Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Матвій О.В.
Мельник В.С.
Мельник Г.В.
Перцов А.С.

Фреймворки JavaScript

Методичні рекомендації та
завдання для лабораторних робіт

Чернівці
Чернівецький національний університет
2026

Укладачі:
Олександр Матвій, кандидат фізико-математичних наук, доцент кафедри математичного моделювання;
Василь Мельник, кандидат фізико-математичних наук, асистент кафедри математичного моделювання;
Галина Мельник, кандидат економічних наук, доцент кафедри прикладної математики;
Андрій Перцов, кандидат фізико-математичних наук, доцент кафедри математичного моделювання.

Т384 Фреймворки JavaScript: методичні рекомендації та завдання для лабораторних робіт.. Укл.: Олександр Матвій, Василь Мельник, Галина Мельник, Андрій Перцов – Чернівці: Чернівецький національний університет, 2026. – 79 с.

Методичні рекомендації містять завдання для лабораторних робіт та теоретичні відомості, необхідні для їх виконання, з дисципліни «Фреймворки JavaScript».

Для студентів, що здобувають освіту в галузі знань „Інформаційні технології” для спеціальностей 122 - “Комп’ютерні науки”, 124 - “Системний аналіз”, 113 - ”Прикладна математика” та інших спеціальностей, що пов'язані із вивченням комп'ютерних інформаційних технологій. УДК 004.65 (076.5)

Лабораторна робота №1: Bootstrap. 4
Лабораторна робота №2: Робота з DOM. JQuery. 10
Лабораторна робота №3: CRUD-додаток на Angular (Standalone + Signals + Reactive Forms + HTTP + NgRx) 20
Очікувані результати навчання 22
Передумови та середовище 22
Початкові команди 23
Завдання (етапи) 24
Доменна область і дані 34
Використайте ChangeDetectionStrategy.OnPush 35
Використайте @for або trackBy у списках 35
Використайте @defer для великих віджетів 36
Увімкніть event coalescing при старті додатку 36
Напишіть модульні тести для ProductsService з використанням HTTP-моків 36
Звіт і критерії оцінювання 37
Порядок здачі 37
Лабораторна робота 4. React 38
Лабораторна робота 5. Node Backend 54

Лабораторна робота №1: Bootstrap.

Bootstrap — це популярний безкоштовний фронтенд-фреймворк з відкритим вихідним кодом, призначений для швидкого та зручного створення сучасних веб-сайтів і веб-застосунків. Він надає готовий набір CSS-стилів, HTML-шаблонів та JavaScript-компонентів, які дозволяють розробникам зосередитися на логіці застосунку, а не на базовій верстці інтерфейсу.

Основною перевагою Bootstrap є наявність стандартизованих компонентів інтерфейсу користувача, таких як кнопки, форми, таблиці, навігаційні панелі, модальні вікна, сповіщення, каруселі та інші елементи. Всі ці компоненти мають узгоджений зовнішній вигляд і поведінку, що дозволяє створювати візуально цілісні інтерфейси без необхідності писати значні обсяги власного CSS-коду.

Однією з ключових можливостей Bootstrap є адаптивна (responsive) верстка. Фреймворк використовує сіткову систему (grid system), побудовану на принципі поділу сторінки на колонки, що автоматично підлаштовуються під розмір екрану. Завдяки цьому веб-застосунки коректно відображаються як на настільних комп’ютерах, так і на планшетах чи мобільних пристроях без додаткових зусиль з боку розробника.

Bootstrap є клієнтським (frontend) фреймворком, тобто він працює на стороні браузера та відповідає за відображення і взаємодію з користувачем. На відміну від серверних технологій, таких як Node.js або ASP.NET, Bootstrap не виконує бізнес-логіку і не працює з базами даних, а використовується для побудови зовнішнього вигляду та поведінки інтерфейсу користувача.

До складу Bootstrap також входять JavaScript-плагіни, які забезпечують інтерактивність компонентів: відкриття модальних вікон, випадаючі меню, вкладки, підказки (tooltips) та інші динамічні елементи. У сучасних версіях Bootstrap ці плагіни можуть працювати без використання бібліотеки jQuery, що робить фреймворк легшим та більш сумісним із сучасними JavaScript-фреймворками.

Bootstrap широко використовується разом з такими фреймворками, як Angular, React та Vue, де він виконує роль базового стилістичного шару інтерфейсу. Завдяки модульній структурі та можливості перевизначення стилів розробники можуть адаптувати зовнішній вигляд компонентів під власні потреби, не порушуючи загальної архітектури застосунку.

Таким чином, Bootstrap є потужним інструментом для швидкої розробки веб-інтерфейсів, який поєднує простоту використання, адаптивність і багатий набір готових компонентів. Його застосування особливо корисне у навчальних, прототипних та комерційних проєктах, де важливими є швидкість розробки та узгоджений дизайн інтерфейсу.

Виконання лабораторної.

1. Виберіть тематику своєї веб-сторінки. Тематика визначає загальний стиль, структуру та зміст майбутнього веб-застосунку, тому до цього етапу слід підійти усвідомлено. Це може бути особистий блог для публікації статей та новин, сторінка студентського парламенту з інформацією про заходи, оголошення та членів команди, або веб-сторінка, присвячена вашому хобі (наприклад, спорт, музика, програмування, фотографія тощо). Обрана тематика повинна бути вам цікавою, оскільки це спростить наповнення сторінки контентом і допоможе краще продумати логіку інтерфейсу та навігації. На цьому етапі також варто приблизно уявити, яку інформацію ви будете розміщувати на сторінці та якими елементами інтерфейсу вона буде наповнена.

2. Додайте до своєї HTML-сторінки необхідні теги для підключення стилів і скриптів Bootstrap версії 5.3.8 за допомогою Content Delivery Network (CDN). Використання CDN дозволяє швидко підключити Bootstrap без попереднього завантаження файлів у проєкт, оскільки всі ресурси підвантажуються з віддалених серверів. Це спрощує початковий етап розробки та зменшує кількість налаштувань.

У тезі <head> необхідно підключити CSS-файл Bootstrap, який відповідає за стилі, сітку, типографіку та зовнішній вигляд компонентів інтерфейсу:

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-sRIl4kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLzeN8tE5Y
BujZqJLB" crossorigin="anonymous">
```

Цей тег забезпечує коректне відображення кнопок, форм, таблиць, навігаційних панелей та інших елементів Bootstrap одразу після завантаження сторінки.

В кінці блоку <body> необхідно підключити JavaScript-бандл Bootstrap, який містить як власні скрипти фреймворку, так і залежності (зокрема Popper). Це потрібно для коректної роботи інтерактивних компонентів, таких як випадаючі меню, модальні вікна, каруселі, тултіпи тощо:

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle
.min.js"
integrity="sha384-FKyoEForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVcXK/BmnVDxM+D2
scQbITxI" crossorigin="anonymous"></script>
```

Розміщення цього скрипта наприкінці <body> рекомендується для того, щоб HTML-контент сторінки завантажувався швидше, а JavaScript не блокував процес рендерингу.

Альтернативно, ви можете встановити Bootstrap як npm-пакет за допомогою команди:

npm i bootstrap

Такий підхід зазвичай використовується в більш складних проєктах, де застосовується система збірки (наприклад, Webpack, Vite або Parcel). У цьому випадку Bootstrap підключається локально з папки node_modules, що дає змогу краще контролювати версії бібліотеки, кастомізувати стилі та інтегрувати Bootstrap у сучасний frontend-пайплайн.

3. Перше, що ми хочемо зробити, це додати панель навігації у верхній частині сторінки. Для цього ми будемо використовувати клас navbar. Це один із елементів за замовчуванням Bootstrap. Він створює елемент навігації, який реагує за замовчуванням і автоматично згортається на

менших екранах. Він також пропонує вбудовану підтримку для додавання брендингу, колірних схем, інтервалів та інших компонентів.:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Мій сайт</a>
    <div class="collapse navbar-collapse">
      <ul class="navbar-nav">
        <li class="nav-item"><a class="nav-link active"
href="#">Головна</a></li>
        <li class="nav-item"><a class="nav-link" href="#">Про нас</a></li>
      </ul>
    </div>
  </div>
</nav>
```

4. Додайте на свою веб-сторінку приклад використання grid-системи Bootstrap. Grid-система є однією з ключових можливостей цього фреймворку і призначена для створення адаптивних макетів, які коректно відображаються на різних пристроях — від мобільних телефонів до великих екранів настільних комп'ютерів. У наведеному нижче прикладі використовується контейнер `.container`, який забезпечує правильні відступи від країв екрана та вирівнює вміст сторінки по центру. Всередині контейнера розташований елемент `.row`, що представляє один ряд сітки. Кожен ряд може містити до 12 колонок.

```
<div class="container">
  <div class="row">
    <div class="col-md-4 bg-light">1</div>
    <div class="col-md-4 bg-secondary text-white">2</div>
    <div class="col-md-4 bg-light">3</div>
  </div>
</div>
```

Класи `col-md-4` означають, що кожен блок займає 4 колонки з 12 можливих на екранах середнього розміру (`md`) і більших. Таким чином, у рядку розміщуються три рівні за шириною колонки. На менших екранах (наприклад, на смартфонах) ці блоки автоматично розташуються один під одним, що забезпечує адаптивність макета без додаткового коду.

Додаткові класи `bg-light`, `bg-secondary` та `text-white` використовуються для надання фонових кольорів і кращої візуальної наочності прикладу. У реальних проєктах замість простих чисел у колонках ви можете розміщувати текст, зображення, форми, картки або будь-які інші компоненти Bootstrap.

Таким чином, використовуючи grid-систему Bootstrap, ви можете легко будувати складні та гнучкі макети сторінок, мінімізуючи кількість власних CSS-стилів.

5. Додайте на свою веб-сторінку кілька кнопок, використовуючи готові стилі Bootstrap. Кнопки є одним із найбільш уживаних елементів інтерфейсу, оскільки вони використовуються для виконання дій: надсилання форм, переходу між сторінками, підтвердження або скасування операцій тощо.Bootstrap надає набір стандартних класів для кнопок, які визначають їхній вигляд, колір та поведінку. Базовим класом є `btn`, а додаткові класи задають стиль кнопки залежно від її призначення.

```
<button class="btn btn-primary">Основна</button>
<button class="btn btn-success">Успіх</button>
<button class="btn btn-danger">Помилка</button>
<button class="btn btn-outline-dark">Контур</button>
```

Клас `btn-primary` зазвичай використовується для головної дії на сторінці. Кнопка з класом `btn-success` сигналізує про успішну дію, наприклад, збереження даних. Клас `btn-danger` застосовується для небезпечних або незворотних операцій, таких як видалення записів. Варіант `btn-outline-dark` створює кнопку з контуром без суцільного фону, що часто використовується для другорядних дій.

За потреби ви можете змінювати розмір кнопок (`btn-sm`, `btn-lg`), вимикати їх (`disabled`), групувати кілька кнопок разом або використовувати кнопки у вигляді посилань. Це дозволяє легко створювати зручний та зрозумілий інтерфейс користувача без написання додаткових CSS-стилів.

6. Додайте на сторінку картки (Cards) — універсальний компонент Bootstrap, який використовується для відображення структурованої інформації, наприклад товарів, новин, статей або профілів користувачів. Картка зазвичай складається з контейнера, зображення, заголовка, текстового опису та елементів керування (кнопок або посилань). Вона допомагає логічно згрупувати інформацію та зробити інтерфейс більш наочним і зручним для користувача.

```
<div class="card" style="width: 18rem;">
  <img src="https://via.placeholder.com/150" class="card-img-top"
alt="Фото">
  <div class="card-body">
    <h5 class="card-title">Назва товару</h5>
    <p class="card-text">Короткий опис товару.</p>
    <a href="#" class="btn btn-primary">Купити</a>
  </div>
</div>
```

Клас `card` створює основний контейнер картки, а `card-img-top` використовується для зображення, яке розміщується у верхній частині. Усередині блоку `card-body` зазвичай розміщують заголовок (`card-title`), опис (`card-text`) та кнопки або посилання для взаємодії з користувачем.

Картки можна легко комбінувати з grid-системою Bootstrap, щоб виводити кілька елементів у ряд або створювати адаптивні списки товарів. Також підтримуються додаткові елементи, такі як `card-footer`, `card-header`, списки (`list-group`) та різні варіанти оформлення, що дозволяє використовувати картки у найрізноманітніших сценаріях веб-розробки.

7. Додайте на сторінку форму для введення даних користувача. Форми є одним з ключових елементів будь-якого веб-застосунку, оскільки саме через них користувачі взаємодіють із системою: реєструються, виконують вхід, надсилають повідомлення або вводять іншу інформацію. Bootstrap надає готові класи для стилізації форм, які забезпечують акуратний вигляд, зручність введення даних та адаптивність на різних пристроях. Клас `form-control` використовується для текстових полів, а `form-label` — для підписів до них. Відступи між елементами форми задаються за допомогою класів відступів, наприклад `mb-3`.

```
<form class="p-3">
```

```
  <div class="mb-3">
    <label class="form-label">Email:</label>
    <input type="email" class="form-control" placeholder="Введіть email">
  </div>
  <div class="mb-3">
    <label class="form-label">Пароль:</label>
    <input type="password" class="form-control">
  </div>
  <button type="submit" class="btn btn-success">Увійти</button>
</form>
```

Поле з типом `email` автоматично виконує базову перевірку коректності введеної електронної адреси, а тип `password` приховує символи пароля. Кнопка з класом `btn btn-success` використовується для надсилання форми.

За потреби ви можете додавати інші елементи форми, такі як чекбокси, радіокнопки, випадаючі списки або повідомлення про помилки валідації. Це дозволяє створювати повноцінні та зручні інтерфейси для введення даних без написання власних стилів CSS.

8. На цьому етапі наповніть веб-сторінку змістом відповідно до обраної вами тематики. Контент повинен логічно відповідати ідеї сайту та демонструвати використання основних компонентів Bootstrap, які ви додали на попередніх кроках.

Розмістіть текстові блоки, заголовки, зображення, картки, кнопки та форми так, щоб сторінка виглядала цілісною та зрозумілою для користувача. Наприклад, для особистого блогу це можуть бути короткі статті або описи публікацій, для сторінки студентського парламенту — інформація про події, учасників та контакти, а для сайту про хобі — галерея робіт або опис улюблених занять.

Зверніть увагу на читабельність тексту, структуру сторінки та візуальну ієрархію елементів. Використовуйте заголовки різних рівнів (`h1`–`h6`), абзаци, списки та відступи, щоб контент був зручним для сприйняття. Метою цього пункту є створення завершеної веб-сторінки, яка не лише коректно працює технічно, але й має зрозумілий, логічний та привабливий вигляд.

9. Після завершення розробки завантажте вихідний код вашого проєкту до власного репозиторію на GitHub. Для цього створіть новий репозиторій, ініціалізуйте git у папці з проєктом та виконайте коміти з усіма необхідними файлами. Переконайтеся, що структура проєкту зрозуміла, а в репозиторії відсутні зайві або тимчасові файли.

Далі задеплойте вашу веб-сторінку за допомогою GitHub Pages. У налаштуваннях репозиторію (Settings → Pages) оберіть гілку, з якої буде виконуватись публікація (зазвичай `main` або `master`), та папку з файлами сайту (кореневу або `docs`). Після збереження налаштувань GitHub згенерує публічне посилання на вашу сторінку.

Перейдіть за отриманою лінкою деплою та перевірте коректність роботи сайту: чи відображається контент, чи підключилися стилі Bootstrap, чи правильно працюють кнопки, картки та форми. У разі помилок перевірте шляхи до файлів, підключення CDN або налаштування репозиторію. Результатом цього етапу має бути доступна в інтернеті веб-сторінка, яку можна відкрити з будь-якого пристрою.

Лабораторна робота №2: Робота з DOM. JQuery.

DOM: Модель об'єктів документа, структуроване представлення веб-сторінки, організоване у вигляді деревоподібної структури. DOM представляє все, що міститься на веб-сторінці: HTML-елементи (такі як `<div>`, `<p>`, `<button>`), їхній вміст і навіть їхні взаємовідносини між собою (наприклад, відносини «батько-дитина» або «брат-брат»). DOM дозволяє нам програмно взаємодіяти з нашою веб-сторінкою. За допомогою JavaScript (або бібліотеки, такої як jQuery) ми можемо отримувати доступ, змінювати, додавати та видаляти елементи.

jQuery — це швидка, компактна та багатофункціональна бібліотека JavaScript, призначена для спрощення маніпуляцій з HTML DOM, обробки подій, анімації та взаємодії AJAX. Створена в 2006 році Джоном Ресігом, jQuery швидко стала однією з найпопулярніших бібліотек JavaScript, оскільки вона дозволяла розробникам писати менше коду для досягнення більших результатів, особливо в питаннях сумісності між браузерами та складних функцій JavaScript.

Сумісність з різними браузерами: jQuery спочатку було створено для стандартизації поведінки JavaScript у різних браузерах, що зробило його надійним інструментом для написання коду, який працює в Chrome, Firefox, Safari та Internet Explorer.

Спрощена синтаксис: Завдяки лаконічній синтаксису jQuery дозволяє розробникам з легкістю виконувати складні завдання, такі як маніпуляції з DOM і AJAX-запити.

Широка екосистема плагінів: jQuery має велику екосистему плагінів і розширень, що дозволяє легко додавати такі функції, як слайдери, вибір дати і перевірка форм.

Виконання лабораторної

1. На першому етапі лабораторної роботи необхідно підключити бібліотеку jQuery до вашої веб-сторінки за допомогою Content Delivery Network (CDN). jQuery — це популярна JavaScript-бібліотека, яка значно

спрощує роботу з DOM-елементами, обробку подій, а також виконання асинхронних запитів. Використання CDN дозволяє швидко підключити бібліотеку без необхідності завантаження файлів у ваш проєкт і забезпечує високу швидкість завантаження за рахунок кешування на серверах користувачів. Для підключення jQuery додайте наступний тег `<script>` у HTML-документ. Рекомендується розміщувати його перед закриваючим тегом `</body>`, щоб сторінка спочатку завантажила HTML-розмітку, а вже потім підключала JavaScript-код.

```
<script
  src="
  https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js">
</script>
```

Після підключення бібліотеки ви зможете використовувати об’єкт `$` для вибору елементів сторінки, додавання обробників подій та реалізації логіки todo-додатку. На наступних етапах лабораторної роботи jQuery буде використовуватися для динамічного додавання задач, обробки кліків користувача та оновлення інтерфейсу без перезавантаження сторінки.

2. На цьому етапі необхідно підготувати HTML-розмітку, яка слугуватиме основою інтерфейсу todo-додатку. Саме в цьому розділі сторінки користувач буде переглядати список задач, додавати нові елементи та бачити загальну кількість відстежуваних пунктів.Додайте наведений нижче код до файлу `index.html`. Контейнер з класом `container` використовується для логічного групування всіх елементів додатку та забезпечує зручне розміщення контенту на сторінці. Заголовок `h1` відображає назву списку, а блок із написом **Tracked items** буде використовуватись для динамічного відображення кількості доданих задач.

```
<div class="container">
  <h1>Item list</h1>
    <div>Tracked items: <label id="tracked"></label></div>
  <div class="input-group">
    <input id="item-input" type="text" placeholder="Add new item">
    <button onclick="addItem()">Add Item</button>
```

```
  </div>
  <ul id="todo-list"></ul>
</div>
```

Поле введення з ідентифікатором `item-input` призначене для введення тексту нової задачі. Кнопка ***Add Item*** викликає JavaScript-функцію `addItem()`, яка буде реалізована на наступних етапах лабораторної роботи та відповідатиме за додавання нового пункту до списку.

Елемент `<ul>` з ідентифікатором `todo-list` є контейнером для списку задач. Саме сюди за допомогою jQuery будуть динамічно додаватися елементи `<li>` під час роботи додатку. Така структура дозволяє чітко розділити статичну HTML-розмітку та динамічну логіку керування задачами.

3. На цьому етапі лабораторної роботи необхідно реалізувати основну логіку todo-додатку за допомогою JavaScript та jQuery. Саме цей код відповідатиме за додавання нових задач до списку, їх динамічне відображення на сторінці та підготовку функціоналу для подальшого видалення і підрахунку елементів. У файлі `script.js` додайте наступний код:

```
let count = 0;
list = $("#todo-list");

function addItem(){
  let inputValue = $("#item-input").val().trim();

  let newToDoElement = $("<div></div>");
  newToDoElement.attr("id", "todo-"+count);

  let delButton = $("<button></button>");
  delButton.attr("id", count);
  delButton.text("delete me");
  delButton.click(deleteToDo.bind(this, count));
```

```
  newToDoElement.text(inputValue);
  newToDoElement.append(delButton);

  list.append(newToDoElement);
  $("#item-input").val("");
  count++;
}

function deleteToDo(){

}

function updateCounts() {

}
```

Змінна `count` використовується як лічильник задач і дозволяє надавати кожному елементу унікальний ідентифікатор. Змінна `list` містить посилання на HTML-елемент `<ul>` з ідентифікатором `todo-list`, у який будуть додаватися всі нові записи.

Рядок
`let inputValue = $("#item-input").val().trim();`
отримує текст, введений користувачем у поле введення, та видаляє зайві пробіли на початку і в кінці рядка. jQuery-селектори дозволяють легко звертатися до елементів сторінки за ідентифікатором, класом або тегом, аналогічно до CSS-селекторів.

Конструкція
`let newToDoElement = $("<div></div>");`
створює новий HTML-елемент `div`, який буде представляти окрему задачу. За допомогою методу `attr()` цьому елементу встановлюється унікальний атрибут `id`, сформований на основі значення лічильника.

Далі створюється кнопка видалення:

`let delButton = $("<button></button>");`.
Для неї встановлюється текст `delete me`, унікальний ідентифікатор та обробник події кліку. Метод
`delButton.click(deleteToDo.bind(this, count))`
прив’язує кнопку до функції `deleteToDo` і передає в неї ідентифікатор відповідного todo-елемента.

У рядку
`newToDoElement.text(inputValue);`
задається текст задачі, введений користувачем, а за допомогою
`newToDoElement.append(delButton);`
до елемента додається кнопка видалення.

Команда
`list.append(newToDoElement);`
додає новостворений todo-елемент до списку на сторінці. Після цього поле введення очищується за допомогою
`$("#item-input").val("");`,
а лічильник `count` збільшується на одиницю.

Функції `deleteToDo()` та `updateCounts()` залишені порожніми навмисно — їх реалізація буде виконана на наступних етапах лабораторної роботи. Вони відповідатимуть за видалення задач зі списку та оновлення кількості відстежуваних елементів.

4. Метод для видалення записів із списку.

На цьому етапі лабораторної роботи необхідно реалізувати функціонал видалення задач зі списку. Це дозволить користувачу керувати своїм todo-списком, видаляючи виконані або непотрібні записи без перезавантаження сторінки.

У файлі `script.js` запишіть наступний код у метод `deleteToDo()`:

```
function deleteToDo(j){
  let toDo = $("#todo-"+j);
  toDo.remove();
  updateCounts();
}
```

Функція `deleteToDo(j)` приймає параметр `j`, який відповідає унікальному ідентифікатору todo-елемента. За допомогою jQuery-селектора
`$("#todo-" + j)`
ми знаходимо конкретний елемент у DOM-дереві сторінки, який потрібно видалити.

Метод `.remove()` повністю видаляє вибраний елемент разом з усіма його вкладеними елементами з DOM. Після виконання цього методу відповідний запис більше не відображається на сторінці і не бере участі у подальшій роботі додатку.

Після видалення елемента викликається функція `updateCounts()`, яка відповідає за оновлення інформації про кількість відстежуваних задач. Такий підхід забезпечує актуальність даних у користувацькому інтерфейсі та дозволяє підтримувати логіку додатку у консистентному стані.

5. Відслідковування записів в списку.

На цьому етапі лабораторної роботи необхідно додати можливість відстежувати виконані або вибрані задачі у todo-списку. Для цього кожен запис буде доповнений елементом `checkbox`, який дозволить користувачу позначати задачі та бачити кількість відмічених пунктів у реальному часі.

Перш за все, модифікуйте метод `addItem()` так, щоб разом із текстом задачі та кнопкою видалення додавався чекбокс для трекінгу:

```
function addItem(){
  let inputValue = $("#item-input").val().trim();

  let newToDoElement = $("<div></div>");
  newToDoElement.attr("id", "todo-"+count);

  let delButton = $("<button></button>");
  delButton.attr("id", count);
  delButton.text("delete me");
  delButton.click(deleteToDo.bind(this, count));

  let checkbox = $("<input>").attr("type", "checkbox");
  checkbox.attr("class", "track-checkbox");
  checkbox.change(updateCounts);

  newToDoElement.text(inputValue);
  newToDoElement.append(delButton);

  newToDoElement.append(checkbox);

  list.append(newToDoElement);
  $("#item-input").val("");
  count++;
}
```

У цьому коді створюється новий елемент типу `input` з атрибутом `type="checkbox"`. Йому задається клас `track-checkbox`, який використовується для подальшого пошуку всіх чекбоксів на сторінці. Подія `change` прив’язується до функції `updateCounts()`, яка буде викликатися кожного разу, коли користувач ставить або знімає прапорець.

Далі необхідно реалізувати метод `updateCounts()`, який відповідає за підрахунок відмічених задач та відображення цієї інформації на сторінці:

```
function updateCounts() {
  let trackedCountElement = $("#tracked")
  count = $('.track-checkbox').length;
  trackedCount = 0;

  Array.from($('.track-checkbox')).forEach(checkbox => {
      if (checkbox.checked) {
          trackedCount++;
      }
  });
  trackedCountElement.html(trackedCount);
}
```

У цій функції всі чекбокси вибираються за допомогою селектора `$(".track-checkbox")`. Оскільки результатом є jQuery-колекція, вона перетворюється на звичайний масив, після чого для кожного елемента виконується цикл `forEach()`. Якщо прапорець встановлений (`checkbox.checked === true`), лічильник `trackedCount` збільшується на одиницю.

Після завершення підрахунку значення лічильника відображається у HTML-елементі з ідентифікатором `tracked`. Таким чином, користувач завжди бачить актуальну кількість відмічених задач, а додаток стає більш інтерактивним і зручним для повсякденного використання.

6. Додаткові можливості jQuery.

Існує багато інших методів jQuery, які можуть допомогти вам у веб-розробці. Ось лише деякі з найбільш широко використовуваних:

- `.css()`: отримує або встановлює властивості стилю CSS елемента.

Приклад:

```
$('#box').css('color', 'blue');
```

- «.addClass()» /«.removeClass()» / «.toggleClass()»: додає, видаляє або перемикає CSS-класи на елементі.

Приклад:

```
$('#box').addClass('active');
$('#box').removeClass('active');
$('#box').toggleClass('highlighted');
```

- «.removeAttr()»: Видаляє атрибут елемента

Приклад:

```
$('img').removeAttr('alt');
```

- «.append()» / «.prepend()»: Вставляє вміст всередину елемента. «.append()» додає вміст в кінці, а «.prepend()» — на початку.

Приклад:

```
$('#list').append('<li>New item</li>');
$('#list').prepend('<li>First item</li>');
```

- «.after()» / «.before()»: Вставляє вміст поза елементом. «.after()» додає вміст після елемента, «.before()» додає його перед елементом.

Приклад:

```
$('#box').after('<p>After the box</p>');
$('#box').before('<p>Before the box</p>');
```

- «.remove()» / «.empty()»: «.remove()» видаляє елемент, а «.empty()» очищає його вміст.

Приклад:

```
$('#box').remove();  // Removes the element
$('#container').empty();  // Clears inner content
```

- ".hide()" / ".show()": Приховує та показує елементи.

Приклад:

```
$('#box').hide();
$('#box').show();
```

- «.fadeIn()» / «.fadeOut()»: Анімує непрозорість елемента, щоб створити ефект з'являння або зникання.

Приклад:

```
$('#box').fadeIn();
$('#box').fadeOut();
```

- «.slideUp()» / «.slideDown()»: Пересуває елемент вгору або вниз, щоб приховати або показати його.

Приклад:

```
$('#box').slideUp();
$('#box').slideDown();
```

- «.on()»: Приєднує обробники подій до елементів для різних подій.

Приклад:

```
$('#button').on('mouseenter', function() {
```

```
    alert('Mouse entered!');
});
```

- «.each()»: Ітерація по колекції jQuery.

Приклад:

```
$('li').each(function(index) {
    console.log('Item ' + index + ': ' + $(this).text());
});
```

- «.animate()»: Створює власні анімації, змінюючи властивості CSS з часом.

Приклад:

```
$('#box').animate({ width: '200px', opacity: 0.5 });
```

- «.ajax()»: відправляє асинхронний HTTP-запит (наприклад, на сервер бекенду або до файлу, розташованого у вашому проектному просторі).

Приклад:

```
$.ajax({
    url: 'data.json',
    method: 'GET',
    success: function(data) {
        console.log(data);
    }
});
```

- «.parent()» / «.children()» / «.find()»: Перебирає елементи DOM, щоб знайти пов'язані елементи. «.parent()» вибирає батьківський елемент, «.children()» вибирає дочірні елементи, а «.find()» шукає нащадків.

Приклад:

```
$('#child').parent();        // Selects parent of #child
$('#parent').children();     // Selects children of #parent
$('#parent').find('.item');  // Selects .item within #parent
```

- «.ready()»: Виконує код, коли DOM повністю завантажений.

Приклад:

```
$(document).ready(function() {
    alert('DOM is ready!');
});
```

Виберіть деякі з них, щоб додати додаткові функції до вашого проекту.

Лабораторна робота №3: CRUD-додаток на Angular (Standalone + Signals + Reactive Forms + HTTP + NgRx)

Angular — це сучасний клієнтський фреймворк з відкритим вихідним кодом, який використовується для створення односторінкових веб-застосунків (Single Page Applications, SPA). Він розробляється та підтримується компанією Google і є одним із найпопулярніших інструментів у професійній веб-розробці. Angular дозволяє створювати масштабовані, структуровані та легко підтримувані застосунки, використовуючи чітко визначену архітектуру та сучасні можливості мови TypeScript.

На відміну від класичних підходів до веб-розробки, де більшість логіки виконувалась на сервері, Angular переносить значну частину обробки на клієнтську сторону. У результаті браузер завантажує застосунок один раз, а подальша взаємодія з користувачем відбувається без повного перезавантаження сторінки. Це забезпечує швидшу реакцію інтерфейсу та кращий користувацький досвід.

Angular побудований на компонентному підході. Уся сторінка або навіть великий застосунок складається з окремих компонентів, кожен з яких відповідає за свою частину інтерфейсу та логіки. Компонент зазвичай містить: 1) HTML-шаблон (вигляд), 2) CSS-стилі (оформлення) та 3) TypeScript-клас (логіка).

Такий підхід дозволяє легко перевикористовувати компоненти, тестувати їх окремо та підтримувати великий кодовий базис.

Ще одним ключовим принципом Angular є двостороннє зв’язування даних (two-way data binding). Це означає, що зміни в даних автоматично відображаються в інтерфейсі, а зміни в інтерфейсі — у даних. Хоча в сучасному Angular частіше використовується однонаправлений потік даних, механізм зв’язування залишається важливою частиною фреймворку.

Angular також активно використовує ін’єкцію залежностей (Dependency Injection). Цей механізм дозволяє передавати сервіси та інші залежності в

компоненти автоматично, без їхнього явного створення. Це робить код більш гнучким, тестованим і менш залежним від конкретних реалізацій.

Dependency Injection (DI) — це шаблон проєктування, який використовується для керування залежностями між частинами програми. Простими словами, DI означає, що об'єкт не створює свої залежності сам, а отримує їх ззовні. Це робить код більш зрозумілим, гнучким і легким для тестування.

Angular побудований на мові TypeScript — надбудові над JavaScript, яка додає статичну типізацію, інтерфейси, декоратори та інші можливості об'єктно-орієнтованого програмування. Використання TypeScript дозволяє виявляти помилки ще на етапі компіляції, покращує читабельність коду та значно спрощує роботу у великих командах.

Типовий Angular-проєкт має чітку та передбачувану структуру. Після створення нового проєкту за допомогою Angular CLI автоматично генерується набір папок і файлів, зокрема:

- `src/app` — основна папка застосунку;
- `main.ts` — точка входу, де відбувається запуск застосунку;
- `app.component.ts` — кореневий компонент;
- файли конфігурації для збірки та тестування.

Angular CLI (Command Line Interface) — це офіційний інструмент для створення, запуску та збірки Angular-застосунків. Він дозволяє швидко генерувати компоненти, сервіси, модулі, запускати локальний сервер розробки та створювати оптимізовану production-збірку.

HTML-шаблони в Angular розширені спеціальним синтаксисом. У них можна використовувати вирази, умовні конструкції та цикли. Для цього Angular надає директиви — спеціальні атрибути або конструкції, які змінюють поведінку DOM-елементів.

Найпоширеніші директиви:

1. структурні (`*ngIf`, `*ngFor`, `@for`) — змінюють структуру DOM;
2. атрибутивні (`ngClass`, `ngStyle`) — змінюють вигляд елементів;
3. власні директиви, створені розробником.

Завдяки директивам шаблони стають динамічними та тісно пов’язаними з логікою компонентів.

Для роботи з даними та бізнес-логікою в Angular використовуються сервіси. Сервіс — це клас, який зазвичай не має власного інтерфейсу користувача і призначений для виконання спільних задач, наприклад:

1. роботи з HTTP-запитами,
2. зберігання стану,
3. обробки даних.

Сервіси легко підключаються до компонентів через механізм ін’єкції залежностей. Такий підхід дозволяє відокремити логіку від інтерфейсу та дотримуватись принципів чистої архітектури.

Angular надає потужний HTTP-клієнт для взаємодії з бекенд-API. У поєднанні з RxJS та Observables це дозволяє ефективно працювати з асинхронними даними.

Маршрутизація є однією з ключових можливостей Angular. Вона дозволяє створювати багатосторінкову логіку всередині односторінкового застосунку. Кожен маршрут відповідає певному компоненту, який відображається залежно від URL-адреси.

Це дозволяє будувати складні застосунки з окремими сторінками для списків, деталей, форм авторизації тощо, не перезавантажуючи сторінку. Маршрути можуть містити параметри, захищатися guards та завантажуватися ліниво для покращення продуктивності.

Angular активно розвивається і включає сучасні механізми оптимізації. Серед них:

1. `ChangeDetectionStrategy.OnPush` для зменшення кількості перевірок змін;
2. Signals для керування локальним станом;
3. Deferrable Views (`@defer`) для відкладеного завантаження важких компонентів;
4. підтримка server-side rendering (SSR).

Ці інструменти дозволяють створювати швидкі та ефективні застосунки, придатні для реального використання у виробничому середовищі.

Angular є чудовим вибором для навчання, оскільки він поєднує в собі багато важливих концепцій сучасної розробки: компонентну архітектуру, типізацію, реактивне програмування, маршрутизацію та тестування. Знання Angular допомагає студентам краще зрозуміти, як будуються великі веб-системи, та підготуватися до роботи з реальними комерційними проєктами.

У практичному сенсі Angular широко використовується для створення корпоративних систем, адміністративних панелей, CRM-систем, внутрішніх веб-застосунків і складних інтерфейсів. Вивчення цього фреймворку відкриває студентам шлях до професійної фронтенд-розробки та глибшого розуміння екосистеми JavaScript.

Мета: опанувати створення SPA на Angular зі сучасними підходами (standalone компоненти, signals), формами, маршрутизацією, взаємодією з API та керуванням станом.

Очікувані результати навчання

- Створювати та запускати проєкт Angular через CLI.
- Реалізовувати компоненти/маршрути/шаблони з прив’язкою даних.
- Будувати Reactive Forms з валідацією.
- Працювати з HttpClient (CRUD) та інтерцепторами.
- Використовувати Signals для локального стану компонентів.
- Інтегрувати NgRx для глобального стану (опціонально — розширено).
- Оптимізувати продуктивність (OnPush, track, @defer).
- Покривати ключову логіку модульними тестами.

Передумови та середовище

- Node.js LTS, NPM, Angular CLI встановлені.
- Браузер Chrome/Firefox, редактор VS Code (рекомендовано).
- Фейковий бекенд: json-server або інший REST-мок.

Початкові команди

1. Інсталюйте необхідні утиліти

Перед початком роботи з Angular необхідно підготувати середовище розробки. Для цього потрібно встановити кілька базових інструментів. Angular працює на платформі Node.js, тому перш за все необхідно встановити LTS-версію Node.js (Long Term Support).

Node.js (LTS): із https://nodejs.org/en. Завантажте та встановіть LTS-версію для вашої операційної системи (Windows / macOS / Linux). Під час інсталяції використовуйте стандартні налаштування.
Після встановлення відкрийте консоль (Terminal / Command Prompt / PowerShell) і перевірте, що Node.js та npm встановлені коректно:

```
node -v
npm -v
```

У консолі мають відобразитися номери версій. Якщо команди не розпізнаються, необхідно перевірити, чи додано Node.js до системної змінної PATH.

Встановіть VS Code (http://code.visualstudio.com/).

2. Створіть новий Angular проект через команду в консолі

Angular-проєкти створюються за допомогою Angular CLI — офіційного інструмента командного рядка. Відкрийте консоль та перейдіть у папку, де ви хочете зберігати лабораторні роботи. Далі виконайте команду:

```
npx @angular/cli@latest new hello-angular --routing --style=scss
cd hello-angular
```

В цій команді:
npx @angular/cli@latest — запуск останньої версії Angular CLI без глобальної установки
new hello-angular — створення нового проєкту з назвою hello-angular
--routing — автоматичне підключення системи маршрутизації
--style=scss — використання SCSS замість звичайного CSS

Під час виконання команди Angular CLI може задати кілька додаткових питань — використовуйте значення за замовчуванням.

3. Запустіть сервер використовуючи наступну команду

Для запуску застосунку виконайте команду:

ng serve -o

Ця команда: компілює застосунок; запускає локальний сервер розробки та автоматично відкриває застосунок у браузері.

За замовчуванням сайт буде доступний за адресою: http://localhost:4200
Якщо у браузері з'явилася стартова сторінка Angular — проєкт створено та запущено успішно.

4. Огляд згенерованого проекту

Після створення проєкту важливо розуміти, за що відповідають основні файли.

`src/main.ts` - Головна точка входу в застосунок. Саме з цього файлу Angular запускає (bootstrap) весь застосунок.
`src/app/app.component.*` - Кореневий компонент застосунку.
Він відповідає за початковий інтерфейс та є контейнером для інших компонентів.
`src/app/app.routes.ts` - Файл маршрутизації, у якому описується список сторінок (route -> component).
`src/app/app.config.ts` - Файл глобальної конфігурації застосунку.
Тут підключаються загальні сервіси (router, HTTP-клієнт, інші провайдери).

Завдання (етапи)

1. Створіть standalone AppComponent і базове меню (Home, Products).

Standalone-компоненти не потребують `NgModule`. Ми можемо оголосити головний компонент наступним чином:

```
// app.component.ts
```

```
import { Component } from '@angular/core';
import { RouterModule } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterModule],
  template: `
    <nav>
      <a routerLink="/">Home</a> |
      <a routerLink="/products">Products</a>
    </nav>
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {}
```

2. Додайте маршрути: '' -> Home, 'products' -> ProductsList, 'products/:id' -> ProductDetails.

Налаштуйте маршрути для головної сторінки, списку товарів і деталей конкретного товару.

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter, Routes } from '@angular/router';
import { AppComponent } from './app/app.component';
import { HomeComponent } from './app/home.component';
import { ProductsListComponent } from './app/products-list.component';
```

```
import { ProductDetailsComponent } from './app/product-details.component';


const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'products', component: ProductsListComponent },
  { path: 'products/:id', component: ProductDetailsComponent },
];


bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)],
});
```

3. Реалізуйте ProductsService з методами list/get/create/update/delete.

Сервіс для роботи з товарами — CRUD (create, read, update, delete). Можна використовувати доступний api, власний бекенд, або мокати запити.

```
// products.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';


export interface Product {
  id?: number;
  name: string;
  category: string;
  price: number;
}


@Injectable({ providedIn: 'root' })
```

```
export class ProductsService {

  private apiUrl = 'https://fakestoreapi.com/products'; // Change to your
own api/ local backend/ in-memory array


  constructor(private http: HttpClient) {}


  list(): Observable<Product[]> {

    return this.http.get<Product[]>(this.apiUrl);

  }


  get(id: number): Observable<Product> {

    return this.http.get<Product>(`${this.apiUrl}/${id}`);

  }


  create(p: Product) {

    return this.http.post<Product>(this.apiUrl, p);

  }


  update(id: number, p: Product) {

    return this.http.put<Product>(`${this.apiUrl}/${id}`, p);

  }


  delete(id: number) {

    return this.http.delete(`${this.apiUrl}/${id}`);

  }

}
```

4. Зробіть список товарів з пагінацією/фільтрами (за категорією) та track для списку.

Використайте просту фільтрацію за категорією і `trackBy` для ефективного рендеру.

```
// products-list.component.ts

import { Component, OnInit, signal, computed } from '@angular/core';

import { NgFor, NgIf } from '@angular/common';

import { ProductsService, Product } from './products.service';


@Component({

  standalone: true,

  selector: 'app-products-list',

  imports: [NgFor, NgIf],

  template: `

    <h2>Products</h2>

    <select [(ngModel)]="selectedCategory">

      <option value="">All</option>

      <option *ngFor="let c of categories">{{ c }}</option>

    </select>


    <ul>

      <li *ngFor="let p of filteredProducts(); trackBy: trackById">

        <a [routerLink]="['/products', p.id]">{{ p.name }}</a> - {{
p.price }}$

      </li>

    </ul>

  `,

})

export class ProductsListComponent implements OnInit {

  products = signal<Product[]>([]);

  selectedCategory = '';
```

```
  categories = ['electronics', 'clothes', 'books'];

  constructor(private service: ProductsService) {}

  ngOnInit() {
    this.service.list().subscribe(data => this.products.set(data));
  }

  filteredProducts = computed(() =>
    this.selectedCategory
      ? this.products().filter(p => p.category === this.selectedCategory)
      : this.products()
  );

  trackById(_: number, item: Product) {
    return item.id;
  }
}
```

5. Створіть форму додавання/редагування (Reactive Forms) з валідацією.

Приклад:

```
// product-form.component.ts
import { Component } from '@angular/core';
import { FormBuilder, Validators, ReactiveFormsModule } from
'@angular/forms';

@Component({
```

```
  standalone: true,
  selector: 'app-product-form',
  imports: [ReactiveFormsModule],
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <input formControlName="name" placeholder="Name" />
      <input formControlName="price" type="number" placeholder="Price" />
      <input formControlName="category" placeholder="Category" />
      <button type="submit" [disabled]="form.invalid">Save</button>
    </form>
  `,
})
export class ProductFormComponent {
  form = this.fb.group({
    name: ['', Validators.required],
    price: [0, [Validators.required, Validators.min(1)]],
    category: ['', Validators.required],
  });

  constructor(private fb: FormBuilder) {}

  onSubmit() {
    console.log(this.form.value);
  }
}
```

6. Додайте інтерцептор для логування запитів та обробки помилок.

Приклад інтерцептора:

```
// product-form.component.ts

import { Component } from '@angular/core';

import { FormBuilder, Validators, ReactiveFormsModule } from
'@angular/forms';


@Component({

  standalone: true,

  selector: 'app-product-form',

  imports: [ReactiveFormsModule],

  template: `

    <form [formGroup]="form" (ngSubmit)="onSubmit()">

      <input formControlName="name" placeholder="Name" />

      <input formControlName="price" type="number" placeholder="Price" />

      <input formControlName="category" placeholder="Category" />

      <button type="submit" [disabled]="form.invalid">Save</button>

    </form>

  `,

})

export class ProductFormComponent {

  form = this.fb.group({

    name: ['', Validators.required],

    price: [0, [Validators.required, Validators.min(1)]],

    category: ['', Validators.required],

  });


  constructor(private fb: FormBuilder) {}
```

```
  onSubmit() {
    console.log(this.form.value);
  }
}
```

Реєстрація в main.ts:

```
import { provideHttpClient, withInterceptors } from
'@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient(withInterceptors([loggingInterceptor])),
  ],
});
```

7. Використайте Signals у ProductDetails для локального стану (loading, error, product).

```
// product-details.component.ts
import { Component, OnInit, signal } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { ProductsService } from './products.service';

@Component({
  standalone: true,
  selector: 'app-product-details',
  template: `
    <div *ngIf="loading()">Loading...</div>
    <div *ngIf="error()">{{ error() }}</div>
```

```
    <div *ngIf="product()">
      <h3>{{ product()?.name }}</h3>
      <p>{{ product()?.price }}$</p>
    </div>
  `,
})
export class ProductDetailsComponent implements OnInit {
  product = signal<any>(null);
  loading = signal(false);
  error = signal('');

  constructor(private route: ActivatedRoute, private service:
ProductsService) {}

  ngOnInit() {
    const id = Number(this.route.snapshot.paramMap.get('id'));
    this.loading.set(true);
    this.service.get(id).subscribe({
      next: data => {
        this.product.set(data);
        this.loading.set(false);
      },
      error: err => {
        this.error.set('Failed to load');
        this.loading.set(false);
      },
    });
  }
}
```

8. Опціонально A: інтегруйте NgRx (store slice 'products', actions/effects/selectors).

NgRx: дії (фрагмент)

```
// products.actions.ts (NgRx, опціонально)
export const load = createAction('[Products] Load');
export const loadSuccess = createAction('[Products] Load Success', props<{
products: Product[] }>());
```

9. Опціонально B: додайте Deferrable Views (@defer) для «важких» віджетів.

Lazy-завантаження важкого віджета:

```
<!-- products-list.component.html -->

@defer (on viewport) {

  <app-product-stats></app-product-stats>

} @placeholder {

  <p>Loading stats...</p>

}
```

10. Додайте unit-тести для сервісу та одного компонента (TestBed).
Приклад тесту для сервісу:

```
// products.service.spec.ts

import { TestBed } from '@angular/core/testing';

import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';

import { ProductsService } from './products.service';



describe('ProductsService', () => {
```

```
  let service: ProductsService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [ProductsService],
    });
    service = TestBed.inject(ProductsService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  it('should list products', () => {
    service.list().subscribe(data => expect(data.length).toBe(2));
    const req = httpMock.expectOne('https://fakestoreapi.com/products');
    req.flush([{ id: 1, name: 'A' }, { id: 2, name: 'B' }]);
  });
});
```

Доменна область і дані

Приклад «Каталог товарів»: Product { id:number; title:string; price:number; category:string; stock:number }.

db.json (фрагмент)

```
{
  "products": [
    { "id": 1, "title": "Keyboard", "price": 1200, "category":
"Peripherals", "stock": 5 },
    { "id": 2, "title": "Mouse", "price": 800, "category": "Peripherals",
"stock": 12 }
  ]
}
```

Додаткові поради:

Використайте `ChangeDetectionStrategy.OnPush`

Для компонентів **списку товарів** та **деталей товару** встановіть стратегію `OnPush`.
Це зменшує кількість непотрібних перевірок змін у шаблоні.

```
import { Component, ChangeDetectionStrategy } from '@angular/core';


@Component({
  selector: 'app-products-list',
  templateUrl: './products-list.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ProductsListComponent {
  // ...
}
```

Використайте `@for` або `trackBy` у списках

Для ефективного оновлення DOM при рендері використайте `trackBy` або новий синтаксис `@for`:

```
<!-- Варіант із *ngFor -->
<li *ngFor="let p of products; trackBy: trackById">{{ p.name }}</li>


<!-- Або варіант із @for (Angular 17+) -->
@for (p of products; track p.id) {
  <li>{{ p.name }}</li>
}
```

Використайте `@defer` для великих віджетів

Відкладіть завантаження важких компонентів (наприклад, графіків або карт) до моменту, коли вони з'являться у в'юпорті.

```
@defer (on viewport) {

  <app-sales-chart></app-sales-chart>

} @placeholder {

  <p>Loading chart...</p>

}
```

Увімкніть event coalescing при старті додатку

Ця опція об'єднує події від користувача (наприклад, кілька кліків) у одну перевірку змін, що зменшує навантаження на рендеринг.

```
import { bootstrapApplication } from '@angular/platform-browser';

import { provideZoneChangeDetection } from '@angular/core';

import { AppComponent } from './app/app.component';


bootstrapApplication(AppComponent, {

  providers: [provideZoneChangeDetection({ eventCoalescing: true })],

});
```

Напишіть модульні тести для `ProductsService` з використанням HTTP-моків

Створіть тест, який перевіряє, що запити надсилаються на правильну адресу та повертають очікувані дані.

Звіт і критерії оцінювання

Підготуйте короткий звіт (2–5 стор.) з описом архітектури, скриншотами та висновками.

Критерій	Бали
Функціональність CRUD + форми + валідація	6
HTTP + інтерцептор + обробка помилок	3
Маршрути/навігація + деталі	2
Signals/локальний стан або NgRx (частково)	3
Продуктивність (@for/OnPush/@defer)	2
Тести (мінімум 3 осмислені)	2
Якість коду/типізація/ESLint	2

Порядок здачі

11. Код у репозиторії (GitHub/GitLab) з інструкцією запуску.

12. Лінка на гітхаб репозиторій в moodle.

13. Демонстрація (5–7 хв.): функціонал, код, питання-відповіді.

Лабораторна робота 4. React

React.js — це потужна бібліотека JavaScript для створення користувацьких інтерфейсів. Вона дозволяє розробникам ефективно створювати динамічні та адаптивні веб-додатки. Однією з її ключових особливостей є декларативна архітектура на основі компонентів, яка робить розробку користувацьких інтерфейсів більш структурованою та керованою.

React був розроблений Джорданом Волке, інженером-програмістом у Facebook, і вперше випущений у 2013 році. Він був створений для поліпшення складної розробки інтерфейсу користувача Facebook шляхом впровадження більш ефективного способу управління оновленнями та рендерингом. Згодом React набув широкого поширення завдяки своїм перевагам у продуктивності та можливості повторного використання компонентів.

Розробка React оптимізована за допомогою функцій автоматичного перезавантаження та гарячого перезавантаження. Коли в код вносяться зміни, вони миттєво відображаються в браузері без необхідності повного оновлення сторінки. Це значно прискорює процес розробки та покращує досвід розробника.

React полегшує розробку односторінкових додатків (SPA). На відміну від традиційних багатосторінкових додатків, SPA завантажує один HTML-файл і динамічно оновлює вміст за потреби без необхідності перезавантаження сторінки. Це забезпечує швидшу навігацію та більш плавний користувацький досвід.

Важливо розуміти різницю між React і React Native. React використовується для створення веб-додатків, а React Native — це фреймворк, який використовується переважно для мобільних додатків. React працює в браузерах, а React Native — на компонентах Native UI в iOS і Android.

Існувало кілька основних версій React:

- React 16 (2017): введено механізм узгодження Fiber для кращої продуктивності.

- React 17 (2020): поліпшено поступові оновлення без руйнівних змін.

- React 18 (2022): додано такі функції, як Concurrent Rendering і Suspense для кращої продуктивності.

- Остання версія: React продовжує розвиватися, отримуючи нові функції та оптимізації.

Виконання лабораторної роботи

На цьому етапі ми створимо найпростіший React-додаток, використовуючи офіційний інструмент Create React App (CRA). Він дозволяє швидко налаштувати середовище розробки без необхідності вручну конфігурувати Webpack, Babel та інші інструменти.

Відкрийте термінал (Command Prompt, PowerShell або Terminal) та виконайте наступну команду:

npx create-react-app new-app

Тут npx це інструмент, який дозволяє запускати npm-пакети без попередньої глобальної інсталяції. create-react-app - офіційний генератор React-проєктів. new-app це назва майбутнього проєкту (ви можете змінити її за потреби).

Після виконання цієї команди, буде створено нову папку new-app. Також автоматично згенерується базова структура React-проєкту, та будуть встановлені всі необхідні залежності. Буде налаштовано середовище для розробки та збірки застосунку.

Процес може тривати кілька хвилин, залежно від швидкості інтернет-з’єднання та продуктивності комп’ютера.

Після завершення генерації перейдіть до створеної директорії:

cd new-app

У цій папці знаходяться всі файли React-додатку, включно з конфігурацією, вихідним кодом та залежностями.

Після цього, для запуску локального сервера розробки виконайте команду:

npm start

Ця команда компілює застосунок, запускає development server та автоматично відкриває застосунок у браузері. За замовчуванням React-додаток буде доступний за адресою http://localhost:3000. Якщо все виконано правильно, у браузері з'явиться стандартна стартова сторінка React з повідомленням "Edit src/App.js and save to reload".

Щоб зупинити сервер, натисніть Ctrl + C у терміналі. Папка node_modules створюється автоматично та містить всі зовнішні бібліотеки. Її не потрібно редагувати вручну.

Розробка React оптимізована за допомогою функцій автоматичного перезавантаження (automatic reload) та гарячого перезавантаження (hot reload). Коли в код вносяться зміни, вони миттєво відображаються в браузері без необхідності повного оновлення сторінки. Це значно прискорює процес розробки та покращує досвід розробника.

Давайте створимо односторінковий додаток, який повертає один HTML-файл. Погляньте на згенерований репозиторій.

Файл index.js служить точкою входу в додаток React. Він відповідає за імпорт необхідних модулів і рендеринг кореневого компонента в DOM. Функція ReactDOM.render() (або її сучасний еквівалент createRoot) виконується для рендерингу головного компонента App всередині певного елемента DOM, зазвичай елемента з ідентифікатором root в index.html.

Файл App.js є центральним елементом будь-якого React-додатку, оскільки саме в ньому зазвичай описується кореневий компонент інтерфейсу користувача. У React інтерфейс будується з компонентів, і кожен

компонент є звичайною функцією JavaScript, яка повертає опис того, що має бути відображено на екрані. Цей опис записується у форматі JSX (JavaScript XML) — спеціальному синтаксисі, що поєднує можливості JavaScript та HTML. Завдяки JSX розмітка інтерфейсу виглядає знайомо для веб-розробників, але при цьому залишається частиною JavaScript-коду, що дозволяє легко працювати зі змінними, умовами та логікою прямо всередині розмітки.

Напишемо компоненту App.js:

```
import './App.css';
import React from 'react';

function App() {
  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <ul className='task-list'>
        <li>Task 1</li>
        <li>Task 2</li>
        <li>Task 3</li>
      </ul>
    </div>
  );
}

export default App;
```

На початку файлу ми імпортуємо файл стилів App.css, який відповідає за візуальне оформлення компонента, а також саму бібліотеку React, необхідну для роботи JSX. Далі оголошується функція App, яка є функціональним компонентом. Усередині цієї функції використовується оператор return, що повертає JSX-структуру. В даному прикладі вона складається з контейнера div з CSS-класом container, який використовується для стилізації та організації макета. Усередині контейнера знаходиться заголовок другого рівня з текстом “Tasks list”, що задає назву списку завдань, а також невпорядкований список ul з класом task-list. Кожен елемент списку li представляє окреме завдання, і на цьому етапі вони є статичними, тобто жорстко заданими без використання стану чи динамічних даних.

Наприкінці файлу використовується конструкція export default App, яка робить компонент App доступним для використання в інших частинах додатку. Саме цей компонент імпортується у файлі index.js і відображається в DOM браузера. Таким чином, файл App.js виступає відправною точкою для подальшої розробки React-додатку, а наведений приклад демонструє базову ідею компонентного підходу, коли інтерфейс описується як сукупність невеликих, логічно ізольованих частин.

Додамо стилізацію в App.css:

```
.container{
  max-width: 1200px;
  margin: 1rem auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 0.5rem;
  background-color: #f9f9f9;
}

.task-list{
  list-style-type: none;
  padding: 0;
}
```

Зверніть увагу, що у React ви вказуєте клас CSS за допомогою className. Він працює так само, як атрибут HTML class.

Перейшовши на http://localhost:3000/, ми бачимо список завдань із App.js.

Компоненти

Компоненти — це незалежні та багаторазові фрагменти коду. Вони виконують ту саму функцію, що й функції JavaScript, але працюють ізольовано та повертають HTML.

У React уся побудова інтерфейсу базується на компонентах. Компонент можна розглядати як окремий, самодостатній блок інтерфейсу, який інкапсулює власну розмітку, стилі та, за потреби, логіку. Такий підхід

робить код зрозумілішим, легшим для підтримки та повторного використання. Замість того щоб описувати весь інтерфейс у одному файлі, ми розбиваємо його на логічні частини, кожна з яких відповідає за свій фрагмент сторінки.

У цьому прикладі ми виносимо логіку відображення списку завдань в окрему компоненту TodoList. Для цього створюємо файл todoList.js у папці components. Усередині файлу оголошується функція TodoList, яка, так само як і компонент App, повертає JSX-розмітку. Компонента містить заголовок і список завдань, оформлений за допомогою HTML-тегів ul та li. На даному етапі список є статичним і використовується лише для демонстрації структури компоненти, але згодом його легко можна перетворити на динамічний, підключивши стан або передаючи дані через властивості (props).

Важливою частиною є рядок export default TodoList, який дозволяє експортувати компоненту та використовувати її в інших файлах проєкту. Після цього ми імпортуємо створену компоненту у файл App.js за допомогою оператора import. Зверніть увагу, що шлях до файлу вказується відносно поточного файлу, а ім’я компоненти починається з великої літери — це загальноприйнята конвенція в React, яка дозволяє відрізняти компоненти від звичайних HTML-тегів.

```
function TodoList() {
    return (
        <div>
            <h2>Tasks list </h2>
            <ul className='task-list'>
                <li>Task 1</li>
                <li>Task 2</li>
                <li>Task 3</li>
            </ul>
        </div>
    );
}

export default TodoList;
```

У компоненті `App` ми більше не описуємо безпосередньо список завдань, а просто вставляємо `<TodoList />` у JSX-розмітку. Таким чином, `App` виступає як контейнер або батьківська компонента, а `TodoList` — як вкладена (дочірня) компонента. Це добре демонструє ідею композиції компонентів у React, коли складні інтерфейси створюються шляхом комбінування простіших елементів. Такий підхід значно полегшує розширення додатку, адже кожну компоненту можна змінювати або перевикористовувати незалежно від інших частин системи.

```
import logo from './logo.svg';
import './App.css';
import TodoList from './components/todoList';

function App() {
  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <TodoList />
    </div>
  );
}

export default App;
```

Props

export default App;

У React компоненти можуть обмінюватися даними між собою за допомогою props (скорочено від properties). Пропси — це механізм передачі даних від батьківської компоненти до дочірньої. За своєю суттю вони дуже схожі на аргументи функцій у звичайному JavaScript: батьківська компонента передає значення, а дочірня — приймає їх і використовує для побудови інтерфейсу. Важливо пам’ятати, що пропси є тільки для читання — дочірня компонента не повинна змінювати дані, які вона отримує від батька.

У цьому кроці ми переносимо список завдань у компоненту App.js, роблячи її джерелом даних для всього додатку. У змінній tasks зберігається масив рядків, кожен з яких представляє окреме завдання. Далі цей масив передається у компоненту TodoList через атрибут tasks, коли ми використовуємо компоненту у JSX у вигляді <TodoList tasks={tasks} />. Таким чином, App виступає як керуюча компонента, яка володіє даними, а TodoList — як презентаційна компонента, відповідальна лише за відображення цих даних.

У файлі todoList.js компонент TodoList приймає об’єкт пропсів як аргумент функції. У нашому випадку ми отримуємо доступ до масиву завдань через властивість tasks.tasks. Далі за допомогою методу map() ми перебираємо масив і для кожного елемента створюємо HTML-елемент <li>. Метод map() дуже часто використовується в React для рендерингу списків, оскільки дозволяє зручно перетворювати масиви даних у JSX-розмітку.

Зверніть увагу на атрибут key, який передається кожному елементу списку. React використовує key для ефективного оновлення DOM, тому він повинен бути унікальним для кожного елемента списку. У цьому прикладі ми використовуємо індекс масиву як ключ, що допустимо для простих навчальних прикладів, хоча в реальних проєктах краще використовувати унікальні ідентифікатори.

У результаті такого підходу ми отримуємо більш гнучку та масштабовану архітектуру додатку. Компонента TodoList більше не залежить від конкретного набору даних — вона може відображати будь-який список завдань, який буде переданий їй через пропси. Це робить компоненту багаторазовою та зручною для використання в різних частинах додатку або навіть у зовсім інших проєктах.

```
function TodoList(tasks) {
    const tasksList = tasks.tasks.map((task, index) => <li
key={index}>{task}</li>);

    return (
        <div>
```

```
            <ul className='task-list'>
                {tasksList}
            </ul>
        </div>
    );
}

export default TodoList;
```

В компоненті App.js передамо через пропси список завдань:

```
import logo from './logo.svg';
import './App.css';
import TodoList from './components/todoList';

function App() {
  let tasks = ['Task 1', 'Task 2', 'Task 3'];

  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <TodoList tasks={tasks}/>
    </div>
  );
}

export default App;
```

useState

Хук useState є одним із базових і найважливіших інструментів у React, оскільки саме він дозволяє функціональним компонентам працювати зі станом. Стан (state) — це дані, які можуть змінюватися з часом і впливають на те, що бачить користувач на екрані. На відміну від звичайних змінних JavaScript, значення стану зберігаються між повторними

рендерами компонента, що робить можливим створення динамічних і інтерактивних інтерфейсів.

Функція useState викликається всередині компонента і приймає початкове значення стану. У нашому випадку це масив рядків ['Task 1', 'Task 2', 'Task 3'], який представляє список завдань. Результатом виклику useState є масив із двох елементів, який ми одразу розпаковуємо за допомогою деструктуризації. Перший елемент — це поточне значення стану (tasks), а другий — функція для його оновлення (updateTasks). Саме цю функцію потрібно використовувати щоразу, коли ми хочемо змінити стан, оскільки пряме редагування змінної стану в React є неправильним і не призведе до повторного рендеру компонента.

У компоненті App змінна tasks тепер є частиною стану, а не простою локальною змінною. Це означає, що при її зміні React автоматично перерендерить компонент і всі його дочірні компоненти, які залежать від цього стану. Далі ми передаємо tasks у компоненту TodoList через пропси, як і раніше. Таким чином, App продовжує залишатися джерелом даних, але тепер ці дані можуть динамічно змінюватися в майбутньому, наприклад, при додаванні або видаленні завдань користувачем.

Використання useState є першим кроком до створення повноцінної інтерактивної логіки в React-додатках. На цьому етапі список завдань ще статичний, але завдяки застосуванню стану ми підготували основу для наступних кроків лабораторної роботи, де будемо додавати можливість змінювати список завдань у відповідь на дії користувача.

App.js:

```
import logo from './logo.svg';
import './App.css';
import TodoList from './components/todoList';
```

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

function App() {
  const [tasks, updateTasks] = useState(['Task 1', 'Task 2', 'Task
3']);

  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <TodoList tasks={tasks}/>
    </div>
  );
}

export default App;
```

Додавання нових сутностей

Додамо новий компонент, який буде відповідати за додавання нових тасок в список. В цей компонент, через пропси, передамо функцію, яка буде використовувати updateTasks метод для додавання нових даних в state компонента App.js:

UpdateList.js:

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

function UpdateList({ onAddTask }) {
    const [value, setValue] = useState('');

    const handleSubmit = (e) => {
        e.preventDefault();
        if (value.trim()) {
            onAddTask(value.trim());
            setValue('');
        }
    };
```

```
    return (
        <form onSubmit={handleSubmit}>
            <input
                type="text"
                value={value}
                onChange={(e) => setValue(e.target.value)}
                placeholder="New task"
            />
            <button type="submit">Add Task</button>
        </form>
    );
}

export default UpdateList;
```

Також ця компонента має свій власний state, який містить дані в полі таски.

Та в App.js:

```
import logo from './logo.svg';
import './App.css';
import TodoList from './components/todoList';
import UpdateList from './components/updateList';
import { useState } from 'react';

function App() {
  const [tasks, updateTasks] = useState(['Task 1', 'Task 2', 'Task
3']);

  const addTask = (newTask) => {
    updateTasks([...tasks, newTask]);
  }

  return (
    <div className='container'>
```

```
      <h2>Tasks list </h2>
      <UpdateList onAddTask={addTask} />
      <TodoList tasks={tasks}/>
    </div>
  );
}

export default App;
```

Видалення сутностей

Для видалення сутностей із списку, додамо нову функцію removeTask в App.js, що буде видаляти відповідний елемент із state, та передамо цю функцію через props в дочірню компоненту TodoList:

todoList.js:

```
function TodoList({ tasks = [], onDelete }) {
    const tasksList = tasks.map((task, index) => (
        <li key={index} style={{display: 'flex', alignItems:
'center', gap: 8}}>
            <span style={{flex: 1}}>{task}</span>
                <button
                    onClick={() => onDelete(index)}
                    aria-label={`Delete task ${index + 1}`}
                    style={{padding: '4px 8px'}}
                >
                    Delete
                </button>
        </li>
    ));

    return (
        <div>
            <ul className='task-list'>
                {tasksList}
            </ul>
        </div>
```

```
    );
}

export default TodoList;
```

App.js:

```
import './App.css';
import TodoList from './components/todoList';
import UpdateList from './components/updateList';
import { useState } from 'react';

function App() {
  const [tasks, updateTasks] = useState(['Task 1', 'Task 2', 'Task
3']);

  const addTask = (newTask) => {
    updateTasks((prev) => [...prev, newTask]);
  };

  const removeTask = (indexToRemove) => {
    updateTasks((prev) => prev.filter((_, i) => i !==
indexToRemove));
  };

  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <UpdateList onAddTask={addTask} />
      <TodoList tasks={tasks} onDelete={removeTask} />
    </div>
  );
}

export default App;
```

Завдання.

- Додайте можливість редагувати окремі записи. Для цього додайте нову функцію та передайте її через пропси в дочірню компоненту.
- Додайте можливість трекінгу сутностей

Додаткові завдання.

- Додайте запит на api/бекенд для отримання сутностей, що відображаються в додатку.

Приклад в App.js:

```
import './App.css';
import TodoList from './components/todoList';
import UpdateList from './components/updateList';
import { useState } from 'react';
import { useEffect } from 'react';
import axios from 'axios';

function App() {
  const [tasks, updateTasks] = useState([]);

  useEffect(() => {
    const fetchTasks = async () => {
      try {
        const res = await axios.get('https://your-api.com/tasks');
        const taskNames = res.data.map(item => item.name); // take
only one field
        updateTasks(taskNames);
      } catch (err) {
        console.error('Error fetching tasks:', err);
      }
    };

    fetchTasks();
  }, []);

  const addTask = (newTask) => {
```

```
    updateTasks((prev) => [...prev, newTask]);
  };

  const removeTask = (indexToRemove) => {
    updateTasks((prev) => prev.filter((_, i) => i !==
indexToRemove));
  };

  return (
    <div className='container'>
      <h2>Tasks list </h2>
      <UpdateList onAddTask={addTask} />
      <TodoList tasks={tasks} onDelete={removeTask} />
    </div>
  );
}

export default App;
```

Тут використовується useEffect - хук useEffect дозволяє виконувати побічні ефекти у ваших компонентах. Деякі приклади побічних ефектів: отримання даних, безпосереднє оновлення DOM та таймери. useEffect приймає два аргументи. Другий аргумент є необов'язковим.useEffect(<function>, <dependency>).

Можна використовувати відкритий api. За можливості, додати запити POST та PUT на додавання та редагування запитів по api.

Не забудьте встановити axios:
npm i axios

Додатки на вибір.

- Todo app: додаток для трекінгу завдань користувача, з можливістю додавати та редагувати таски.
- Інтернет-магазин: магазин для онлайн-закупівель, з можливістю додавати покупки в кошик.
- Особистий блог: блог користувача з можливістю додавати та редагувати пости.
- Додаток для нотатків: додаток для особистих нотатків користувача з можливістю додавати та редагувати нотатки.
- Фінансовий трекер: додаток для трекінгу особистих фінансів користувача з можливістю підключати онлайн банкінг.
- Форум: онлайн форум з можливістю для користувачів додавати та редагувати пости по тематиках.
- etc.

Деплой на Render.com

Додайте наступний Dockerfile в ваш репозиторій з react додатком

```
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install --silent
COPY . .
RUN npm run build
FROM nginx:stable-alpine AS production
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Перейдіть на render.com, зареєструйтеся, та виберіть new web service -> Docker. Вкажіть лінку на ваш github репозиторій. Через декілька хвилин ви отримаєте лінку на ваш задеплоєний реакт додаток.

Лабораторна робота 5. Node Backend

Node.js — це серверне середовище виконання JavaScript, побудоване на рушії V8 (Google Chrome). Воно дозволяє виконувати JavaScript-код поза браузером, що зробило можливим використання JavaScript для створення серверних застосунків, API та інструментів розробки.

Головною особливістю Node.js є асинхронна, подієво-орієнтована модель виконання з неблокуючим введенням-виведенням (non-blocking I/O). Завдяки цьому Node.js добре підходить для застосунків, які обробляють велику кількість одночасних запитів, наприклад веб-серверів, чатів або стрімінгових сервісів.

Node.js використовує однопотокову модель, але за рахунок механізму event loop може ефективно працювати з багатьма запитами паралельно. Це спрощує розробку та зменшує витрати системних ресурсів порівняно з класичними багатопотоковими серверами.

Важливою складовою екосистеми Node.js є npm (Node Package Manager) — менеджер пакетів, який надає доступ до величезної кількості готових бібліотек і фреймворків. За допомогою npm розробники можуть швидко додавати необхідний функціонал до своїх проєктів.

Node.js широко використовується для створення REST та GraphQL API, а також серверів реального часу (WebSocket, чати). Варто також відмітити широке застосування для побудови мікросервісної архітектури, інструментів збірки та автоматизації (Webpack, Vite, ESLint) та повноцінних full-stack застосунків разом із фреймворками Angular, React або Vue.

Таким чином, Node.js є фундаментальною технологією сучасної веб-розробки, яка об’єднує клієнтську та серверну частини застосунку навколо однієї мови програмування — JavaScript.

Docker — це платформа контейнеризації, яка використовується для деплою, розгортання та запуску веб-застосунків у стандартизованому середовищі. Вона дозволяє упакувати застосунок разом із усіма його

залежностями (бібліотеки, налаштування, середовище виконання) в ізольований контейнер.

Основна ідея Docker полягає в принципі «працює на моєму комп'ютері — працює всюди». Контейнер Docker запускається однаково на локальному комп'ютері, тестовому сервері або у хмарі, що значно спрощує процес деплою та зменшує кількість помилок, пов'язаних із відмінностями середовищ.

Docker-контейнери базуються на образах (images) — незмінних шаблонах, які описуються у файлі Dockerfile. У Dockerfile крок за кроком визначається, яке базове середовище використовувати, як встановити залежності та як запустити веб-застосунок. На основі образу створюються контейнери — запущені екземпляри застосунку.

У контексті веб-розробки Docker найчастіше використовується як для деплою frontend-застосунків (Angular, React, Vue) разом із веб-сервером (Nginx) та і для запуску backend-застосунків (Node.js, .NET, Python). Можна також налаштувати розгортання баз даних та кешів (PostgreSQL, MongoDB, Redis).

Для складніших сценаріїв застосовується Docker Compose, який дозволяє описати кілька взаємопов'язаних сервісів (frontend, backend, база даних) в одному конфігураційному файлі та запускати їх однією командою.

Docker значно спрощує процес CI/CD (безперервної інтеграції та доставки), оскільки кожен етап пайплайну працює з однаковими контейнерами. Це робить деплой швидшим, надійнішим і передбачуваним.

1. Згенеруйте Node.js-застосунок локально та встановіть усі необхідні для роботи пакети за допомогою наведених команд. Спочатку створюється окрема директорія проєкту та ініціалізується файл package.json, у якому зберігається інформація про застосунок, його залежності та скрипти запуску. Параметр -y дозволяє створити package.json з типовими налаштуваннями без додаткових запитань.

```
mkdir myapp && cd myapp
npm init -y
```

npm install express mongoose cors dotenv

Після цього за допомогою менеджера пакетів npm встановлюються основні бібліотеки для розробки серверного застосунку. Фреймворк Express використовується для створення веб-сервера та обробки HTTP-запитів, Mongoose — для роботи з базою даних MongoDB, CORS — для керування міждоменними запитами, а dotenv — для зберігання конфігураційних параметрів і секретів у змінних середовища. У результаті виконання цих команд проєкт готовий до написання першого серверного коду та подальшого розширення функціональності.

2. На цьому етапі необхідно налаштувати віддалену базу даних MongoDB, з якою буде працювати Node.js-застосунок. Для цього зареєструйтеся на платформі MongoDB Atlas за адресою https://cloud.mongodb.com/
 та створіть новий кластер (для навчальних цілей достатньо безкоштовного shared-кластера). Кластер є керованим середовищем, у якому зберігатимуться дані вашого застосунку.

Після створення кластера перейдіть у розділ Database Access, додайте нового користувача та задайте для нього ім’я і пароль. Ці облікові дані використовуватимуться серверним застосунком для підключення до бази даних. Далі в налаштуваннях кластера натисніть кнопку Connect, оберіть варіант підключення через application та збережіть згенеровану строку з’єднання (connection string).

Приклад строки з’єднання має такий вигляд:

```
mongodb+srv://<username>:<password>@cluster0.xxxxx.mongodb.net/mydb
```

У цій строкі замість <username> та <password> необхідно підставити створені раніше облікові дані користувача. Отриману строку з’єднання рекомендується зберігати у файлі .env та використовувати в коді застосунку через змінні середовища, що підвищує безпеку та спрощує конфігурацію проєкту.

3. На цьому етапі необхідно налаштувати конфігурацію серверного застосунку за допомогою змінних середовища. Відкрийте ваш

backend-проєкт у редакторі коду та створіть у кореневій директорії файл .env. Цей файл використовується для зберігання чутливих даних і налаштувань, які не повинні бути жорстко закодовані безпосередньо у вихідному коді. У файл .env додайте параметри для підключення до бази даних MongoDB та для визначення порту, на якому буде запущений сервер:

```
MONGO_URI=mongodb+srv://<username>:<password>@cluster0.xxxxx.mongodb
.net/mydb
PORT=3000
```

Замість значення MONGO_URI підставте власну строку з'єднання, отриману під час налаштування кластера MongoDB Atlas. Параметр PORT визначає порт, на якому працюватиме Node.js-сервер. Надалі ці значення зчитуються в застосунку за допомогою пакета dotenv, що дозволяє легко змінювати конфігурацію без редагування програмного коду та підвищує безпеку і переносимість застосунку.

4. На цьому етапі необхідно встановити основні npm-пакети, які забезпечують роботу серверної частини застосунку та взаємодію з базою даних. Для цього у кореневій директорії проєкту виконайте наступну команду:

```
npm install express mongoose cors dotenv
```

Пакет **Express** використовується для створення веб-сервера, визначення маршрутів та обробки HTTP-запитів (GET, POST, PUT, DELETE). Він є одним із найпопулярніших фреймворків для розробки backend-застосунків на Node.js.

Бібліотека **Mongoose** надає зручний інтерфейс для роботи з базою даних MongoDB, дозволяючи описувати схеми даних, виконувати запити та реалізовувати валідацію на рівні моделі.

Пакет **CORS** використовується для налаштування політики міждоменного доступу та дозволяє frontend-застосунку звертатися до backend-API, розташованого на іншому домені або порту.

Бібліотека **dotenv** дозволяє зчитувати змінні середовища з файлу .env, що використовується для зберігання конфігураційних параметрів, таких як строка підключення до бази даних або порт сервера.

Детальніше з документацією кожного з пакетів можна ознайомитися за наступними посиланнями:

https://www.npmjs.com/package/express
https://www.npmjs.com/package/mongoose
https://www.npmjs.com/package/cors
https://www.npmjs.com/package/dotenv

Після встановлення цих пакетів проєкт готовий до реалізації підключення до MongoDB та створення власного REST API.

5. На цьому етапі необхідно реалізувати логіку підключення до бази даних MongoDB. Для цього у проєкті створіть папку services (якщо вона ще не існує) та додайте до неї файл db.js. У цьому файлі буде зосереджена відповідальність за встановлення з'єднання з MongoDB, що відповідає принципу розділення відповідальностей у програмному коді.

У файлі db.js використовується бібліотека Mongoose, яка надає зручний асинхронний API для роботи з MongoDB. Функція connectDB приймає строку з'єднання як параметр, що дозволяє передавати її зі змінних середовища та легко змінювати конфігурацію без модифікації коду. Підключення виконується асинхронно з використанням async/await, а можливі помилки обробляються в блоці try/catch.

```
import mongoose from "mongoose";

export const connectDB = async (mongoURI) => {
    try {
        await mongoose.connect(mongoURI);
        console.log("MongoDB connected");
    } catch (err) {
        console.error("Mongo error:", err);
        throw err;
```

```
    }
};
```

У разі успішного підключення в консолі з’явиться повідомлення про успіх, що полегшує налагодження застосунку. Якщо ж з’єднання не вдається встановити, помилка буде виведена в консоль і передана далі, що дозволяє коректно обробити ситуацію на рівні запуску серверного застосунку.

6. Далі необхідно описати модель даних, яка визначає структуру документів у базі даних MongoDB. Для цього у папці models створіть файл Product.js. Модель відповідає за те, як дані зберігаються, перевіряються та обробляються на рівні бази даних, і є центральною частиною роботи з Mongoose.

У даному прикладі створюється модель Product, яка може використовуватися в онлайн-магазині. Схема (Schema) визначає набір полів документа, їх типи та правила валідації. Наприклад, для поля name задається обов’язковість, мінімальна довжина та автоматичне видалення зайвих пробілів, а для поля price — перевірка на додатне числове значення. Такі обмеження допомагають зберігати коректні та узгоджені дані в базі.

```
import mongoose from "mongoose";

// Define the schema (structure of a document saved to the mongo db)
const productSchema = new mongoose.Schema(
  {
    name: {
      type: String,
      required: [true, "Name is required"],
      trim: true, // removes extra spaces
      minlength: [2, "Name must be at least 2 characters long"]
    },
    price: {
      type: Number,
      required: [true, "Price is required"],
      min: 0,
    },
```

```
    description: {
      type: String,
      default: "",
      maxlength: [200, "Description too long"],
    },
    createdAt: {
      type: Date,
      default: Date.now,
    },
  },
  { versionKey: false } // disables the "__v" version field
);


// Export model so it can be used in routes or controllers
export const Product = mongoose.model("Product", productSchema);
```

Після створення схеми вона реєструється як модель за допомогою `mongoose.model`. Експортована модель **Product** може використовуватися у маршрутах або контролерах для виконання CRUD-операцій (створення, читання, оновлення та видалення товарів). За потреби ви можете розширювати цю схему або створювати додаткові моделі для інших сутностей вашого застосунку.

7. На цьому етапі необхідно реалізувати проміжний обробник (middleware) для валідації вхідних даних, які надходять до API під час створення або оновлення продуктів. Для цього у папці middleware створіть файл validate.js, у якому буде описана логіка перевірки коректності даних, отриманих із тіла HTTP-запиту.
Middleware validateProduct аналізує поля name та price, що передаються в req.body, і перевіряє їх на відповідність базовим вимогам. Якщо дані некоректні (наприклад, ім’я товару відсутнє або має замалу довжину, чи ціна є від’ємним або нечисловим значенням), сервер одразу повертає клієнту відповідь з кодом 400 (Bad Request) та повідомленням про помилку. Це дозволяє запобігти збереженню некоректних даних у базі ще до виконання бізнес-логіки.

```
export const validateProduct = (req, res, next) => {
  const { name, price } = req.body;

  if (!name || typeof name !== "string" || name.trim().length < 2) {
    return res
      .status(400)
      .json({ message: "Invalid product name. Minimum 2 characters
required." });
  }

  if (price === undefined || typeof price !== "number" || price < 0)
{
    return res.status(400).json({ message: "Invalid product price."
});
  }

  next(); // proceed to controller
};
```

У разі успішної перевірки викликається функція next(), яка передає керування наступному обробнику — зазвичай контролеру. Використання middleware для валідації сприяє чистій архітектурі застосунку, оскільки логіка перевірки даних відокремлюється від логіки обробки запитів і може повторно використовуватися в різних маршрутах.

8. На цьому етапі необхідно реалізувати контролер, який відповідатиме за бізнес-логіку роботи з продуктами та взаємодію з базою даних MongoDB. Для цього у папці controllers створіть файл productController.js. Контролер містить набір функцій, які обробляють HTTP-запити та виконують відповідні операції над колекцією продуктів. У файлі productController.js використовується модель Product, яка була визначена раніше за допомогою Mongoose. Кожна функція контролера реалізує одну з базових CRUD-операцій: отримання списку продуктів, створення нового продукту, оновлення існуючого та його видалення. Всі операції виконуються асинхронно з використанням async/await, що дозволяє коректно працювати з запитами до бази даних.

```
import { Product } from "../models/Product.js";

export const getProduct = async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};

export const createProduct = async (req, res) => {
  try {
    const product = new Product(req.body);
    await product.save();
    res.status(201).json(product);
  } catch (err) {
    res.status(400).json({ message: err.message });
  }
};

export const updateProduct = async (req, res) => {
  try {
    const { id } = req.params;

    // { new: true } returns UPDATED document
    const updated = await Product.findByIdAndUpdate(id, req.body, {
      new: true,
      runValidators: true,
    });

    if (!updated) {
      return res.status(404).json({ message: "Product not found" });
    }

    res.json(updated);
```

```
  } catch (err) {
    res.status(400).json({ message: err.message });
  }
};

export const deleteProduct = async (req, res) => {
  try {
    const { id } = req.params;

    const deleted = await Product.findByIdAndDelete(id);

    if (!deleted) {
      return res.status(404).json({ message: "Product not found" });
    }

    res.json({ message: "Product deleted successfully" });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};
```

Функція getProduct повертає список усіх продуктів із бази даних. createProduct створює новий запис на основі даних, отриманих із тіла запиту, та зберігає його в MongoDB. Функція updateProduct оновлює існуючий продукт за ідентифікатором, а параметр runValidators забезпечує повторну перевірку даних відповідно до схеми. Функція deleteProduct видаляє продукт з бази даних.

Такий підхід дозволяє чітко відокремити логіку обробки запитів від маршрутизації та забезпечує масштабовану й зрозумілу структуру backend-застосунку.

9. Далі необхідно реалізувати маршрути (routes) для роботи з продуктами та пов’язати їх із відповідними контролерами і middleware. Для цього у папці routes створіть файл productRoutes.js. Маршрути визначають, які HTTP-запити обробляє сервер і яку логіку слід виконати для кожного з них. У файлі productRoutes.js використовується об’єкт Router

з фреймворку Express, який дозволяє логічно групувати маршрути за певною сутністю — у даному випадку продуктами. Кожен маршрут викликає відповідну функцію з контролера productController, а для операцій створення та оновлення додатково підключається middleware для валідації вхідних даних.

```
import express from "express";
import { getProducts, createProduct } from
"controllers/productController.js";
import { validateProduct } from "../middleware/validateProduct.js";

const router = express.Router();

router.get("/", getProducts);
router.post("/", validateProduct, createProduct);
router.put("/:id", validateProduct, updateProduct);     // or
router.patch
router.delete("/:id", deleteProduct);

export default router;
```

Маршрут GET / використовується для отримання списку всіх продуктів. Маршрут POST / дозволяє створити новий продукт і перед виконанням контролера перевіряє коректність даних за допомогою validateProduct. Запити PUT /:id (або PATCH) оновлюють існуючий продукт за його ідентифікатором, а DELETE /:id видаляє продукт із бази даних.

Використання окремого файлу маршрутів спрощує структуру проєкту, робить код більш читабельним і полегшує подальше розширення API.

10. На цьому етапі відбувається запуск серверного застосунку та підключення всіх необхідних компонентів. Для цього у кореневій директорії створіть файл server.js. У ньому налаштовується веб-сервер, політика доступу (CORS), маршрути API, підключення до бази даних і запуск серверу на визначеному порті.

```
import express from "express";
```

```
import cors from "cors";
import dotenv from "dotenv";
import { connectDB } from "./services/db.js";
import productRoutes from "productRoutes.js";

dotenv.config();
const app = express();

// CORS setup (allow only your UI domain)
const allowedOrigins = process.env.CLIENT_ORIGINS.split(",");
app.use(
  cors({
    origin: (origin, callback) => {
      // allow REST tools like Postman (no origin)
      if (!origin) return callback(null, true);

      if (allowedOrigins.includes(origin)) {
        return callback(null, true);
      }

      callback(new Error("Not allowed by CORS"));
    },
    credentials: true,
    methods: ["GET", "POST", "PUT", "DELETE"],
  })
);

// Middleware
app.use(express.json());

// Routes
app.use("/products", productRoutes);

// DB Connection
const dbURI = process.env.MONGO_URI;
connectDB(dbURI);
```

```
// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```

11. Зробіть push ваших змін в гітхаб репозиторій, та зайдіть на https://dashboard.render.com/ (попередньо зареєструвавшись, якщо ви іще не зареєстровані на цьому сервісі). Клікніть new web service, додайте лінку на ваш репозиторій.
12. В графі Build Command введіть ‘npm install’, Start Command: `node server.js`. Додайте змінні оточення (строку з'єднання до кластера mongo db та порт). Клікніть Deploy Web Service та почекайте, поки ваш сервіс не буде на хост сервері.
13. Інтегруйте фронтенд-частину із лабораторних 3 або 4 до бекенд частини. Там, де в фронтенд додатку використовувалася стороння api або ж inmemory, використовуйте запити на власний бекенд сервіс.
14. Для того щоб дозволити cross-origin запити пропишіть оріджін запитів. Наприклад, якщо ваш фронтенд додаток хоститься на https://example.com, а також ви локально запускаєте його на http://localhost:4200, можна прописати це наступним чином в .env:

```
CLIENT_ORIGINS=http://localhost:4200,https://example.com
```

та в server.js:

```
const allowedOrigins = process.env.CLIENT_ORIGINS.split(",");

app.use(
  cors({
    origin: (origin, callback) => {
      // allow REST tools like Postman (no origin)
      if (!origin) return callback(null, true);

      if (allowedOrigins.includes(origin)) {
        return callback(null, true);
      }
```

```
      callback(new Error("Not allowed by CORS"));
    },
    credentials: true,
    methods: ["GET", "POST", "PUT", "DELETE"],
  })
);
```

не рекомендуєтсья дозволяти cors для всіх ресурсів, проте ви можете зробити це наступним чином:

```
const allowedOrigin = “*”;
```

15. (За бажанням) Додайте JWT-аутентифікацію до вашого бекенд додатку. Для цього встановіть наступний пакет:

npm i jsonwebtoken

Приклад middleware для аутентифікації (перевіряє наявність jwt-токену в запиті):

```
import jwt from "jsonwebtoken";

export const authRequired = (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (!authHeader?.startsWith("Bearer ")) {
    return res.status(401).json({ message: "No token provided" });
  }

  const token = authHeader.split(" ")[1];

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // attach user to request
    next();
  } catch {
    res.status(401).json({ message: "Invalid or expired token" });
  }
};
```

Приклад ендпоінту для авторизації (authController.js вертає jwt-токен, отримавши в запиті username та password, а також має метод для реєстрації нового юзера)

```
import jwt from "jsonwebtoken";
import bcrypt from "bcrypt";
import { User } from "../models/User.js";

export const register = async (req, res) => {
  try {
    const { userName, password } = req.body;

    const existingUser = await User.findOne({ userName });
    if (existingUser) {
      return res.status(400).json({ message: "User already exists"
});
    }

    const hashedPassword = await bcrypt.hash(password, 10);

    const newUser = new User({
      userName,
      password: hashedPassword,
    });
    await newUser.save();

    const token = jwt.sign(
      { userId: newUser._id, userName: newUser.userName },
      process.env.JWT_SECRET,
      { expiresIn: "1h" }
    );

    res.status(201).json({
      message: "User registered successfully",
      token,
    });
  } catch (err) {
```

```
    console.error("Registration error:", err);
    res.status(500).json({ message: "Server error" });
  }
};

export const login = async (req, res) => {
  try {
    const { userName, password } = req.body;

    const user = await User.findOne({ userName });
    if (!user) {
      return res.status(401).json({ message: "Invalid credentials"
});
    }

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(401).json({ message: "Invalid credentials"
});
    }

    const token = jwt.sign(
      { userId: user._id, userName: user.userName },
      process.env.JWT_SECRET,
      { expiresIn: "1h" }
    );

    res.json({ token });
  } catch (err) {
    console.error("Login error:", err);
    res.status(500).json({ message: "Server error" });
  }
};
```

Приклад використання в роутах, в роуті authRoutes.js:

```
import express from "express";
import { register, login } from "../controllers/authController.js";

const router = express.Router();

router.post("/register", register);
router.post("/login", login);

export default router;
```

Та в роуті productRoutes.js:

```
import express from "express";
import { getProducts, createProduct, updateProduct, deleteProduct }
from "../controllers/productController.js";
import { validateProduct } from "../middleware/validateProduct.js";
import { authRequired } from "../middleware/authMiddleware.js";

const router = express.Router();

router.get("/", getProducts);
router.post("/", authRequired, validateProduct, createProduct);
router.put("/:id", authRequired, validateProduct, updateProduct);
// or router.patch
router.delete("/:id", authRequired, deleteProduct);

export default router;
```

Та в server.js:

```
import authRoutes from "./auth/authRoutes.js";
import { authRequired } from "./middleware/authMiddleware.js";
import productRoutes from "./routes/productRoutes.js";
```

```
app.use("/auth", authRoutes); // auth routes

app.use("/products", productRoutes); // get is open
```

Не забудьте додати ключ в .env файл:

```
JWT_SECRET=super_secret_key__for_your_app
```

16. (За бажанням). Додайте Dockerfile (файл саме з такою назвою) наступного наповнення до вашого проекту:

```
FROM node
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
CMD ["node","server.js"]
```

Контейнеризуйте ваш бекенд додаток за допомогою наступної команди (попередньо встановивши докер в вашому середовищі):

docker build -t node-application .

Запушіть Dockerfile в ваш репозиторій, після цього виберіть в render.com New Web Service та, серед запропонованих опцій деплою, Docker. Через деякий час ви отримаєте лінку на ваш задеплоєний контейнер.

17. Додайте запити до бекенду в ваш фронтенд-додаток. Приклад реакт-компонент, що роблять запити до бекенду для виведення даних:

```
import { useEffect, useState } from "react";
import axios from "axios";

export default function ProductsList() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    axios.get("http://your-deployed-application/api/products")
      .then(res => setProducts(res.data))
      .catch(err => console.error(err));
```

```
  }, []);

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {products.map(p => (
          <li key={p._id}>
            {p.name} — ${p.price}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Та для додавання нових даних через POST запит:

```
import { useState } from "react";
import axios from "axios";

export default function AddProduct() {
  const [name, setName] = useState("");
  const [price, setPrice] = useState("");

  const handleSubmit = async (e) => {
    e.preventDefault();

    try {
      const token = localStorage.getItem("token"); // get previously
saved JWT

      await axios.post(
        "http://localhost:5000/api/products",
        { name, price },
        {
          headers: {
            Authorization: `Bearer ${token}`
          }
        }
      );

      setName("");
```

```
      setPrice("");
      alert("Product added!");
    } catch (err) {
      console.error(err);
      alert("Error adding product");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <h2>Add Product</h2>

      <input
        placeholder="Name"
        value={name}
        onChange={e => setName(e.target.value)}
      />

      <input
        placeholder="Price"
        type="number"
        value={price}
        onChange={e => setPrice(e.target.value)}
      />

      <button type="submit">Add</button>
    </form>
  );
}
```

В цих прикладах використовується npm бібліотека axios (npm i axios).