуків React для керування побічними ефектами та оптимізації продуктивності.

1. Props (Властивості)

1.1. Введення в Props

Props – це об'єкт, що містить дані, які передаються від батьківського компонента до дочірнього. Це основний спосіб комунікації між компонентами в ієрархії React.

Ключові принципи Props:

- ***Односпрямованість:*** Дані течуть лише зверху вниз (від батька до дитини).
- ***Read-Only (Тільки для читання):*** Компонент ніколи не повинен змінювати свої власні пропси. Вони вважаються "чистими" (immutable).
- ***Пропс-дрілінг (Prop Drilling):*** Процес передачі даних через декілька рівнів компонентів (може бути проблемою у великих додатках).

1.2. State (Стан)

State – це вбудований об'єкт React, який використовується для зберігання даних, специфічних для конкретного компонента, які можуть змінюватися протягом часу.

1.3. Вступ до React Hooks

Хуки – це функції, які дозволяють "підчепитися" до стану та функцій життєвого циклу React у функціональних компонентах. Вони були введені в версії 16.8.

Правила хуків:

1. Викликати хуки лише на верхньому рівні (не всередині циклів, умов чи вкладених функцій).
2. Викликати хуки лише у функціональних компонентах React або власних (custom) хуках.

2. Робота з хуком useState

Хук useState дозволяє додати стан до функціонального компонента.

Практичний приклад: Керування формою

```
import React, { useState } from 'react';

const UserProfile = () => {
  // Стан для імені користувача
  const [name, setName] = useState<string>('Гість');
  // Стан для статусу онлайн
  const [isOnline, setIsOnline] = useState<boolean>(false);

  return (
    <div style={{ padding: '20px', border: '1px solid #ccc' }}>
      <h3>Профіль: {name}</h3>
      <p>Статус: {isOnline ? 'В мережі' : 'Офлайн'}</p>

      <input
        type="text"
        onChange={(e) => setName(e.target.value)}
        placeholder="Введіть ім'я"
      />

      <button onClick={() => setIsOnline(!isOnline)}>
        Змінити статус
      </button>
    </div>
  );
};
```

3. Керування побічними ефектами з useEffect

Хук useEffect дозволяє виконувати побічні ефекти (Side Effects) у функціональних компонентах: запити до API, підписки, маніпуляції з DOM.

Синтаксис: useEffect(callback, dependencies)

1. ***Без масиву залежностей:*** Виконується після кожного рендеру.
2. ***Порожній масив []:*** Виконується лише один раз при монтуванні (аналог componentDidMount).
3. ***Зі значеннями [prop, state]:*** Виконується при зміні вказаних значень.

Практичний приклад: Запит до API

```
import React, { useState, useEffect } from 'react';
```

```
interface Post {
  id: number;
  title: string;
}

const PostLoader = () => {
  const [posts, setPosts] = useState<Post[]>([]);
  const [loading, setLoading] = useState<boolean>(true);

  useEffect(() => {
    // Функція для завантаження даних

fetch('[https://jsonplaceholder.typicode.com/posts?_limit=5](https:
//jsonplaceholder.typicode.com/posts?_limit=5)')
      .then(response => response.json())
      .then(data => {
        setPosts(data);
        setLoading(false);
      })
      .catch(error => console.error('Помилка:', error));

    // Функція очищення (Cleanup) - спрацьовує при розмонтуванні
    return () => console.log('Компонент видалено');
  }, []); // Порожній масив - лише при завантаженні сторінки

  if (loading) return <p>Завантаження...</p>;

  return (
    <ul>
      {posts.map(post => <li key={post.id}>{post.title}</li>)}
    </ul>
  );
};
```

4. Оптимізація з useMemo

Хук useMemo повертає мемоїзоване значення. Він корисний, коли у вас є складні обчислення, які не потрібно повторювати при кожному рендері, якщо вхідні дані не змінилися.

Практичний приклад: Фільтрація великого списку

```
import React, { useState, useMemo } from 'react';
```

```
    const ExpensiveCalculation = () => {
      const [count, setCount] = useState(0);
      const [items] = useState(['Яблуко', 'Банан', 'Апельсин',
'Груша']);
      const [search, setSearch] = useState('');

      // Обчислення виконується лише тоді, коли змінюється 'search'
або 'items'
      const filteredItems = useMemo(() => {
        console.log('Фільтрація...');
        return items.filter(item =>
item.toLowerCase().includes(search.toLowerCase()));
      }, [search, items]);

      return (
        <div>
          <input
            value={search}
            onChange={(e) => setSearch(e.target.value)}
            placeholder="Пошук..."
          />
          <ul>
            {filteredItems.map(item => <li key={item}>{item}</li>)}
          </ul>
          <hr />
          <button onClick={() => setCount(count + 1)}>
            Оновити лічильник (не викличе перерахунок списку):
{count}
          </button>
        </div>
      );
    };
```

5. Контрольні запитання

1. У чому головна відмінність між Props та State? Опишіть життєвий шлях даних у обох випадках.
2. Що таке "Prop Drilling" і чому це вважається антипатерном? Які існують способи його уникнення?
3. Поясніть принцип роботи функції замикання в контексті useState. Чому ми не можемо просто змінити змінну let x = 5 всередині компонента?

4. Коли саме виконується функція очищення (cleanup function) у useEffect? Наведіть приклад її використання для запобігання витоку пам'яті.
5. Яка різниця між useMemo та useCallback? (Додаткове дослідження).
6. Що станеться, якщо передати об'єкт у масив залежностей useEffect без попередньої мемоїзації? Як React порівнює залежності?
7. Чому не можна викликати хуки всередині умови if (condition) { ... }? Як це впливає на внутрішній механізм збереження стану в React?
8. Як реалізувати аналог componentDidUpdate за допомогою useEffect?
9. У чому небезпека використання занадто великої кількості useMemo у додатку? Чи завжди мемоїзація пришвидшує роботу?
10. Як передати функцію від батька до дитини через пропси, щоб змінити стан батька?

6. Завдання для самостійної роботи

1. ***"Таймер зворотного відліку":*** Створіть компонент, який приймає кількість секунд через пропси та запускає таймер за допомогою useEffect та setInterval. Не забудьте очистити інтервал при розмонтуванні.
2. ***"Калькулятор вартості":*** Створіть форму з вибором кількості товарів та ціни. Використовуйте useMemo для обчислення загальної вартості з урахуванням податку, щоб уникнути зайвих обчислень при зміні інших UI-елементів.
3. ***"Синхронізація з LocalStorage":*** Реалізуйте компонент, який зберігає введене ім'я користувача в localStorage за допомогою useEffect кожного разу, коли стан name змінюється. При завантаженні (монтуванні) компонент має зчитувати значення з сховища.

ТЕМА 17. МАРШРУТИЗАЦІЯ В SPA (REACT ROUTER)

Мета: Вивчити архітектуру Single Page Application (SPA) та навчитися реалізовувати навігацію без перезавантаження сторінки. Опанувати бібліотеку React Router для створення динамічних маршрутів, обробки параметрів URL та захисту приватних шляхів.

1. SPA

1.1. SPA та навігація на клієнті

Традиційні веб-сайти (Multi-Page Applications) працюють за принципом: "новий URL = запит на сервер = повне перезавантаження сторінки". *Single Page Application (SPA)* завантажує лише одну HTML-сторінку. Навігація відбувається шляхом маніпуляції DOM-деревом за допомогою JavaScript.

Переваги клієнтського роутингу:

- Відсутність білого екрана при переході між сторінками.
- Швидкість (завантажуються лише дані, а не весь макет).
- Збереження стану додатка (наприклад, музичний плеєр продовжує грати при зміні сторінок).

1.2. React Router: Основні поняття

React Router – це стандартна бібліотека для маршрутизації в React. Вона дозволяє синхронізувати UI з URL-адресою в браузері.

Основні компоненти:

- BrowserRouter: Контейнер, який використовує HTML5 History API для відстеження навігації.
- Routes: Обгортка, яка шукає збіг серед вкладених маршрутів.
- Route: Визначає зв'язок між шляхом (path) та компонентом (element).
- Link / NavLink: Компоненти для створення посилань замість стандартного <a>.

1.3. Динамічні маршрути та параметри

Часто нам потрібно відобразити одну і ту саму сторінку для різних об'єктів (наприклад, профіль користувача або деталі товару). Для цього використовуються параметри шляху: :id, :slug тощо.

2. Налаштування та базовий приклад

Для початку роботи необхідно встановити бібліотеку: npm install react-router-dom

Практичний приклад: Базова структура навігації

```
import React from 'react';
import { BrowserRouter, Routes, Route, Link, NavLink } from 'react-router-dom';

// Компоненти сторінок
const Home = () => <h2>Головна сторінка</h2>;
const About = () => <h2>Про нас</h2>;
const NotFound = () => <h2>404 - Сторінку не знайдено</h2>;

const App = () => {
  return (
    <BrowserRouter>
      <nav style={{ padding: '10px', backgroundColor: '#f4f4f4' }}>
        {/* NavLink дозволяє стилізувати активне посилання */}
        <NavLink
          to="/"
          style={({ isActive }) => ({ color: isActive ? 'red' : 'black', marginRight: '10px' })}
        >
          Головна
        </NavLink>
        <NavLink
          to="/about"
          style={({ isActive }) => ({ color: isActive ? 'red' : 'black' })}
        >
          Про нас
        </NavLink>
      </nav>

      <div style={{ padding: '20px' }}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          {/* Обробка неіснуючих маршрутів */}
          <Route path="*" element={<NotFound />} />
        </Routes>
```

```
        </div>
      </BrowserRouter>
    );
  };

  export default App;
```

3. Робота з динамічними параметрами (useParams)

Коли URL виглядає як /products/123, ми використовується хук useParams для отримання значення 123.

Практичний приклад: Сторінка товару

```
  import { useParams, useNavigate } from 'react-router-dom';

  const ProductDetails = () => {
    const { productId } = useParams<{ productId: string }>();
    const navigate = useNavigate();

    const goBack = () => {
      // Програмна навігація (на крок назад)
      navigate(-1);
    };

    return (
      <div>
        <h3>Деталі товару №{productId}</h3>
        <p>Тут може бути запит до API для отримання даних про товар {productId}.</p>
        <button onClick={goBack}>Повернутися до списку</button>
        <button onClick={() => navigate('/')}>На головну</button>
      </div>
    );
  };

  // В маршрутах:
  // <Route path="/products/:productId" element={<ProductDetails />} />
```

4. Вкладені маршрути (Nested Routes) та Outlet

Вкладені маршрути дозволяють створювати складні інтерфейси, де частина сторінки залишається незмінною (наприклад, бічна панель в особистому кабінеті), а інша частина змінюється залежно від підмаршруту.

Практичний приклад: Панель керування (Dashboard)

```
import { Outlet, Link } from 'react-router-dom';

const Dashboard = () => {
  return (
    <div style={{ display: 'flex' }}>
      <aside style={{ width: '200px', borderRight: '1px solid
gray' }}>
        <ul>
          <li><Link to="profile">Профіль</Link></li>
          <li><Link to="settings">Налаштування</Link></li>
        </ul>
      </aside>
      <main style={{ padding: '20px' }}>
        <h2>Вітаємо в кабінеті!</h2>
        {/* Outlet – це місце, куди будуть рендеритися вкладені
компоненти */}
        <Outlet />
      </main>
    </div>
  );
};

// Конфігурація в Routes:
/*
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<UserProfile />} />
  <Route path="settings" element={<UserSettings />} />
</Route>
*/
```

5. Програмна навігація та захист маршрутів

5.1. Хук useNavigate

Використовується для зміни URL через код (наприклад, після успішної реєстрації або натискання кнопки). const navigate = useNavigate(); Maps('/login');

5.2. Захищені маршрути (Private Routes)

Це патерн, який перевіряє наявність авторизації перед тим, як показати сторінку. Якщо користувач не авторизований, він автоматично перенаправляється на сторінку входу.

```
import { Navigate } from 'react-router-dom';

const PrivateRoute = ({ children, isAuthenticated }: { children:
JSX.Element, isAuthenticated: boolean }) => {
  if (!isAuthenticated) {
    // Перенаправлення, якщо немає доступу
    return <Navigate to="/login" replace />;
  }
  return children;
};

// Використання:
// <Route path="/admin" element={
//   <PrivateRoute isAuthenticated={userLoggedIn}>
//     <AdminPanel />
//   </PrivateRoute>
// } />
```

6. Контрольні запитання

1. Поясніть принципову різницю між Link та звичайним тегом <a>. Чому використання <a> ламає SPA?
2. Для чого потрібен компонент BrowserRouter? Чи можна використовувати кілька таких компонентів в одному додатку?
3. Як передати необов'язковий параметр у маршруті? (Дослідіть документацію v6).
4. У чому різниця між useParams та useSearchParams? Наведіть приклади використання обох хуків.
5. Що таке "Індексний маршрут" (Index Route)? Як його оголосити всередині батьківського маршруту?
6. Як реалізувати програмне повернення на попередню сторінку?
7. Опишіть призначення компонента <Outlet />. Що станеться, якщо його не вказати в батьківському компоненті з вкладеними маршрутами?
8. Як обробити помилку 404 (сторінка не знайдена) в React Router?
9. Що робить пропс replace у компоненті Maps або функції Maps? Як це впливає на історію браузера?
10. Як стилізувати активне посилання в меню за допомогою NavLink? Назвіть доступні параметри у функції стилю.

7. Завдання для самостійної роботи

1. ***"Багатосторінкове портфоліо":*** Створіть додаток з трьома сторінками: "Головна", "Проєкти", "Контакти". Додайте навігаційну панель, яка підсвічує активну сторінку.
2. ***"Каталог фільмів":*** Реалізуйте список фільмів на головній сторінці. При натисканні на фільм користувач повинен переходити на сторінку /movie/:id, де відображається ID фільму, отриманий з URL.
3. ***"Система авторизації (імітація)":*** Створіть сторінку SecretPage, доступ до якої можливий лише при натисканні кнопки "Login" на головній сторінці (збережіть стан авторизації в useState). Якщо користувач намагається зайти на /secret напряму без логіну – перенаправляйте на головну.

ТЕМА 18. ВСТУП ДО NEXT.JS: ГІБРИДНИЙ РЕНДЕРИНГ (SSR, SSG, ISR)

Мета: Вивчити архітектурні особливості фреймворку Next.js, зрозуміти відмінності між клієнтським та серверним рендерингом. Опанувати методи генерації контенту (SSR, SSG, ISR) та навчитися обирати оптимальну стратегію для різних типів бізнес-завдань.

1. Next.js

1.1. Введення в Next.js

Next.js – це React-фреймворк для продакшну, який надає розширені можливості, такі як серверний рендеринг, статична генерація та оптимізація зображень "з коробки".

У стандартному React (SPA) браузер отримує порожній HTML-файл та великий JavaScript-бандл. Це створює дві проблеми:

1. ***SEO:*** Пошукові роботи можуть не побачити контент, який генерується динамічно.
2. ***FCP (First Contentful Paint):*** Користувач змушений чекати завантаження JS, перш ніж побачить інтерфейс.

Next.js вирішує це шляхом пре-рендерингу – генерації HTML на сервері.

1.2. Основні стратегії рендерингу в Next.js

1. ***Client-Side Rendering (CSR):*** Стандартний підхід React. Все виконується в браузері.
2. ***Static Site Generation (SSG):*** HTML генерується один раз під час збірки (build time). Ідеально для блогів, документації.
3. ***Server-Side Rendering (SSR):*** HTML генерується для кожного запиту (request time). Підходить для персоналізованих сторінок (профіль користувача, кошик).
4. ***Incremental Static Regeneration (ISR):*** Гібрид, що дозволяє оновлювати статичні сторінки у фоновому режимі після збірки без повного перезапуску проєкту.

2. Static Site Generation (SSG)

При SSG сторінка створюється на сервері під час команди npm run build. Користувач отримує готовий сталий файл.

Переваги:

- Найвища швидкість (файли віддаються з CDN).
- Мінімальне навантаження на сервер.

Практичний приклад (App Router):

В Next.js компоненти за замовчуванням є *Server Components,* і якщо вони не використовують динамічні функції (наприклад, cookies), вони стають статичними.

```
// app/blog/page.tsx
async function getPosts() {
  const res = await fetch('[https://api.example.com/posts](https://api.example.com/posts)');
  return res.json();
}

export default async function BlogPage() {
  const posts = await getPosts(); // Дані отримані під час збірки

  return (
    <div>
      <h1>Мій Блог</h1>
      {posts.map((post: any) => (
        <article key={post.id}>
          <h2>{post.title}</h2>
        </article>
      ))}
    </div>
  );
}
```

3. Server-Side Rendering (SSR)

SSR генерує сторінку щоразу, коли приходить запит від клієнта. Це необхідно, якщо дані часто змінюються або залежать від конкретного користувача.

Практичний приклад:

Щоб змусити Next.js використовувати SSR, потрібно додати опцію cache: 'no-store' у запит fetch.

```
// app/profile/page.tsx
async function getUserProfile() {
```

```
    // Вимкнення кешування робить цей запит динамічним (SSR)
    const res = await fetch('[https://api.example.com/user/me](https://api.example.com/user/me)', {
      cache: 'no-store'
    });
    return res.json();
  }

  export default async function ProfilePage() {
    const user = await getUserProfile();

    return (
      <div>
        <h1>Вітаємо, {user.name}</h1>
        <p>Ваш баланс: {user.balance} грн</p>
      </div>
    );
  }
```

4. Incremental Static Regeneration (ISR)

ISR –це «золота середина». Ви створюєте статичну сторінку, але вказуєте інтервал часу, через який Next.js повинен перевірити оновлення даних.

Практичний приклад:

Параметр next: { revalidate: 60 } означає, що сторінка буде оновлюватися не частіше ніж раз на хвилину.

```
  // app/stock-prices/page.tsx
  async function getPrices() {
    const res = await fetch('[https://api.example.com/stocks](https://api.example.com/stocks)', {
      next: { revalidate: 60 } // Оновлення кожні 60 секунд
    });
    return res.json();
  }

  export default async function Stocks() {
    const data = await getPrices();
    return (
      <div>
```

```
          <h3>Ціни на акції (Оновлено: {new
Date().toLocaleTimeString()})</h3>
          {/* Вивід даних */}
        </div>
      );
    }
```

5. Client Components та Server Components

Це фундаментальна концепція Next.js App Router.

- ***Server Components (RSC):***
  - Виконуються на сервері.
  - Не мають useState, useEffect.
  - Мають менший бандл (JS не відправляється в браузер).
  - Безпечні для звернення до баз даних та API-ключів.
- ***Client Components:***
  - Позначаються директивою 'use client' зверху файлу.
  - Підтримують інтерактивність (обробники подій, стейт).
  - Рендериться і на сервері (пре-рендеринг), і гідратується (оживає) в браузері.

Приклад поєднання:

```
// layout.tsx (Server Component)
import SearchBar from './SearchBar'; // Client Component

export default function Layout({ children }) {
  return (
    <html>
      <body>
        <nav>
          <h1>Магазин</h1>
          <SearchBar /> {/* Клієнтська логіка всередині
серверної */}
        </nav>
        {children}
      </body>
    </html>
  );
}
```

6. Контрольні запитання

1. Чим Next.js принципово відрізняється від Create React App? Поясніть з точки зору архітектури.
2. Опишіть процес гідратації (Hydration). Чому він важливий для клієнтських компонентів?
3. Коли варто обрати SSG замість SSR? Наведіть 3 приклади реальних проєктів.
4. Як працює ISR? Що відбувається, якщо другий користувач заходить на сторінку через 30 секунд після першого при інтервалі revalidate: 60?
5. Чи можна використовувати window або localStorage у Server Components? Чому?
6. Які переваги дає використання Server Components для продуктивності (LCP, TBT)?
7. Як передати дані від Server Component до Client Component? Які обмеження існують при передачі (пропси)?
8. Що таке Dynamic Routes в Next.js? Як згенерувати статичні шляхи для динамічних маршрутів (функція generateStaticParams)?
9. Як Next.js допомагає в SEO оптимізації? (Метатеги, Metadata API).
10. Поясніть концепцію "Streaming" та компонент <Suspense> у контексті серверного рендерингу.

7. Завдання для самостійної роботи

1. ***Інсталяція:*** Створіть новий проєкт за допомогою npx create-next-app@latest. Виберіть App Router, TypeScript та Tailwind CSS.
2. ***Реалізація SSG:*** Створіть сторінку "Про нас", яка виводить статичний текст. Перевірте вихідний код сторінки в браузері (Ctrl+U) – текст має бути присутній там.
3. ***Робота з API:*** Використовуючи публічне API (наприклад, JSONPlaceholder), виведіть список постів на головній сторінці.
   - Спробуйте спочатку з cache: 'no-store'.
   - Потім додайте revalidate: 10. Спостерігайте за часом оновлення даних.
4. ***Інтерактивність:*** Додайте кнопку "Поставити лайк" до кожного поста. Оскільки це потребує useState, винесіть кнопку в окремий файл з позначкою 'use client'.

ТЕМА 19. СУЧАСНИЙ ІНСТРУМЕНТАРІЙ: VITE, NPM СКРИПТИ, ЗМІННІ ОТОЧЕННЯ

Мета: Ознайомитися з принципами роботи сучасних інструментів збірки проєктів на прикладі Vite. Навчитися автоматизувати рутинні завдання за допомогою NPM скриптів та безпечно керувати конфігураціями проєкту через змінні оточення (.env).

1. Еволюція інструментів збірки: Від Webpack до Vite

1.1. Проблема застарілих бандлерів

До появи Vite основним інструментом був Webpack. Він працює за принципом *bundle-based*: перед тим, як запустити сервер розробки, він повинен просканувати весь проєкт, побудувати граф залежностей і зібрати (bundle) весь код в один файл. У великих проєктах це займає хвилини.

1.2. Концепція Vite

Vite (французькою –"швидкий") використовує кардинально інший підхід:

1. ***Native ESM:*** У режимі розробки Vite не збирає бандл. Він віддає вихідний код частинами за запитом браузера, використовуючи нативні модулі JavaScript (import/export).
2. ***Esbuild:*** Для попередньої обробки залежностей (node_modules) Vite використовує Esbuild, написаний на мові Go, що в 10–100 разів швидше за бандлери на базі JS.
3. ***HMR (Hot Module Replacement):*** Оновлення коду в браузері відбувається миттєво, незалежно від розміру проєкту.

1.3. Створення проєкту

```
# Створення проєкту через термінал
npm create vite@latest my-app
# Вибір: React -> TypeScript -> SWC (найшвидший компілятор)
cd my-app
npm install
npm run dev
```

2. NPM скрипти

Секція scripts у файлі package.json – це потужний інструмент автоматизації. Будь-яка команда, прописана там, може бути запущена через npm run <назва>.

2.1. Стандартні та кастомні скрипти

```
{
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview",
    "lint": "eslint . --ext ts,tsx --report-unused-disable-
directives --max-warnings 0",
    "test": "vitest",
    "clean": "rm -rf dist"
  }
}
```

2.2. Пре- та Пост-скрипти

NPM дозволяє створювати ланцюжки команд за допомогою префіксів pre та post:

- prebuild: виконається автоматично перед build.
- postbuild: виконається автоматично після успішного build.

Приклад автоматизації очищення перед збіркою:

```
"scripts": {
  "prebuild": "npm run clean",
  "build": "vite build",
  "clean": "rimraf dist"
}
```

3. Керування конфігурацією: Змінні оточення (Environment Variables)

Змінні оточення дозволяють змінювати поведінку програми залежно від середовища (Development, Staging, Production) без зміни коду.

3.1. Файли конфігурації (.env)

Vite автоматично завантажує змінні з наступних файлів у корені проєкту:

- .env: Загальний.
- .env.local: Локальний (ігнорується Git).
- .env.development: Тільки для розробки.
- .env.production: Тільки для фінальної збірки.

3.2. Префікс VITE_

З міркувань безпеки Vite дозволяє доступ до змінних у клієнтському коді лише якщо вони починаються з префікса VITE_.

Приклад .env:

```
VITE_API_URL=[https://api.example.com](https://api.example.com)
VITE_STRIPE_KEY=pk_test_12345
# Ця змінна НЕ буде доступна в React:
DB_PASSWORD=secret_pass
```

3.3. Використання в коді

У Vite доступ до змінних здійснюється через об'єкт import.meta.env:

```
const ApiService = () => {
  const url = import.meta.env.VITE_API_URL;

  console.log("Поточний режим:", import.meta.env.MODE); // 'development' або 'production'

  const fetchData = async () => {
    const response = await fetch(`${url}/data`);
    return response.json();
  };
};
```

4. Оптимізація збірки (Build Process)

Коли відбувається запуск npm run build, Vite виконує низку складних трансформацій:

1. ***Tree Shaking:*** Видалення коду, який не використовується (наприклад, функцій з бібліотек, які ви не імпортували).
2. ***Minification:*** Стискання коду (видалення пробілів, скорочення назв змінних).
3. ***Code Splitting:*** Розбиття великого бандла на менші частини (chunks). Це дозволяє браузеру завантажувати лише ті частини коду, які потрібні для поточної сторінки.
4. ***Content Hashing:*** Додавання унікального хешу до назви файлу (наприклад, main.a7b2c8.js). Це вирішує проблему кешування в браузерах.

5. Контрольні запитання

1. Поясніть різницю між Cold Start у Webpack та Vite. Чому Vite запускається швидше?
2. Що таке "нативні ESM модулі"? Як вони змінили підхід до розробки?
3. Навіщо потрібен файл .env.local? Чи варто додавати його до Git репозиторію?
4. Як запустити скрипт, який виконує дві команди послідовно? (Підказка: оператори && та ;).
5. Що станеться, якщо спробувати отримати доступ до змінної оточення без префікса VITE_ у React-компоненті?
6. Яка роль import.meta.env.MODE? Наведіть приклад використання для логування помилок.
7. Для чого використовується команда npm run preview? Чим вона відрізняється від npm run dev?
8. Що таке "Tree Shaking" і чому це важливо для мобільних користувачів?
9. Як налаштувати Vite для підтримки старих браузерів, які не розуміють ESM? (Згадка про @vitejs/plugin-legacy).
10. Поясніть життєвий цикл NPM скриптів: pre, command, post.

6. Завдання для самостійної роботи

1. ***Ініціалізація:*** Створіть проєкт на Vite + React + TS.
2. ***Налаштування середовища:*** Створіть файл .env. Додайте в нього VITE_APP_TITLE=Мій Супер Додаток.
   - Відобразіть цей заголовок у компоненті App.
3. ***Автоматизація скриптів:*** Створіть у package.json скрипт "deploy", який спочатку запускає лінтер (lint), потім робить збірку (build), а в кінці виводить у консоль "Build completed!".
4. ***Аналіз збірки:***
   - Запустіть npm run build.
   - Зайдіть у папку dist/assets та проаналізуйте назви файлів. Знайдіть у коді (використовуючи пошук) значення вашої змінної оточення.

ТЕМА 20. АРХІТЕКТУРА NODE.JS. EVENT LOOP. СТВОРЕННЯ REST СЕРВЕРА НА EXPRESS.JS

Мета: Розглянути Node.js не просто як інструмент, а як середовище виконання. Зрозуміти, як однопотокова модель справляється з високим навантаженням завдяки Event Loop. Вивчити архітектуру REST та навчитися створювати масштабовані веб-сервери за допомогою Express.js.

1. Node.js

1.1. Введення в Node.js

Node.js – це середовище виконання JavaScript (runtime environment), побудоване на рушії V8 від Google Chrome. Воно дозволяє запускати JavaScript поза браузером (на сервері).

Ключові архітектурні особливості:

1. ***Подійно-орієнтованість (Event-driven):*** Вся логіка будується на реакції на події.
2. ***Неблокуючий I/O (Non-blocking I/O):*** Операції введення-виведення (читання файлів, запити до БД) не зупиняють виконання програми.
3. ***Однопотоковість (Single Threaded):*** JS виконується в одному основному потоці, але важкі операції делегуються системному ядру через бібліотеку libuv.

Порівняння з традиційними серверами (наприклад, Apache/PHP або Java):

- ***Традиційні:*** Створюють новий потік (thread) для кожного запиту клієнта. Це вимагає багато пам'яті.
- ***Node.js:*** Обробляє тисячі з'єднань в одному потоці, використовуючи асинхронні виклики.

1.2. Event Loop (Цикл подій) Node.js

Розуміння Event Loop є критичним для написання продуктивного – коду. Це механізм, який дозволяє Node.js виконувати неблокуючі операції вводу/виводу.

Складові частини:

1. ***Call Stack (Стек викликів):*** В стеку виконується синхронний код (LIFO - Last In, First Out).

2. ***Node APIs (C++):*** Сюди потрапляють асинхронні задачі (таймери, HTTP-запити). Вони виконуються паралельно основним потоком.
3. ***Callback Queue (Черга зворотних викликів):*** Сюди потрапляють колбеки готових задач.
4. ***Event Loop:*** Безкінечний цикл, який перевіряє: *"Чи пустий стек? Якщо так, чи є щось у черзі? Якщо так – перемістити це в стек"*.

Фази Event Loop (спрощено):

1. ***Timers:*** Виконуються колбеки setTimeout() та setInterval().
2. ***Pending Callbacks:*** Системні операції (наприклад, помилки TCP).
3. ***Idle, prepare:*** Внутрішні процеси Node.js.
4. ***Poll:*** Отримання нових подій I/O (читання файлів, вхідні запити). Найважливіша фаза.
5. ***Check:*** Тут виконується setImmediate().
6. ***Close Callbacks:*** Закриття з'єднань, socket.on('close', ...).

Microtasks: Окрім основних фаз, є черга мікрозадач (Promise.then, process.nextTick). Вона має найвищий пріоритет і спустошується *повністю* після кожної операції та перед переходом до наступної фази Event Loop.

1.3. Модульна система

Node.js використовує модулі для організації коду.

- ***CommonJS (CJS):*** Стандартний формат для Node.js.
- const http = require('http'); // імпорт
- module.exports = { ... }; // експорт
- ***ES Modules (ESM):*** Сучасний стандарт (як у браузері/React), активується через "type": "module" у package.json.
- import http from 'http';
- export default { ... };

2. Вступ до Express.js

Express.js – це мінімалістичний веб-фреймворк для Node.js, який значно спрощує створення серверів. Без нього довелося б вручну парсити URL, обробляти потоки даних (streams) та керувати заголовками для кожного запиту.

2.1. Концепція Middleware (Проміжне ПЗ)

Це функції, які мають доступ до об'єкта запиту (req), об'єкта відповіді (res) та наступної функції (next). Вони виконуються послідовно.

Типи Middleware:

1. ***Вбудовані:*** express.json() (парсинг тіла запиту), express.static() (роздача файлів).
2. ***Сторонні:*** cors, morgan (логування).
3. ***Кастомні:*** Власні функції для перевірки авторизації тощо.

2.2. Маршрутизація (Routing)

Визначає, як програма відповідає на клієнтський запит до певної кінцевої точки (URI) та певним HTTP-методом (GET, POST тощо).

3. Практична розробка REST API

Створимо сервер *для керування списком "Завдань" (Todo API).*

***Крок* 1: Ініціалізація**

```
mkdir my-express-server
cd my-express-server
npm init -y
npm install express
npm install -D nodemon # для автоперезавантаження
```

У package.json додайте скрипт: "dev": "nodemon index.js".

Крок 2: Базовий сервер (index.js)

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware для парсингу JSON
app.use(express.json());

// Базовий маршрут
app.get('/', (req, res) => {
    res.send('Вітаємо на нашому API!');
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
```

```
});
```

Крок 3: Реалізація CRUD (Create, Read, Update, Delete)

Зберігатимемо дані у пам'яті (масив) для простоти.

```
let todos = [
    { id: 1, title: "Вивчити Node.js", completed: false },
    { id: 2, title: "Написати сервер", completed: true }
];

// GET: Отримати всі завдання
app.get('/api/todos', (req, res) => {
    res.json(todos);
});

// GET: Отримати одне завдання за ID
app.get('/api/todos/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const todo = todos.find(t => t.id === id);

    if (!todo) {
        return res.status(404).json({ message: "Завдання не
знайдено" });
    }
    res.json(todo);
});

// POST: Створити нове завдання
app.post('/api/todos', (req, res) => {
    if (!req.body.title) {
        return res.status(400).json({ message: "Поле title
обов'язкове" });
    }

    const newTodo = {
        id: todos.length + 1, // Проста генерація ID
        title: req.body.title,
        completed: false
    };

    todos.push(newTodo);
    res.status(201).json(newTodo); // 201 Created
});
```

```
    // PUT: Оновити завдання
    app.put('/api/todos/:id', (req, res) => {
        const id = parseInt(req.params.id);
        const todoIndex = todos.findIndex(t => t.id === id);

        if (todoIndex === -1) {
            return res.status(404).json({ message: "Завдання не
знайдено" });
        }

        // Оновлюємо тільки передані поля
        todos[todoIndex] = {
            ...todos[todoIndex],
            ...req.body
        };

        res.json(todos[todoIndex]);
    });

    // DELETE: Видалити завдання
    app.delete('/api/todos/:id', (req, res) => {
        const id = parseInt(req.params.id);
        const todoIndex = todos.findIndex(t => t.id === id);

        if (todoIndex === -1) {
            return res.status(404).json({ message: "Завдання не
знайдено" });
        }

        const deletedTodo = todos.splice(todoIndex, 1);
        res.json({ message: "Видалено успішно", todo:
deletedTodo[0] });
    });
```

Крок 4: Обробка помилок (Global Error Handler)

Express має стандартний механізм для перехоплення помилок. Це middleware з 4 аргументами, яке розміщується в кінці всіх маршрутів.

```
    app.use((err, req, res, next) => {
        console.error(err.stack);
        res.status(500).json({ message: "Щось пішло не так на
сервері!" });
    });
```

4. Глибоке розуміння асинхронності в Node.js

Проблема блокування Event Loop

Незважаючи на швидкість, Node.js легко "покласти", якщо виконати важку синхронну операцію.

Поганий приклад (Блокування):

```
app.get('/heavy', (req, res) => {
    // Цикл на 5 мільярдів ітерацій заблокує потік
    // Жоден інший користувач не отримає відповідь, поки це не завершиться
    let sum = 0;
    for (let i = 0; i < 5000000000; i++) sum += i;
    res.send(`Сума: ${sum}`);
});
```

Висновок: Важкі обчислення (обробка зображень, складна криптографія) не слід виконувати в основному потоці Node.js. Для цього використовують Worker Threads або окремі мікросервіси.

5. Контрольні запитання

1. Чому Node.js називають однопотоковим, якщо він може виконувати файлові операції паралельно? Поясніть роль бібліотеки libuv та пулу потоків (Thread Pool).
2. У чому різниця між process.nextTick() та setImmediate()? Який з них спрацює раніше в циклі подій?
3. Що таке Middleware в Express? Наведіть приклад ланцюжка middleware.
4. Як отримати доступ до даних, відправлених методом POST у форматі JSON? Яке налаштування для цього потрібне?
5. Які переваги та недоліки використання Node.js для CPU-інтенсивних задач?
6. Що означають HTTP статуси 201, 400, 404, 500?
7. Як працює маршрутизація з параметрами (наприклад, /users/:userId/posts/:postId)? Як отримати ці значення в коді?
8. Що станеться, якщо у функції middleware не викликати next() і не повернути відповідь res.send()?
9. Чим відрізняється CommonJS (require) від ES Modules (import) в контексті завантаження модулів (синхронне/асинхронне)?

10. Як організувати структуру проєкту Express для великого додатка (Controllers, Routes, Services)?

6. Завдання для самостійної роботи

1. ***Розширення API:*** Додайте до створеного Todo API можливість пошуку завдань за назвою через query-параметри (наприклад, /api/todos?search=node).
2. ***Валідація:*** Створіть middleware функцію validateTodo, яка перевіряє, чи присутнє поле title і чи воно не порожнє перед створенням або оновленням.
3. ***Логування:*** Напишіть middleware, яке виводить у консоль метод запиту, URL та час отримання запиту для кожного звернення до сервера.
4. ***Робота з файлами:*** Замість масиву в пам'яті, реалізуйте читання та запис списку завдань у файл todos.json за допомогою модуля fs (асинхронно).

ТЕМА 21. РОБОТА З БАЗАМИ ДАНИХ POSTGRESQL/MONGODB. ORM/ODM (PRISMA/MONGOOSE)

Мета: Зрозуміти фундаментальні відмінності між реляційними (SQL) та нереляційними (NoSQL) базами даних, вивчити концепцію ORM (Object-Relational Mapping) та ODM (Object-Document Mapping) як шару абстракції над даними, навчитися моделювати дані та виконувати операції CRUD, використовуючи Mongoose (для MongoDB) та Prisma (для PostgreSQL), розглянути переваги типізації при роботі з базою даних у TypeScript.

1. Екосистема баз даних

1.1. SQL в порівнянні з NoSQL

Вибір бази даних – це одне з найважливіших архітектурних рішень.

Характеристика	SQL (Реляційні)	NoSQL (Нереляційні)
Приклади	PostgreSQL, MySQL, MS SQL	MongoDB, Redis, Cassandra
Структура даних	Таблиці, рядки, стовпці	Документи (JSON), пари ключ-значення
Схема	Сувора (Schema-first). Треба визначити структуру до запису.	Гнучка (Schema-less). Структура може змінюватися "на льоту".
Зв'язки	Потужні JOIN-запити	Обмежені, часто денормалізація даних
Масштабування	Вертикальне (потужніший сервер)	Горизонтальне (більше серверів - шардінг)
ACID	Гарантується (Атомарність, Узгодженість...)	Часто компроміс на користь швидкості (BASE)

Коли обирати PostgreSQL:

- Фінансові транзакції, складні звіти.
- Чітко структуровані дані, які рідко змінюють формат.
- Необхідність у складних SQL-запитах.

Коли обирати MongoDB:

- Великі обсяги неструктурованих даних (логи, профілі соцмереж).
- Швидка розробка (Rapid Prototyping), коли схема часто змінюється.

– Високе навантаження на запис/читання.

1.2. ORM та ODM

Використовувати SQL-запити на зразок SELECT * FROM users WHERE... у коді програми часто незручно, небезпечно (SQL Injection) і складно підтримувати.

– ***ORM (Object-Relational Mapping):*** Технологія, що зв'язує таблиці в базі даних з класами або об'єктами в коді.
  – *Приклад:* Замість SQL запиту ви пишете User.find({ id: 1 }).
  – *Інструменти:* Prisma, TypeORM, Sequelize.
– ***ODM (Object-Document Mapping):*** Аналог для документо-орієнтованих баз. Зв'язує JSON-документи з об'єктами в коді.
  – *Інструменти:* Mongoose.

Переваги використання:

1. ***Безпека:*** Автоматичне екранування даних.
2. ***Швидкість розробки:*** Інтуїтивні методи для CRUD операцій.
3. ***Абстракція:*** Можливість змінити БД без переписування всього коду.
4. ***Валідація:*** Перевірка даних перед записом у БД на рівні додатку.

2. Робота з MongoDB через Mongoose

Mongoose – це найпопулярніша ODM-бібліотека для MongoDB у середовищі Node.js. Вона додає те, чого немає у MongoDB – схеми.

2.1. Підключення та створення схеми

Спочатку необхідно встановити з'єднання з базою.

```
import mongoose from 'mongoose';

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB Connected');
  } catch (error) {
    console.error('Connection Error:', error);
    process.exit(1);
  }
};
```

Схема описує структуру документа, типи полів, валідацію та значення за замовчуванням.

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Будь ласка, введіть ім\'я'],
    trim: true, // Видаляє пробіли по краях
    maxLength: 50
  },
  email: {
    type: String,
    required: true,
    unique: true, // Унікальний індекс
    lowercase: true
  },
  role: {
    type: String,
    enum: ['user', 'admin'], // Дозволені значення
    default: 'user'
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});

// Створення Моделі на основі Схеми
const User = mongoose.model('User', userSchema);
export default User;
```

2.2. Основні операції (CRUD) в Mongoose

Create (Створення):

```
const newUser = await User.create({
  name: 'Микола',
  email: 'mykola@example.com'
});
```

Read (Читання):

```
// Знайти всіх адміністраторів
const admins = await User.find({ role: 'admin' });
```

```
// Знайти одного користувача за ID
const user = await User.findById('64b5f...');

// Вибірка конкретних полів (проекція)
const users = await User.find().select('name email -_id');
```

Update (Оновлення):

```
// Знайти і оновити (повертає стару версію за замовчуванням, тому new: true)
const updatedUser = await User.findByIdAndUpdate(
  id,
  { role: 'admin' },
  { new: true, runValidators: true } // runValidators - важливо!
);
```

Delete (Видалення):

```
await User.findByIdAndDelete(id);
```

3. Робота з PostgreSQL через Prisma ORM

Prisma – це ORM нового покоління. Вона складається з трьох частин:

1. ***Prisma Client:*** Автоматично генерований, типобезпечний клієнт для запитів (ідеально для TypeScript).
2. ***Prisma Migrate:*** Система міграцій (зміни структури БД).
3. ***Prisma Studio:*** GUI для перегляду даних.

3.1. Визначення схеми (schema.prisma)

На відміну від Mongoose, схема визначається в спеціальному файлі, а не в JS-коді.

```
// schema.prisma

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
```

```
  id        Int      @id @default(autoincrement())
  email     String   @unique
  name      String?
  role      Role     @default(USER)
  posts     Post[]   // Зв'язок one-to-many
  createdAt DateTime @default(now())
}

model Post {
  id        Int      @id @default(autoincrement())
  title     String
  content   String?
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references:
[id])
  authorId  Int
}

enum Role {
  USER
  ADMIN
}
```

Після створення схеми необхідно виконати міграцію: npx prisma migrate dev --name init

3.2. Використання Prisma Client

Prisma надає методи, які повністю відповідають вашій схемі. Якщо ви зміните назву поля в схемі, TypeScript підсвітить помилку в коді.

```
import { PrismaClient } from '@prisma/client';
const prisma = new PrismaClient();

async function main() {
  // 1. Create (Створення з вкладеними даними)
  const newUser = await prisma.user.create({
    data: {
      email: 'ivan@example.com',
      name: 'Іван',
      posts: {
        create: {
          title: 'Мій перший пост в Prisma'
        }
```

```
      }
    }
  });

  // 2. Read (Читання з фільтрацією та пагінацією)
  const users = await prisma.user.findMany({
    where: {
      email: { endsWith: '@example.com' }
    },
    include: { posts: true }, // Аналог JOIN в SQL
    take: 10, // Limit
    skip: 0   // Offset
  });

  // 3. Update
  const updatedPost = await prisma.post.update({
    where: { id: 1 },
    data: { published: true }
  });
}
```

4. Порівняння підходів: Mongoose та Prisma

Аспект	Mongoose (ODM)	Prisma (ORM)
База даних	MongoDB (NoSQL)	PostgreSQL, MySQL, SQL Server, MongoDB
Визначення схеми	JavaScript/TypeScript класи	Власний формат .prisma
Типізація (TS)	Потребує ручного опису інтерфейсів або InferSchemaType	Генерується автоматично, "з коробки"
Міграції	Відсутні (schema-less природа)	Потужний інструмент міграцій SQL
Зв'язки (Relations)	Через .populate() (повільніше)	Нативні Foreign Keys (швидко)

5. Контрольні запитання

1. У чому полягає різниця між горизонтальним та вертикальним масштабуванням баз даних? Який тип масштабування притаманний MongoDB?

2. Що таке "SQL Injection" і як використання ORM допомагає захиститися від нього?
3. Поясніть термін "схема" в контексті Mongoose. Чи є вона обов'язковою для самої бази даних MongoDB?
4. Для чого потрібні міграції в реляційних базах даних? Як вони реалізовані в Prisma?
5. Як реалізувати зв'язок "один до багатьох" (One-to-Many) у Prisma?
6. Що робить функція populate() у Mongoose і чому зловживання нею може сповільнити додаток?
7. Які переваги дає використання enum у схемі бази даних?
8. Чому при оновленні запису в Mongoose (findByIdAndUpdate) важливо передавати опцію { new: true }?
9. Як Prisma забезпечує типобезпеку (Type Safety) при роботі з базою даних?
10. Що таке "Connection Pool" і чому важливо закривати (або правильно керувати) з'єднаннями з БД?

6. Завдання для самостійної роботи

1. ***Проєктування схеми (Prisma):*** Опишіть у файлі schema.prisma структуру для інтернет-магазину: моделі Product, Category, Order. Товар повинен належати одній категорії, замовлення може містити багато товарів (Many-to-Many).
2. ***Валідація (Mongoose):*** Створіть схему для коментарів, де поле rating має бути числом від 1 до 5, а текст коментаря не повинен містити нецензурних слів (кастомний валідатор).
3. ***API з БД:*** Створіть простий Express сервер, підключіть його до MongoDB Atlas (хмарна версія) через Mongoose. Реалізуйте один ендпоінт POST /users, який зберігає користувача в хмарі.
4. ***Агрегація даних:*** (Підвищена складність) Використовуючи Prisma або Mongoose, напишіть запит, який повертає топ-5 користувачів за кількістю написаних постів.

ТЕМА 22. ПРОЄКТУВАННЯ API: ДОКУМЕНТУВАННЯ (SWAGGER/OPENAPI)

Мета: Зрозуміти важливість якісної документації для Developer Experience (DX) та інтеграції систем. Вивчити різницю між специфікацією OpenAPI та інструментами Swagger. Опанувати структуру YAML-файлів для опису ендпоінтів, схем даних та механізмів авторизації; навчитися генерувати інтерактивну документацію для Node.js/Express додатків; розглянути підходи "Code-First" та "Design-First".

1. API

1.1. Критична важливість документації API

API (Application Programming Interface) – це як контракт між сервером та клієнтом. Якщо контракт не задокументований або задокументований погано, інтеграція стає неможливою або перетворюється на метод спроб і помилок.

Хороша документація повинна відповідати на питання:

1. ***Куди*** відправляти запит (URL/Endpoint)?
2. ***Який метод*** використовувати (GET, POST, PUT, DELETE,…)?
3. ***Що*** потрібно відправити (Headers, Body, Params)?
4. ***Що*** поверне сервер у разі успіху або помилки (Response code, Body structure)?

1.2. OpenAPI Specification (OAS) в порівнянні з Swagger

Часто ці терміни вживають як синоніми, але в них є відмінність:

- ***OpenAPI Specification (OAS):*** Це відкритий стандарт опису RESTful API. Це формат файлу (JSON або YAML), який описує ваш API. Актуальна версія – 3.0/3.1.
- ***Swagger:*** Це набір інструментів (від компанії SmartBear) для роботи з OAS.
  - *Swagger Editor:* Редактор для написання специфікацій.
  - *Swagger UI:* Генератор красивої веб-сторінки з документацією на основі файлу специфікації.
  - *Swagger Codegen:* Генератор коду клієнта (SDK) або заглушки сервера на основі специфікації.

1.3. Підходи до розробки: Design-First в порівнянні з Code-First

Підхід	Code-First (Код спочатку)	Design-First (Дизайн спочатку)
Суть	Спочатку пишеться код контролерів, а документація генерується автоматично з коментарів або декораторів.	Спочатку пишеться файл специфікації (YAML), узгоджується з командою, і лише потім пишеться код.
Плюси	Швидко для розробника; документація завжди відповідає коду.	Паралельна робота фронтенду і бекенду; чіткий контракт до початку робіт.
Мінуси	Документація може бути "брудною"; фронтенд чекає на бекенд.	Потребує знання синтаксису OpenAPI; ризик розсинхронізації коду і документації.

2. Структура специфікації OpenAPI 3.0

Специфікація зазвичай пишеться у форматі **YAML**, оскільки він читабельніший за JSON.

2.1. Основні секції

1. ***openapi:*** Версія стандарту (наприклад, 3.0.0).
2. ***info:*** Метадані (назва API, версія, опис).
3. ***servers:*** Список базових URL (dev, prod).
4. ***paths:*** Опис маршрутів (ендпоінтів).
5. ***components:*** Перевикористовувані частини (схеми даних, параметри, відповіді).

2.2. Приклад опису одного маршруту (YAML)

```
openapi: 3.0.0
info:
  title: Todo API
  version: 1.0.0
paths:
  /todos:
    get:
      summary: Отримати список завдань
      responses:
        '200':
          description: Успішний запит
          content:
```

```
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Todo'
components:
  schemas:
    Todo:
      type: object
      properties:
        id:
          type: integer
        title:
          type: string
        completed:
          type: boolean
```

3. Практична інтеграція Swagger в Express.js (Code-First)

Для Node.js найпопулярнішим рішенням є комбінація бібліотек swagger-jsdoc (сканує коментарі JSDoc) та swagger-ui-express (відображає UI).

Крок 1: Встановлення

```
npm install swagger-jsdoc swagger-ui-express
```

Крок 2: Налаштування (index.js)

```
const express = require('express');
const swaggerJsDoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

const app = express();

const swaggerOptions = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'My Express API',
      version: '1.0.0',
      description: 'Документація для навчального API',
    },
    servers: [
      { url: 'http://localhost:3000' }
```

```
        ],
      },
      // Шляхи до файлів, де містяться анотації
      apis: ['./routes/*.js'],
    };

    const swaggerDocs = swaggerJsDoc(swaggerOptions);
    app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocs));

    app.listen(3000, () => console.log('Server running on port 3000'));
```

Крок 3: Написання документації в коді (JSDoc)

Створіть папку routes і файл todos.js. Додайте спеціальний коментар YAML перед маршрутом.

```
    /**
     * @swagger
     * components:
     * schemas:
     * Book:
     * type: object
     * required:
     * - title
     * - author
     * properties:
     * id:
     * type: string
     * description: Автоматично згенерований ID
     * title:
     * type: string
     * description: Назва книги
     * author:
     * type: string
     * description: Автор книги
     */

    /**
     * @swagger
     * /books:
     * get:
     * summary: Отримати список всіх книг
```

```
 * tags: [Books]
 * responses:
 * 200:
 * description: Список книг
 * content:
 * application/json:
 * schema:
 * type: array
 * items:
 * $ref: '#/components/schemas/Book'
 */
router.get('/', (req, res) => {
    // логіка...
});
```

4. Покращені теми документування

4.1. Авторизація (Security Schemes)

Опис того, як клієнт має передавати токен (наприклад, JWT).

```
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
security:
  - bearerAuth: []
```

4.2. Параметри запиту

Параметри можуть бути в:

- path (/users/{id})
- query (/users?role=admin)
- header (X-Custom-Header)
- cookie

Приклад опису path параметра:

```
parameters:
  - in: path
    name: id
    schema:
      type: integer
    required: true
    description: ID користувача
```

5. Контрольні запитання

1. У чому різниця між поняттями "OpenAPI" та "Swagger"?
2. Назвіть основні секції файлу специфікації OpenAPI.
3. Поясніть переваги підходу "Design-First". У яких випадках він кращий за "Code-First"?
4. Що таке Swagger UI і яку проблему він вирішує для фронтенд-розробників?
5. Як описати, що поле в JSON-об'єкті є обов'язковим (required)?
6. Для чого використовується секція components?
7. Що означає $ref у YAML-файлі специфікації?
8. Як задокументувати, що ендпоінт вимагає Bearer Token авторизації?
9. Чи можна за допомогою Swagger згенерувати код клієнта для Angular або React?
10. Як описати можливі помилки (400, 404, 500) для конкретного маршруту?

6. Завдання для самостійної роботи

1. ***Базова документація:*** Для API з попередньої лекції (Todo list або Users) і підключіть до нього swagger-ui-express. Опишіть методи GET (список) та GET (по ID).
2. ***Створення схеми:*** Опишіть у блоці components/schemas сутність User (поля: id, username, email, created_at). Використайте цю схему у відповідях.
3. ***POST запит:*** Задокументуйте метод створення користувача. Опишіть requestBody, вказавши, які поля обов'язкові для відправки. Перевірте роботу кнопки "Try it out" у браузері.
4. ***Swagger Editor:*** Зайдіть на *https://editor.swagger.io/*. Спробуйте написати YAML-специфікацію для API інтернет-магазину (товари, кошик) без прив'язки до реального коду (Design-First). Експортуйте результат у JSON.

ТЕМА 23. ЕЗПЕКА: ШИФРУВАННЯ ПАРОЛІВ, РОБОТА З JWT ТА CORS

Мета: Зрозуміти різницю між хешуванням та шифруванням, та чому паролі ніколи не можна зберігати у відкритому вигляді. Опанувати роботу з бібліотекою bcrypt для безпечного хешування. Вивчити стандарт JSON Web Token (JWT) для реалізації stateless-автентифікації. Розібратися з механізмом CORS (Cross-Origin Resource Sharing) та навчитися налаштовувати його на сервері Express.js. Ознайомитися з найкращими практиками захисту веб-додатків.

1. Безпека паролів

1.1. Головне правило безпеки

Ніколи не зберігайте паролі користувачів у відкритому вигляді (Plain text)! Якщо база даних буде скомпрометована (зламана), зловмисники отримають доступ до акаунтів користувачів не тільки на вашому сайті, а й на інших ресурсах, де люди часто використовують однакові паролі.

1.2. Шифрування та хешування

- ***Шифрування (Encryption):*** Двосторонній процес. Дані можна зашифрувати ключем і розшифрувати назад (наприклад, повідомлення в месенджері).
- ***Хешування (Hashing):*** Односторонній процес. Дані перетворюються на унікальний рядок фіксованої довжини (хеш). Відновити вихідний пароль з хешу математично неможливо.

1.3. Сіль (Salt) та атаки

Простого хешування (наприклад, MD5 або SHA-256) недостатньо, оскільки існують Rainbow Tables – величезні бази даних попередньо обчислених хешів для популярних паролів.

Для захисту використовується Сіль (Salt) – випадковий рядок, який додається до пароля перед хешуванням. Hash = Function(Password + Salt)

1.4. Алгоритм Bcrypt

На сьогодні стандартом індустрії є алгоритми, які працюють *повільно* (щоб ускладнити перебір brute-force), наприклад Bcrypt або Argon2.

Практичний приклад використання bcryptjs:

1. Встановлення: npm install bcryptjs
2. Хешування (при реєстрації):

```
const bcrypt = require('bcryptjs');

async function hashPassword(plainPassword) {
    const saltRounds = 10; // Фактор складності
    const hashedPassword = await bcrypt.hash(plainPassword, saltRounds);
    return hashedPassword;
}
```

3. Перевірка (при логіні):

```
async function verifyPassword(plainPassword, storedHash) {
    // Порівнює введений пароль із збереженим хешем
    const isMatch = await bcrypt.compare(plainPassword, storedHash);
    return isMatch; // true або false
}
```

2. JSON Web Token (JWT)

JWT – це відкритий стандарт (RFC 7519) для безпечної передачі даних між сторонами у вигляді JSON-об'єкта. Найчастіше використовується для авторизації.

2.1. Структура токена

JWT складається з трьох частин, розділених крапкою (.): Header.Payload.Signature

1. ***Header (Заголовок):*** Тип токена (JWT) та алгоритм підпису (наприклад, HS256).
2. ***Payload (Корисне навантаження):*** Дані про користувача (ID, роль, email) та службові поля (exp – час життя). *Важливо:* Не передавайте сюди конфіденційні дані (паролі), оскільки цю частину легко декодувати (Base64).
3. ***Signature (Підпис):*** Гарантує, що токен не був змінений. Створюється за допомогою секретного ключа, який знає тільки сервер.

2.2. Access Token та Refresh Token

- ***Access Token:*** Короткоживучий токен (15-30 хв), який використовується для доступу до захищених ресурсів API.
- ***Refresh Token:*** Довгоживучий токен (7-30 днів), який зберігається в БД і використовується для отримання нової пари токенів, коли Access Token закінчується.

2.3. Реалізація в Node.js (jsonwebtoken)

Встановлення: npm install jsonwebtoken

Генерація токена (Login):

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.JWT_SECRET; // Зберігати в .env!

function generateAccessToken(user) {
    const payload = {
        id: user.id,
        role: user.role
    };
    return jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
}
```

Middleware для перевірки токена: Цей код захищає маршрути, перевіряючи наявність валідного токена в заголовку Authorization.

```
function authenticateToken(req, res, next) {
    const authHeader = req.headers['authorization'];
    // Формат заголовка: "Bearer <TOKEN>"
    const token = authHeader && authHeader.split(' ')[1];

    if (!token) return res.sendStatus(401); // Unauthorized

    jwt.verify(token, SECRET_KEY, (err, user) => {
        if (err) return res.sendStatus(403); // Forbidden
(токен недійсний або прострочений)

        req.user = user; // Додаємо дані користувача в об'єкт
запиту
        next();
    });
}

// Використання:
```

```
app.get('/api/profile', authenticateToken, (req, res) => {
    res.json({ message: `Привіт, користувач з ID ${req.user.id}` });
});
```

3. CORS (Cross-Origin Resource Sharing)

3.1. Same-Origin Policy (SOP)

Браузери мають вбудований механізм безпеки, який забороняє сайту на одному домені (наприклад, http://localhost:3000) робити запити до API на іншому домені (http://localhost:5000), якщо сервер явно цього не дозволив.

3.2. Як працює CORS

Коли фронтенд робить запит на інший домен, браузер може спочатку відправити попередній запит (Preflight request) методом OPTIONS. Сервер повинен відповісти заголовками, що дозволяють доступ:

- Access-Control-Allow-Origin: Які домени мають доступ.
- Access-Control-Allow-Methods: Які методи (GET, POST...) дозволені.
- Access-Control-Allow-Headers: Які заголовки (Content-Type, Authorization) дозволені.

3.3. Налаштування в Express (cors)

Найпростіший спосіб – використати пакет cors.

Встановлення: npm install cors

Базове налаштування:

const cors = require('cors');

app.use(cors()); // Дозволяє ВСІМ доменам (небезпечно для продакшну)

Безпечне налаштування (Whitelist):

```
const corsOptions = {
    origin: 'http://localhost:3000', // Дозволити тільки нашому фронтенду
    optionsSuccessStatus: 200 // Для старих браузерів
};

app.use(cors(corsOptions));
```

4. Найкращі практики безпеки

1. ***HTTPS:*** Завжди використовуйте HTTPS. Без нього токени та паролі передаються відкритим текстом і можуть бути перехоплені (Man-in-the-Middle attack).

2. ***Зберігання токенів:***
   - *LocalStorage:* Зручно, але вразливо до XSS атак (шкідливий JS може вкрасти токен).
   - *HttpOnly Cookies:* Безпечніше, бо JS не має до них доступу, але потребує захисту від CSRF.
3. ***Змінні оточення:*** Ніколи не комітьте секретні ключі (JWT_SECRET, паролі БД) у Git. Використовуйте файл .env та бібліотеку dotenv.
4. ***Rate Limiting:*** Обмежуйте кількість запитів з однієї IP-адреси, щоб захиститися від Brute-force атак та DDoS.

5. Контрольні запитання

1. Чому хешування є односторонньою операцією? Чим це відрізняється від шифрування?
2. Що таке "Сіль" (Salt) і від якого типу атак вона захищає?
3. Чому не можна зберігати паролі в MD5?
4. З яких трьох частин складається JWT токен? Яка з них відповідає за цілісність даних?
5. Що станеться, якщо зловмисник змінить Payload у JWT токені, але не оновить підпис?
6. Поясніть різницю між помилками 401 (Unauthorized) та 403 (Forbidden) в контексті JWT.
7. Навіщо потрібен Refresh Token, якщо можна просто зробити Access Token "вічним"?
8. Що таке CORS і чому браузери блокують запити до інших доменів за замовчуванням?
9. Який HTTP метод використовується для Preflight запитів CORS?
10. Де безпечніше зберігати JWT токен на клієнті: в LocalStorage чи в HttpOnly Cookies? Аргументуйте.

6. Завдання для самостійної роботи

1. ***Автентифікація:*** Створіть Express-сервер з двома ендпоінтами: /register та /login.
   - /register: Приймає email/password, хешує пароль через bcryptjs і зберігає користувача (можна в масив).
   - /login: Перевіряє пароль і повертає JWT токен.

2. ***Захищений ресурс:*** Створіть ендпоінт /me, який повертає дані профілю користувача тільки при наявності валідного токена.
3. ***CORS тест:*** Створіть просту HTML-сторінку (окремо від сервера), яка намагається зробити fetch запит до вашого API. Налаштуйте CORS на сервері так, щоб спочатку заблокувати, а потім дозволити цей запит.
4. ***Декодер:*** Напишіть функцію на клієнті, яка декодує payload JWT токена (без бібліотек, використовуючи atob), щоб дізнатися термін дії токена (exp).

ТЕМА 24. WEBSOCKET: РЕАЛІЗАЦІЯ ОБМІНУ ДАНИМИ В РЕАЛЬНОМУ ЧАСІ

Мета: Ознайомитися з обмеженнями протоколу HTTP для інтерактивних додатків; вивчити принцип роботи протоколу WebSocket та процес "рукостискання" (Handshake). Навчитися реалізовувати сервер на Node.js за допомогою бібліотеки ws та Socket.io. опанувати роботу з клієнтським API для обміну повідомленнями. Зрозуміти різницю між подіями, кімнатами (rooms) та широкомовними розсилками (broadcast).

1. Призначення WebSocket

1.1. Еволюція взаємодії клієнта та сервера

Традиційний веб побудований на моделі Request-Response (Запит-Відповідь) протоколу HTTP:

1. Клієнт відправляє запит.
2. Сервер обробляє його.
3. Сервер повертає відповідь і закриває з'єднання.

Це створює проблеми для додатків реального часу (чати, біржі, сповіщення), де дані на сервері можуть оновитися в будь-який момент.

1.2. Старі методи імітації реального часу

До появи WebSocket використовували:

- ***Short Polling (Коротке опитування):*** Клієнт кожні 2-5 секунд надсилає HTTP-запит, питаючи: "Чи є нові дані?". Це створює величезне навантаження на сервер.
- ***Long Polling (Довге опитування):*** Сервер тримає запит відкритим, поки не з'являться дані. Як тільки дані надіслано, клієнт одразу відкриває новий запит.

1.3. Протокол WebSocket

WebSocket (WS) – це протокол зв'язку поверх TCP, що забезпечує постійне, двостороннє (Full-Duplex) з'єднання.

- ***Повнодуплексність:*** Обидві сторони можуть надсилати дані одночасно.
- ***Низькі затримки:*** Не потрібно перевідкривати з'єднання для кожного повідомлення.
- ***Економічність:*** Заголовки пакетів значно менші, ніж в HTTP.

2. Процес встановлення з'єднання (Handshake)

З'єднання починається з HTTP-запиту, який повідомляє серверу про бажання змінити протокол.

Запит клієнта:

1. GET /chat HTTP/1.1
2. Host: example.com
3. Upgrade: websocket
4. Connection: Upgrade
5. Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Відповідь сервера:

6. HTTP/1.1 101 Switching Protocols
7. Upgrade: websocket
8. Connection: Upgrade

Після отримання статусу 101, канал залишається відкритим, і сторони переходять на бінарний протокол WebSocket.

3. Практична реалізація на Node.js

Розглянемо два підходи: "чистий" ws та популярну бібліотеку Socket.io.

3.1. Використання бібліотеки ws

Це мінімалістична реалізація для тих, кому потрібен чистий протокол.

Серверна частина (server.js):

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
    console.log('Новий клієнт підключився');

    ws.on('message', (message) => {
        console.log(`Отримано: ${message}`);

        // Відправка відповіді клієнту
        ws.send(`Сервер отримав ваше: ${message}`);
    });

    ws.on('close', () => {
```

```
        console.log('Клієнт відключився');
    });
});
```

3.2. Використання Socket.io

Socket.io – це не просто WebSocket, а надбудова, яка забезпечує:

- Автоматичне перепідключення.
- Підтримку "кімнат" (Rooms).
- Відправку об'єктів без ручного JSON.stringify.

Сервер з Socket.io:

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

io.on('connection', (socket) => {
    console.log('User connected:', socket.id);

    // Слухаємо кастомну подію
    socket.on('chat message', (msg) => {
        // Розсилаємо всім підключеним клієнтам (Broadcast)
        io.emit('chat message', msg);
    });
});

server.listen(3000, () => {
    console.log('Сервер запущено на порту 3000');
});
```

4. Клієнтська частина (Frontend)

Браузери мають вбудований об'єкт WebSocket для роботи з нативним протоколом.

4.1. Нативний браузерний API

```
const socket = new WebSocket('ws://localhost:8080');

// Коли з'єднання відкрито
socket.onopen = () => {
```

```
    console.log('З'єднано з сервером');
    socket.send('Привіт, сервер!');
};

// Обробка вхідних повідомлень
socket.onmessage = (event) => {
    console.log('Повідомлення від сервера:', event.data);
};

// Обробка помилок
socket.onerror = (error) => {
    console.error('Помилка:', error);
};
```

4.2. Робота з Socket.io на клієнті

Для роботи з Socket.io потрібно підключити спеціальний клієнтський скрипт.

```
<script src="/socket.io/socket.io.js"></script>
<script>
    const socket = io(); // Автоматично підключається до хоста

    function sendMessage() {
        const text = document.getElementById('messageInput').value;
        socket.emit('chat message', text); // Відправляємо іменовану подію
    }

    socket.on('chat message', (msg) => {
        const item = document.createElement('li');
        item.textContent = msg;
        document.getElementById('messages').appendChild(item);
    });
</script>
```

5. Поняття широкомовлення та кімнат

Однією з найпотужніших функцій WebSocket-бібліотек є керування потоками даних.

1. ***Direct Message:*** Відправка повідомлення конкретному сокету.
   - `socket.emit('event', data)` – тільки відправнику.
2. ***Broadcast:*** Відправка всім, крім відправника.

```
socket.broadcast.emit('event', data)
```

3. ***Rooms (Кімнати):*** Логічне групування користувачів (наприклад, "Чат техпідтримки" або "Кімната гри №5").
4. ***Сервер:*** Додавання в кімнату
```
socket.join('room1');
```
5. ***Сервер:*** Відправка всім у room1
```
io.to('room1').emit('new message', 'Привіт всім у цій кімнаті!');
```

6. Безпека та масштабування

- ***Автентифікація:*** Передавайте JWT токен у параметрах запиту при підключенні або через спеціальне поле в Socket.io middleware.
- ***Валідація:*** Сервер повинен перевіряти всі вхідні повідомлення. Не довіряйте клієнту.
- ***Масштабування:*** Оскільки WebSocket тримає з'єднання в пам'яті, один сервер може не впоратися. Для роботи на кількох серверах використовують Redis Adapter, щоб повідомлення, надіслане на Сервер А, дійшло до клієнта, підключеного до Сервера Б.

7. Контрольні запитання

1. У чому полягає основна відмінність між HTTP та WebSocket?
2. Що таке "Upgrade header" і яку роль він відіграє у встановленні з'єднання?
3. Поясніть термін "Full-Duplex" стосовно мережевої взаємодії.
4. Чому метод Short Polling вважається неефективним?
5. Які переваги надає бібліотека Socket.io порівняно з нативним модулем ws?
6. Як реалізувати відправку повідомлення всім користувачам одночасно (Broadcasting)?
7. Що таке "Rooms" у Socket.io і наведіть приклад їх реального застосування.
8. Чи можна використовувати WebSocket без HTTP? Обґрунтуйте відповідь.
9. Як забезпечити безпеку WebSocket-з'єднання?
10. Як браузер розуміє, що сервер підтримує WebSocket під час процедури Handshake?

8. Завдання для самостійної роботи

1. ***Ехо-сервер:*** Напишіть сервер на ws, який повертає клієнту надісланий ним текст у верхньому регістрі (UPPERCASE).
2. ***Груповий чат:*** Використовуючи Socket.io, створіть чат, де користувач може вводити своє ім'я та бачити список повідомлень від усіх учасників.
3. ***Індикатор друку:*** Реалізуйте функцію "Користувач X пише повідомлення...", яка спрацьовує при події oninput у текстовому полі на клієнті та транслюється іншим учасникам.
4. ***Сенсорні дані:*** Напишіть сторінку, яка відправляє координати курсора миші на сервер через WebSocket, а сервер транслює ці координати іншому підключеному клієнту, який відображає рух "чужого" курсора на екрані.

ТЕМА 25. ДЕПЛОЙ: ОСНОВИ DOCKER ТА КОНТЕЙНЕРИЗАЦІЯ ВЕБ-ЗАСТОСУНКІВ

Мета: Зрозуміти різницю між віртуалізацією та контейнеризацією. Вивчити архітектуру Docker та його основні компоненти (Engine, Images, Containers); навчитися створювати Dockerfile для автоматизації збірки образів. Опанувати роботу з Docker Compose для запуску багатоконтейнерних застосунків. Розглянути кращі практики підготовки веб-застосунків до деплою в ізольованих середовищах.

1. Поняття контейнеризації

1.1. Проблема "It works on my machine"

У традиційній розробці виникає конфлікт версій бібліотек, операційних систем або конфігурацій між комп'ютером розробника та сервером. Контейнеризація вирішує цю проблему, пакуючи код разом із усім оточенням (runtime, бібліотеки, системні інструменти).

1.2. Віртуальні машини (VM) в порівнянні з контейнерами

- ***Віртуальні машини:*** Кожна VM включає повну копію операційної системи, віртуальні драйвери та додаток. Це споживає багато ресурсів (ГБ пам'яті) і довго завантажується.
- ***Контейнери:*** Використовують ядро основної (хостової) ОС, але ізолюють процеси на рівні користувача. Вони легкі (МБ), запускаються за секунди та споживають мінімум ресурсів.

1.3. Docker

Docker – це платформа з відкритим вихідним кодом, яка автоматизує розгортання додатків у легких, портативних контейнерах.

2. Основні поняття та архітектура Docker

Для роботи з Docker важливо розрізняти три ключові терміни:

1. ***Dockerfile:*** Текстовий файл-інструкція, за якою збирається образ.
2. ***Image (Образ):*** "Зліпок" або шаблон додатка, що включає код та залежності (схоже на інсталяційний диск).
3. ***Container (Контейнер):*** Запущений екземпляр образу. Це ізольований процес, у якому виконується додаток.

4. ***Docker Hub:*** Публічний реєстр (хмарне сховище), де зберігаються готові образи (Node.js, Python, MySQL тощо).

3. Практична робота з Dockerfile

Dockerfile – це фундамент контейнеризації. Кожна команда в ньому створює новий "шар" (layer) образу.

Приклад: Контейнеризація Node.js додатка

Уявімо, що у нас є веб-сервер. Створимо файл з назвою Dockerfile у корені проєкту:

```
# 1. Вибираємо базовий образ (офіційний Node.js)
FROM node:18-alpine

# 2. Встановлюємо робочу директорію всередині контейнера
WORKDIR /app

# 3. Копіюємо файли залежностей
COPY package*.json ./

# 4. Встановлюємо бібліотеки
RUN npm install

# 5. Копіюємо решту вихідного коду
COPY . .

# 6. Вказуємо порт, який буде прослуховувати додаток
EXPOSE 3000

# 7. Команда для запуску додатка
CMD ["node", "index.js"]
```

Основні команди Docker CLI:

- docker build -t my-web-app . – збірка образу з поточної директорії.
- docker run -p 8080:3000 my-web-app – запуск контейнера (прокидання порту 8080 хоста на 3000 контейнера).
- docker ps – список запущених контейнерів.
- docker images – список усіх завантажених образів.

4. Багатоконтейнерні застосунки: Docker Compose

Сучасні веб-застосунки зазвичай складаються з кількох частин: Frontend, Backend та База даних. Замість того, щоб запускати кожен контейнер окремо, використовують Docker Compose.

Конфігурація описується у файлі docker-compose.yml:

```
version: '3.8'

services:
  # Наш веб-додаток
  web:
    build: .
    ports:
      - "8080:3000"
    depends_on:
      - db
    environment:
      - DB_URL=mongodb://db:27017/myapp

  # База даних MongoDB
  db:
    image: mongo:latest
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

volumes:
  mongodb_data:
```

Керування через Compose:

- docker-compose up – запуск усієї інфраструктури однією командою.
- docker-compose down – зупинка та видалення всіх контейнерів проєкту.

5. Робота з даними та мережею

5.1. Volumes (Томи)

Контейнери за своєю природою є *ephemeral* (тимчасовими). Якщо видалити контейнер бази даних, усі дані всередині зникнуть. Щоб зберегти дані, використовують *Volumes* – вони монтують папку на реальному диску сервера до папки всередині контейнера.

5.2. Networking

Docker створює віртуальну мережу для кожного проєкту. Всередині цієї мережі контейнери можуть звертатися один до одного за іменами сервісів (наприклад, додаток звертається до бази просто за адресою db, а не за IP).

6. Життєвий цикл деплою (CI/CD контекст)

Процес розгортання за допомогою Docker зазвичай виглядає так:

1. Розробник пише код та Dockerfile.
2. Система автоматизації (GitHub Actions/GitLab CI) збирає образ.
3. Образ тестується в ізольованому середовищі.
4. Готовий образ завантажується в Container Registry (Docker Hub / AWS ECR).
5. На продакшн-сервері виконується команда docker pull та оновлення контейнера.

7. Контрольні запитання

1. Яка головна перевага контейнеризації перед запуском додатка безпосередньо в ОС?
2. Поясніть різницю між поняттями Image та Container.
3. Для чого потрібна команда EXPOSE у Dockerfile?
4. Чим відрізняється команда RUN від CMD у Dockerfile?
5. Що таке Docker Hub і яку роль він відіграє в екосистемі?
6. Для чого використовується Docker Compose? У яких випадках він незамінний?
7. Як забезпечити збереження даних бази даних, якщо контейнер з нею буде видалено?
8. Що таке "прокидання портів" (Port mapping) і як воно працює?
9. Чому рекомендується використовувати образи на базі Alpine Linux (наприклад, node:alpine)?
10. Як Docker допомагає синхронізувати роботу в команді розробників?

8. Завдання для самостійної роботи

1. ***Базовий Dockerfile:*** Створіть просту HTML-сторінку та напишіть Dockerfile на базі образу nginx:alpine, щоб запустити цю сторінку всередині контейнера. Перевірте роботу в браузері.

2. ***Оптимізація образу:*** Спробуйте змінити порядок команд у Dockerfile для Node.js так, щоб команда npm install не перетворювалася в кеш при кожній зміні вихідного коду (використовуйте принцип Layer Caching).
3. ***Docker Compose та DB:*** Спробуйте розгорнути локально WordPress або іншу CMS за допомогою готового docker-compose.yml файлу, знайденого в офіційній документації.
4. ***Аналіз шарів:*** Використайте команду docker history <image_name>, щоб побачити, скільки місця займає кожна команда вашого Dockerfile.

Список рекомендованої літератури

1. W3C. HTML5.3 Specification. W3C Recommendation. World Wide Web Consortium, 2021. – [Електронний ресурс]. Режим доступу: https://www.w3.org/
2. ECMA International. ECMAScript® 2023 Language Specification. Standard ECMA-262, 14th Edition. 2023. – [Електронний ресурс]. Режим доступу: https://ecma-international.org/wp-content/uploads/ECMA-262_14th_edition_june_2023.pdf
3. HTML Підручник – W3schoolsUA.українською – [Електронний ресурс]. Режим доступу: https://w3schoolsua.github.io/html/index.html#gsc.tab=0
4. CSS Підручник – [Електронний ресурс]. Режим доступу: https://w3schoolsua.github.io/css/index.html#gsc.tab=0
5. Сучасний підручник з JavaScript – [Електронний ресурс]. Режим доступу: https://uk.javascript.info
6. JavaScript Підручник. Основи вебпрограмування – [Електронний ресурс]. Режим доступу: https://w3schoolsua.github.io/js/index.html#gsc.tab=0
7. MDN Web Docs (Mozilla Developer Network) [Електронний ресурс]. Режим доступу: developer.mozilla.org
8. React Documentation (Beta/Latest) [Електронний ресурс]. Режим доступу: react.dev
9. Next.js Documentation [Електронний ресурс]. Режим доступу: nextjs.org/docs
10. Refactoring Guru (Патерни проєктування) [Електронний ресурс]. Режим доступу: refactoring.guru/uk