МІНІСТЕРСТВО ОСВІТИ УКРАЇНИ

Івано-Франківський національний технічний

Університет нафти і газу

Л.І.Григорчук, Г.В.Григорчук В.В.Бандура

Web‑базовані інтелектуальні додатки та сервіси

Методичні вказівки для самостійної роботи

для студентів стаціонарної та заочної форми навчання

спеціальності 121 – інженерія програмного забезпечення

Івано – Франківськ 2025

**Григорчук Л.І. Григорчук Г.В. Бандура В.В.**  Web‑базовані інтелектуальні додатки та сервіси. Методичні вказівки для самостійної роботи. – Івано-Франківськ: 2025.- с.104

Дисципліна базується на практичних завданнях, тісно наближених до реальних кейсів із сучасної веб-розробки. Усі лабораторні роботи передбачають створення робочих проєктів, тестування, деплой і обов’язкову підготовку звітів із прикладами реалізації та скріншотами. Отримані знання та навички дозволять студентам не лише впевнено орієнтуватися в сучасних технологіях веб-розробки, але й створювати інтелектуальні додатки, здатні інтегруватися з різними сервісами, аналізувати дані та забезпечувати якісний користувацький досвід.

Методичні вказівки розраховані на студентів спеціальності інженерія програмного забезпечення

Завідувач кафедри

Інженерії програмного забезпечення В.В. Бандура

Член експертно-рецензійної

комісії                                       В. В. Бандура

 Бібліотекар 1 категорії І.С. Макар

**Вступ.** Сучасні веб-додатки дедалі частіше поєднують інтелектуальні функції, інтеграцію з хмарними сервісами та обробку великих обсягів даних у реальному часі. Мета дисципліни **«Web-базовані інтелектуальні додатки та сервіси»** — сформувати в студентів комплексні знання та практичні навички, необхідні для проєктування, розробки та розгортання веб-застосунків із використанням сучасних технологій.

У межах курсу студенти отримають досвід роботи з **фронтендом, бекендом, базами даних, API, хмарними сервісами, аналітикою та машинним навчанням**. Послідовність лабораторних робіт побудована за принципом поступового занурення: від створення статичних веб-сторінок до розробки повноцінних інтелектуальних сервісів із рекомендаційними системами, чат-ботами та інтеграціями зі сторонніми API.

Під час виконання лабораторних робіт студенти відпрацьовують та поглиблюють навички з таких напрямків:

* робота зі стеком **HTML5/CSS3/JavaScript** та фреймворками **React/Vue** для створення інтерактивних SPA;
* проєктування та реалізація **REST** та **GraphQL API** для обміну даними між клієнтом і сервером;
* налаштовувати **баз даних** (PostgreSQL, MongoDB) та інтеграція із застосунками;
* **деплой** сервісів на **Heroku, Render, AWS** і використання змінних середовища;
* підключення **NLP-моделі** та створення **рекомендаційних систем** та **чат-боти**;
* аналіз даних користувацької активності та побудова **дашбордів** з використанням **Chart.js/D3.js**;
* застосовування принципів **оптимізації** та **захисту веб-додатків**: кешування, rate limiting, JWT-аутентифікація, шифрування даних;
* розробка фінальних проєктів, документування API за допомогою **Swagger/OpenAPI** і презентація результатів роботи.

Дисципліна базується на практичних завданнях, тісно наближених до реальних кейсів із сучасної веб-розробки. Усі лабораторні роботи передбачають створення робочих проєктів, тестування, деплой і обов’язкову підготовку звітів із прикладами реалізації та скріншотами.

Отримані знання та навички дозволять студентам не лише впевнено орієнтуватися в сучасних технологіях веб-розробки, але й створювати інтелектуальні додатки, здатні інтегруватися з різними сервісами, аналізувати дані та забезпечувати якісний користувацький досвід.

# Зміст

1. Лабораторна робота №1. . Розробка клієнтської частини веб-застосунків.............................................................................................. 3

2. Лабораторна робота №2. Дослідження клієнт–серверної взаємодії та роботи з API…………………………………........................................ 8

3. Лабораторна робота №3. Проєктування та реалізація серверної частини веб-застосунків....................................................................... 22

4. Лабораторна робота №4. Інтеграція інтелектуальних сервісів (AI API) ........................................................................................................ 37

5. Лабораторна робота №5. Методи обробки та управління даними користувача........................................................................................... 49

6. Лабораторна робота №6. Застосування базових моделей машинного навчання ............................................................................................... 58

7. Лабораторна робота №7. Використання хмарних сервісів та деплоймент застосунків....................................................................... 65

8. Лабораторна робота №8. Розробка чат-ботів або рекомендаційної системи ................................................................................................ 74

9. Лабораторна робота №9. Аналітика користувацької активності та візуалізація даних ............................................................................... 80

10. Лабораторна робота №10. . Інтеграція веб-застосунків із зовнішніми сервісами.............................................................................................. 86

11. Лабораторна робота №11. . Оптимізація продуктивності та забезпечення безпеки веб-застосунків.............................................. 91

12. Лабораторна робота №12. Підсумковий проєкт: розробка, документування та презентація......................................................... 96

# Лабораторна робота 1. Розробка клієнтської частини веб-застосунків

Дисципліна: Web‑базовані інтелектуальні додатки та сервіси

Перша лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* має ознайомчий характер і спрямована на формування базових умінь у сфері веб-розробки. Вона закладає фундамент для подальших робіт курсу, де будуть розглядатися складніші аспекти клієнт–серверної взаємодії, інтеграції інтелектуальних сервісів, безпеки та хмарних технологій.

У процесі виконання цієї роботи студенти:

* дізнаються про основні принципи роботи браузера та структуру веб-сторінки;
* навчаться застосовувати HTML для створення семантичної розмітки контенту;
* опанують базові можливості CSS для стилізації сторінок та створення адаптивного дизайну;
* закріплять навички використання JavaScript для організації інтерактивності та обробки подій;
* ознайомляться з Git як інструментом контролю версій і навчаться публікувати власні проєкти за допомогою GitHub Pages.

Практична цінність цієї роботи полягає в тому, що студенти отримають перший досвід створення робочого середовища розробника, організації структури веб-додатку та базової інтерактивності. Цей досвід є необхідним для подальшого розгортання більш складних проєктів і стане підґрунтям для формування професійних навичок у сфері веб-інженерії.

Тривалість: 1 лабораторне заняття  
Формат звіту: PDF або DOCX + посилання на репозиторій GitHub і опубліковану сторінку.

## Мета роботи

* Удосконалити знання зі стеком HTML5/CSS3/JavaScript і основами REST API.
* Налаштувати робоче середовище: VS Code, Git, Node.js (або Python), локальний сервер.
* Створити мінімальний веб‑проєкт, запустити локально, завантажити в GitHub і опублікувати на GitHub Pages або Vercel.

## Результати навчання

* Встановлювати й налаштовувати VS Code, Git, Node/Python.
* Працювати з Git та GitHub (clone, commit, push).
* Створювати базову сторінку HTML/CSS/JS і запускати її на локальному сервері.
* Виконувати прості HTTP‑запити через fetch() та публікувати сайт.

## Попередні вимоги

* Базові навички роботи з терміналом та файловою системою.
* Обліковий запис GitHub (обов’язково), Vercel — за бажанням.

## Необхідне ПЗ

* VS Code (Live Server, Prettier, HTML CSS Support).
* Git (Windows: Git Bash).
* Node.js LTS з npm або Python 3.x.
* Сучасний браузер.

## Короткі теоретичні відомості

* HTML5 — семантичні теги (header, nav, main, section, footer).
* CSS3 — макети (flex, grid), адаптивність, медіа‑запити.
* JavaScript — DOM, події, fetch() і JSON.
* REST API — HTTP‑методи (GET/POST/PUT/DELETE), коди стану.
* Локальний сервер — віддача файлів на http://localhost:<порт>.

## Хід виконання

### Крок 1. Перевірка та встановлення інструментів

У терміналі перевірте версії:

node -v && npm -v

python --version

git --version

Встановіть відсутні інструменти. У VS Code додайте Live Server, Prettier, HTML CSS Support.

Налаштуйте Git:

git config --global user.name "Ваше Ім’я"

git config --global user.email "you@example.com"

### Крок 2. Створення репозиторію

Створіть репозиторій lab1-web-intro на GitHub, клонування:

git clone https://github.com/<user>/lab1-web-intro.git

cd lab1-web-intro

### Крок 3. Структура проєкту і файли

Створіть index.html, css/style.css, js/app.js.

index.html підключає style.css і app.js; додайте кнопку для запиту до API.

### Крок 4. Запуск локального сервера

Варіант А — Live Server у VS Code.

Варіант Б — Python: python -m http.server 5500

Варіант В — Node: npx serve -p 5500 .

### Крок 5. Перший коміт і пуш

git add .

git commit -m "Init: базова сторінка, стилі, JS"

git push origin main

### Крок 6. Публікація сайту

GitHub Pages: Settings → Pages → Deploy from a branch (main /root) або Vercel: Import Git Repository → Deploy.

### Крок 7. Перевірка та самооцінка

Сайт відкривається локально і за публічним URL; кнопка завантажує JSON; стилі підключені; HTML семантичний.

## Приклади файлів

index.html

<!doctype html>  
<html lang="uk">  
<head>  
 <meta charset="utf-8" />  
 <meta name="viewport" content="width=device-width, initial-scale=1" />  
 <title>ЛР1 — Ваша сторінка</title>  
 <link rel="stylesheet" href="./css/style.css" />  
</head>  
<body>  
 <header>  
 <h1>Вітаю у ЛР1</h1>  
 <nav>  
 <a href="#about">Про роботу</a>  
 <a href="#api">Дані з API</a>  
 </nav>  
 </header>  
 <main>  
 <section id="about">  
 <p>Це базова сторінка з HTML/CSS/JS.</p>  
 </section>  
 <section id="api">  
 <h2>REST API приклад</h2>  
 <button id="load">Завантажити факт</button>  
 <pre id="out">(тут з’являться дані)</pre>  
 </section>  
 </main>  
 <footer>© 2025, Ваше ПІБ</footer>  
 <script src="./js/app.js"></script>  
</body>  
</html>  
css/style.css:

:root { font-family: system-ui, Arial, sans-serif; }  
body { max-width: 960px; margin: 0 auto; padding: 16px; line-height: 1.5; }  
header, footer { padding: 12px 0; }  
nav a { margin-right: 12px; text-decoration: none; }  
button { cursor: pointer; padding: 8px 12px; }  
pre { background: #f4f4f4; padding: 12px; border-radius: 8px; }

js/app.js:

document.getElementById("load").addEventListener("click", async () => {  
 const out = document.getElementById("out");  
 out.textContent = "Завантаження...";  
 try {  
 const res = await fetch("https://jsonplaceholder.typicode.com/todos/1");  
 const data = await res.json();  
 out.textContent = JSON.stringify(data, null, 2);  
 } catch (e) {  
 out.textContent = "Помилка завантаження: " + e.message;  
 }  
});

## Індивідуальні завдання (оберіть одне)

1. Адаптивна шапка (flex або grid, 2 брейкпойнти).
2. Форма з валідацією на JS (e‑mail/телефон).
3. Дані з іншого публічного API (мінімум 3 елементи).
4. Якірна навігація, плавний скрол і «кнопка нагору».

## Звіт: обов’язкові скріншоти

* Версії інструментів у терміналі (node/npm або python, git).
* Розширення VS Code (Live Server, Prettier, HTML CSS Support).
* Сторінка репозиторію на GitHub.
* Структура проєкту у VS Code.
* Вміст index.html (з підключеннями).
* Вміст style.css та app.js.
* Запуск локального сервера (термінал або Go Live).
* Браузер з http://localhost:... і JSON після натискання кнопки.
* Commits у GitHub.
* Налаштування GitHub Pages або сторінка деплою Vercel.
* Публічний URL сайту з вашим ПІБ у футері.

## Контрольні питання

1. Різниця між <section>, <article>, <div>.
2. Призначення meta viewport і семантики HTML.
3. Налаштування імені та e‑mail у Git і навіщо це потрібно.
4. Способи підняти локальний сервер для статичного сайту.
5. Що таке REST і типові коди стану HTTP.
6. Відмінності GitHub Pages та Vercel для статичних сайтів.

# Лабораторна робота 2. Дослідження клієнт–серверної взаємодії та роботи з API

Дисципліна: Web-базовані інтелектуальні додатки та сервіси  
Тривалість: 2 пари  
Формат звіту: PDF/DOCX + посилання на репозиторій і опублікований демо.

## Мета роботи

* Засвоїти базові підходи до розробки SPA‑інтерфейсів у React або Vue.js.
* Навчитися динамічному рендерингу даних (стан, властивості, списки, події, форми).
* Підключити UI‑бібліотеку компонентів (MUI/Ant Design / Vuetify тощо).
* Організувати компонентну архітектуру та роутинг.

## Результати навчання

* Ініціалізувати проєкт на Vite (React/Vue) і запускати dev‑сервер.
* Створювати функціональні компоненти, працювати зі станом і подіями.
* Рендерити списки з API; робити керовані форми та фільтрацію.
* Інтегрувати UI‑бібліотеку та налаштовувати тему.
* Налаштовувати роутинг (React Router/Vue Router), збирати й деплоїти застосунок.

## Попередні вимоги

* Виконана ЛР1; встановлені Node.js LTS, Git, VS Code; базові HTML/CSS/JS; обліковий запис GitHub.

## Необхідне ПЗ

* Node.js LTS (npm).
* VS Code (ESLint, Prettier, Auto Rename Tag).
* Сучасний браузер.

## Короткі теоретичні відомості

* SPA/CSR: одна HTML‑сторінка з оновленням контенту клієнтським JS.
* React: функціональні компоненти, JSX, хуки useState/useEffect.
* Vue: реактивність ref/reactive, <template> + <script setup>, життєвий цикл onMounted.
* Динамічний рендеринг: списки (map/v‑for), умовний рендеринг, керовані інпути (або v‑model).
* UI‑бібліотеки: готові компоненти (кнопки, картки, таблиці, модальні).
* Роутинг: react-router-dom або vue-router.

## Хід виконання

### Крок 1. Створення проєкту на Vite та запуск

Перевірити версії:

node -v

npm -v

React:

npm create vite@latest lab2-ui-react -- --template react

cd lab2-ui-react

npm i

npm run dev

(або --template react-ts)

Vue:

npm create vite@latest lab2-ui-vue -- --template vue

cd lab2-ui-vue

npm i

npm run dev

(або --template vue-ts)

### Крок 2. Базова структура та компоненти

Рекомендована структура: src/components, src/pages, src/services, src/App.jsx|App.vue, src/main.jsx|main.js.

Створіть компонент UserCard та сторінку Users.

### Крок 3. Динамічний рендеринг і робота з API

Завантажте користувачів з https://jsonplaceholder.typicode.com/users і відобразіть у вигляді карток.

Додайте поле пошуку/фільтрацію.

### Крок 4. Роутинг (мінімум 2 сторінки)

React:

npm i react-router-dom

Налаштуйте RouterProvider з маршрутами '/', '/users'.

Vue:

npm i vue-router

Налаштуйте createRouter з маршрутами '/', '/users'.

### Крок 5. Підключення UI‑бібліотеки

React (MUI):

npm i @mui/material @emotion/react @emotion/styled @mui/icons-material

Використайте Button/Card тощо.

Vue (Vuetify 3):

npm i vuetify

Імпортуйте 'vuetify/styles' і створіть createVuetify().

### Крок 6. Архітектура

Винесіть запити в src/services/api.js, розбийте інтерфейс на дрібні перевикористовувані компоненти, додайте ESLint + Prettier.

За потреби використайте Context/Pinia для глобального стану.

### Крок 7. Збірка та деплой

npm run build

Vercel/Netlify: імпорт репозиторію; фреймворк Vite; output 'dist'.

GitHub Pages: використайте GitHub Actions або плагін gh-pages.

## Фрагменти коду (React)

src/components/UserCard.jsx

export default function UserCard({ user }) {  
 return (  
 <div className="card">  
 <h3>{user.name}</h3>  
 <p>{user.email}</p>  
 </div>  
 );  
}

src/pages/Users.jsx

import { useEffect, useState } from "react";  
import UserCard from "../components/UserCard.jsx";  
export default function Users(){  
 const [users,setUsers] = useState([]);  
 const [query,setQuery] = useState("");  
 useEffect(()=>{  
 fetch("https://jsonplaceholder.typicode.com/users")  
 .then(r=>r.json()).then(setUsers);  
 },[]);  
 const filtered = users.filter(u=>u.name.toLowerCase().includes(query.toLowerCase()));  
 return (  
 <main>  
 <input placeholder="Пошук" value={query} onChange={e=>setQuery(e.target.value)} />  
 <div className="grid">  
 {filtered.map(u => <UserCard key={u.id} user={u} />)}  
 </div>  
 </main>  
 );  
}

Роутинг React (main.jsx)

import { createRoot } from 'react-dom/client';  
import { createBrowserRouter, RouterProvider } from 'react-router-dom';  
import App from './App.jsx';  
import Users from './pages/Users.jsx';  
const router = createBrowserRouter([  
 { path: '/', element: <App/> },  
 { path: '/users', element: <Users/> }  
]);  
createRoot(document.getElementById('root')).render(<RouterProvider router={router} />);

## Фрагменти коду (Vue)

src/components/UserCard.vue

<template>  
 <div class="card">  
 <h3>{{ user.name }}</h3>  
 <p>{{ user.email }}</p>  
 </div>  
</template>  
<script setup>  
defineProps({ user: Object })  
</script>  
<style scoped>  
.card{padding:12px;border:1px solid #eee;border-radius:8px}  
</style>

src/pages/Users.vue

<template>  
 <main>  
 <input v-model="query" placeholder="Пошук" />  
 <div class="grid">  
 <UserCard v-for="u in filtered" :key="u.id" :user="u" />  
 </div>  
 </main>  
</template>  
<script setup>  
import { ref, computed, onMounted } from 'vue'  
import UserCard from '../components/UserCard.vue'  
const users = ref([])  
const query = ref("")  
onMounted(async()=>{  
 users.value = await (await fetch('https://jsonplaceholder.typicode.com/users')).json()  
})  
const filtered = computed(()=> users.value.filter(u=>u.name.toLowerCase().includes(query.value.toLowerCase())))  
</script>

Роутинг Vue (router.js)

import { createRouter, createWebHistory } from 'vue-router'  
import Home from './App.vue'  
import Users from './pages/Users.vue'  
export default createRouter({  
 history: createWebHistory(),  
 routes: [ { path: '/', component: Home }, { path: '/users', component: Users } ]  
})

## Індивідуальні завдання (оберіть одне)

1. Додати форму створення користувача (валідація, повідомлення про помилки).
2. Зробити таблицю з пагінацією/сортуванням (можна компонент UI‑бібліотеки).
3. Реалізувати світлу/темну тему.
4. Додати пошук із debounce (300–500 мс).
5. Винести роботу з API у сервіс + обробка помилок/loader.

## Звіт: обов’язкові скріншоти

* Версії Node/npm і встановлення Vite (термінал).
* Структура src/ у VS Code.
* Працюючий список користувачів (рендеринг карток/таблиці).
* Підключена UI‑бібліотека (видимі компоненти та тема).
* Роутинг між сторінками Home ↔ Users.
* Вивід збірки npm run build.
* Сторінка деплою у Vercel/Netlify або GitHub Pages.
* Публічний URL застосунку.

## Контрольні питання

1. Що таке компонент і чим props відрізняються від стану?
2. useState/useEffect (React) та ref/reactive/onMounted (Vue) — роль і відмінності.
3. Як реалізується умовний та списковий рендеринг?
4. Базова конфігурація роутера у SPA.
5. Переваги/ризики використання UI‑бібліотек.
6. Організація структури проєкту і розділення відповідальностей.

**Лабораторна робота №3**

«Проєктування та реалізація серверної частини веб-застосунків»

**Вступ.** Третя лабораторна робота дисципліни «Web-базовані інтелектуальні додатки та сервіси» присвячена основам створення серверної частини веб-застосунків. Якщо попередні завдання були зосереджені на клієнтській частині та взаємодії з API, то ця робота вводить студентів у сферу бекенд-розробки.

Основна увага приділяється побудові REST API та інтеграції з базами даних. Це дозволяє студентам зрозуміти, як реалізується обробка запитів користувачів, збереження та управління інформацією, а також як забезпечити стабільність і масштабованість веб-застосунків.

У процесі виконання роботи студенти навчаться:

* створювати проєкт серверного застосунку (Node.js/Express або Python/Flask/Django);
* реалізовувати CRUD-операції та організовувати структуру коду (routes, controllers, models);
* підключати PostgreSQL або MongoDB і налаштовувати схеми даних;
* застосовувати змінні середовища для безпечного зберігання конфігурацій;
* тестувати API за допомогою Postman, створюючи колекції запитів та автоматичні тести.

Практична цінність цієї роботи полягає в тому, що студенти отримують перший досвід у сфері серверної розробки, що є ключовим етапом у формуванні повноцінних навичок веб-інженера. Цей досвід стане основою для подальшої роботи з інтеграцією інтелектуальних сервісів, оптимізацією та забезпеченням безпеки веб-додатків.

# 1. Мета та результати навчання

Мета: засвоїти підходи до побудови простої серверної частини застосунку з REST API, підключенням до бази даних (PostgreSQL або MongoDB) та первинним тестуванням запитів через Postman.

Після виконання роботи студент повинен уміти:

* ініціалізувати проєкт і налаштувати середовище (Node.js/Express або Python/Flask/Django);
* створити REST API для базових CRUD-операцій;
* під’єднатися до PostgreSQL або MongoDB, виконати міграцію/створити схему;
* написати мінімальні тести запитів у Postman (колекція + змінні оточення);
* оформити звіт із результатами тестування.

# 2. Попередні вимоги

* Встановлені інструменти: Git, Node.js LTS або Python (3.10+), пакетний менеджер (npm/pip), Postman.
* База даних: PostgreSQL 14+ або MongoDB 6+ (локально чи в Docker).
* ОС: Windows 11 або Ubuntu 24.04.

# 3. Короткі теоретичні відомості

* REST API — інтерфейс взаємодії клієнт–сервер із ресурсами (endpoint’и: /api/items, методи: GET/POST/PUT/DELETE).
* ORM/ODM: спрощує роботу з БД (Sequelize/Prisma для Node.js; SQLAlchemy для Python; Mongoose для MongoDB).
* ENV змінні: зберігати рядки підключення, порти, секрети поза кодом (.env).
* Структура репозиторію: src/, routes/, controllers/, models/, tests/, .env.example.

# 4. Варіант A: Node.js / Express + PostgreSQL

**4.1. Ініціалізація проєкту**

# Windows/Ubuntu  
mkdir lab3-express && cd lab3-express  
npm init -y  
npm i express pg pg-h store dotenv cors  
npm i -D nodemon

**4.2. Структура папок**

lab3-express/  
 ├─ src/  
 │ ├─ index.js  
 │ ├─ routes/  
 │ │ └─ books.routes.js  
 │ ├─ controllers/  
 │ │ └─ books.controller.js  
 │ └─ db.js  
 ├─ .env  
 ├─ .env.example  
 └─ package.json

**4.3. Файл .env (приклад)**

PORT=3000  
PGHOST=localhost  
PGPORT=5432  
PGDATABASE=lab3  
PGUSER=labuser  
PGPASSWORD=labpass

**4.4. Налаштування БД PostgreSQL (SQL)**

CREATE DATABASE lab3;  
CREATE USER labuser WITH ENCRYPTED PASSWORD 'labpass';  
GRANT ALL PRIVILEGES ON DATABASE lab3 TO labuser;  
  
-- У самій БД:  
CREATE TABLE books (  
 id SERIAL PRIMARY KEY,  
 title TEXT NOT NULL,  
 author TEXT NOT NULL,  
 year INT,  
 created\_at TIMESTAMP DEFAULT NOW()  
);

**4.5. Підключення до БД (src/db.js)**

const { Pool } = require('pg');  
require('dotenv').config();  
const pool = new Pool({  
 host: process.env.PGHOST,  
 port: process.env.PGPORT,  
 database: process.env.PGDATABASE,  
 user: process.env.PGUSER,  
 password: process.env.PGPASSWORD,  
});  
module.exports = { pool };

**4.6. Точка входу (src/index.js)**

const express = require('express');  
const cors = require('cors');  
require('dotenv').config();  
const app = express();  
app.use(cors());  
app.use(express.json());  
  
app.get('/api/health', (req, res) => res.json({ status: 'ok' }));  
  
const booksRouter = require('./routes/books.routes');  
app.use('/api/books', booksRouter);  
  
const port = process.env.PORT || 3000;  
app.listen(port, () => console.log(`Server running on http://localhost:${port}`));

**4.7. Роути та контролери**

// src/routes/books.routes.js  
const router = require('express').Router();  
const ctrl = require('../controllers/books.controller');  
router.get('/', ctrl.getAll);  
router.get('/:id', ctrl.getOne);  
router.post('/', ctrl.create);  
router.put('/:id', ctrl.update);  
router.delete('/:id', ctrl.remove);  
module.exports = router;  
  
// src/controllers/books.controller.js  
const { pool } = require('../db');  
exports.getAll = async (req, res) => {  
 const { rows } = await pool.query('SELECT \* FROM books ORDER BY id');  
 res.json(rows);  
};  
exports.getOne = async (req, res) => {  
 const { id } = req.params;  
 const { rows } = await pool.query('SELECT \* FROM books WHERE id=$1', [id]);  
 rows[0] ? res.json(rows[0]) : res.status(404).json({ message: 'Not found' });  
};  
exports.create = async (req, res) => {  
 const { title, author, year } = req.body;  
 const { rows } = await pool.query(  
 'INSERT INTO books(title, author, year) VALUES ($1,$2,$3) RETURNING \*',  
 [title, author, year]  
 );  
 res.status(201).json(rows[0]);  
};  
exports.update = async (req, res) => {  
 const { id } = req.params; const { title, author, year } = req.body;  
 const { rowCount, rows } = await pool.query(  
 'UPDATE books SET title=$1, author=$2, year=$3 WHERE id=$4 RETURNING \*',  
 [title, author, year, id]  
 );  
 rowCount ? res.json(rows[0]) : res.status(404).json({ message: 'Not found' });  
};  
exports.remove = async (req, res) => {  
 const { id } = req.params;  
 const r = await pool.query('DELETE FROM books WHERE id=$1', [id]);  
 r.rowCount ? res.status(204).end() : res.status(404).json({ message: 'Not found' });  
};

**4.8. Скрипти запуску (package.json – фрагмент)**

"scripts": {  
 "dev": "nodemon src/index.js",  
 "start": "node src/index.js"  
}

# 5. Варіант B: Python / Flask + PostgreSQL

**5.1. Віртуальне середовище і залежності**

mkdir lab3-flask && cd lab3-flask  
python3 -m venv venv  
source venv/bin/activate # Windows: venv\Scripts\activate  
pip install flask psycopg2-binary python-dotenv flask-cors

**5.2. Структура**

lab3-flask/  
 ├─ app.py  
 ├─ db.py  
 ├─ .env  
 ├─ .env.example  
 └─ requirements.txt

**5.3. .env (приклад)**

FLASK\_ENV=development  
PORT=3001  
PGHOST=localhost  
PGPORT=5432  
PGDATABASE=lab3  
PGUSER=labuser  
PGPASSWORD=labpass

**5.4. Код застосунку**

# db.py  
import os  
import psycopg2  
from psycopg2.extras import RealDictCursor  
from dotenv import load\_dotenv  
load\_dotenv()  
def get\_conn():  
 return psycopg2.connect(  
 host=os.getenv('PGHOST'),  
 port=os.getenv('PGPORT'),  
 dbname=os.getenv('PGDATABASE'),  
 user=os.getenv('PGUSER'),  
 password=os.getenv('PGPASSWORD')  
 )  
  
# app.py  
import os  
from flask import Flask, request, jsonify  
from flask\_cors import CORS  
from db import get\_conn  
  
app = Flask(\_\_name\_\_)  
CORS(app)  
  
@app.get('/api/health')  
def health():  
 return jsonify(status='ok')  
  
@app.get('/api/books')  
def books\_all():  
 with get\_conn() as conn:  
 with conn.cursor(cursor\_factory=RealDictCursor) as cur:  
 cur.execute("SELECT \* FROM books ORDER BY id")  
 return jsonify(cur.fetchall())  
  
@app.get('/api/books/<int:id>')  
def books\_one(id):  
 with get\_conn() as conn:  
 with conn.cursor(cursor\_factory=RealDictCursor) as cur:  
 cur.execute("SELECT \* FROM books WHERE id=%s", (id,))  
 row = cur.fetchone()  
 return (jsonify(row), 200) if row else (jsonify(message='Not found'), 404)  
  
@app.post('/api/books')  
def books\_create():  
 data = request.get\_json()  
 with get\_conn() as conn:  
 with conn.cursor(cursor\_factory=RealDictCursor) as cur:  
 cur.execute("INSERT INTO books(title, author, year) VALUES (%s,%s,%s) RETURNING \*",  
 (data.get('title'), data.get('author'), data.get('year')))  
 row = cur.fetchone()  
 conn.commit()  
 return jsonify(row), 201  
  
@app.put('/api/books/<int:id>')  
def books\_update(id):  
 data = request.get\_json()  
 with get\_conn() as conn:  
 with conn.cursor(cursor\_factory=RealDictCursor) as cur:  
 cur.execute("UPDATE books SET title=%s, author=%s, year=%s WHERE id=%s RETURNING \*",  
 (data.get('title'), data.get('author'), data.get('year'), id))  
 row = cur.fetchone()  
 conn.commit()  
 return (jsonify(row), 200) if row else (jsonify(message='Not found'), 404)  
  
@app.delete('/api/books/<int:id>')  
def books\_delete(id):  
 with get\_conn() as conn:  
 with conn.cursor() as cur:  
 cur.execute("DELETE FROM books WHERE id=%s", (id,))  
 deleted = cur.rowcount  
 conn.commit()  
 return ('', 204) if deleted else (jsonify(message='Not found'), 404)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 port = int(os.getenv('PORT', 3001))  
 app.run(host='0.0.0.0', port=port, debug=True)

# 6. Альтернатива: MongoDB (Node.js + Mongoose, приклад моделі)

npm i mongoose  
// model  
const mongoose = require('mongoose');  
const BookSchema = new mongoose.Schema({  
 title: { type: String, required: true },  
 author: { type: String, required: true },  
 year: Number  
}, { timestamps: true });  
module.exports = mongoose.model('Book', BookSchema);

# 7. Тестування Postman

* Створити колекцію «Lab3 API» з базовою змінною {{baseUrl}} (напр. http://localhost:3000).
* Додати запити: GET {{baseUrl}}/api/health; GET/POST/PUT/DELETE для /api/books.
* Налаштувати приклади body (raw JSON) для створення/оновлення.
* Додати тести (Tests) на коди відповіді, структуру JSON, наявність полів.
* Експортувати колекцію та змінні оточення до звіту.

*Приклад тесту (Postman → Tests):*

pm.test("Status is 200", function () {  
 pm.response.to.have.status(200);  
});  
pm.test("Has JSON", function () {  
 pm.response.to.be.json;  
});

# 8. Завдання до виконання (мінімум)

1. Обрати стек: A) Node.js/Express + PostgreSQL або B) Python/Flask + PostgreSQL (дозволено MongoDB як варіант).
2. Створити REST API для ресурсу books (поля: id, title, author, year).
3. Реалізувати всі CRUD endpoints та обробку помилок (404/400/500).
4. Підготувати Postman-колекцію з тестами і змінними.
5. Забезпечити читання конфігурацій із .env, додати .env.example.
6. Оприлюднити код у публічному репозиторії (GitHub/GitLab) або здати архів.

# 9. Вимоги до звіту

* Титульна сторінка (предмет, тема, автор, група).
* Мета роботи, обраний стек, версії інструментів.
* Схема БД (SQL або ER-діаграма/опис колекції).
* Опис структури проєкту, скріншоти запуску сервера та Postman-тестів.
* Лістинги ключових фрагментів (роути/контролери/моделі/конфіг).
* Висновки (що вдалося, труднощі, подальші покращення).
* Посилання на репозиторій або архів з кодом, експорт Postman.

# 10. Питання до захисту

* Що таке REST та основні HTTP-методи?
* Як організовані шари у вашому застосунку (routes/controllers/models)?
* Різниця між PostgreSQL і MongoDB. Коли яку БД доречно обирати?
* Принципи підключення до БД та транзакції.
* Як конфігуруються змінні середовища? Навіщо .env.example?
* Що перевіряють ваші тести в Postman?

# Додаток A. Підказки Docker (за бажанням)

# PostgreSQL  
docker run --name pg-lab3 -e POSTGRES\_PASSWORD=labpass -e POSTGRES\_USER=labuser -e POSTGRES\_DB=lab3 -p 5432:5432 -d postgres:15  
  
# MongoDB  
docker run --name mongo-lab3 -p 27017:27017 -d mongo:6

**Лабораторна робота №4**

«Інтеграція інтелектуальних сервісів (AI API)»

**Вступ.** Четверта лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена інтеграції веб-застосунків із зовнішніми інтелектуальними сервісами (AI API). На цьому етапі студенти переходять від базової клієнт–серверної взаємодії до використання потужних можливостей сучасних мовних і аналітичних моделей, що забезпечують новий рівень функціональності додатків.

Основна увага приділяється підключенню до готових API провайдерів штучного інтелекту (OpenAI, Hugging Face, Google AI) через безпечний бекенд-проксі. Це дозволяє організувати роботу чат-ботів, систем аналізу текстів та інших інтелектуальних сервісів без розкриття ключів користувачам і клієнтській частині.

У процесі виконання роботи студенти навчаться:

* налаштовувати середовище та API-ключі для підключення зовнішніх моделей;
* реалізовувати ендпоінти для роботи з LLM (генерація тексту) та NLP-моделями (аналіз тональності, категоризація);
* застосовувати архітектуру «клієнт → сервер → провайдер AI», що забезпечує безпечність і контроль;
* організовувати обробку відповідей у форматі JSON та в режимі потокової передачі (streaming);
* створювати простий клієнтський інтерфейс для взаємодії з AI API;
* проводити тестування сервісів за допомогою Postman, створюючи тести для перевірки стабільності роботи.

Практична цінність цієї роботи полягає в оволодінні сучасними інструментами інтеграції штучного інтелекту у веб-додатки. Це навичка, яка особливо затребувана в сучасній індустрії, адже більшість цифрових сервісів сьогодні використовують AI для покращення взаємодії з користувачами, персоналізації контенту та підвищення ефективності бізнес-процесів.

# 1. Мета та результати навчання

Мета: навчитись підключати готові AI API (OpenAI, Hugging Face, Google AI) до веб-застосунку через безпечний бекенд-проксі, виконувати обробку відповідей і відображення результатів на клієнті.

Після виконання роботи студент повинен уміти:

* налаштувати API-ключі та середовище;
* реалізувати ендпоінти для чат-бота/аналізу тексту;
* виконати запити до моделей LLM і моделей NLP (тональність, категоризація);
* опрацьовувати відповіді (JSON/stream) та показувати їх на фронтенді;
* тестувати API у Postman і підготувати звіт.

# 2. Попередні вимоги

* Встановлені: Git, Node.js LTS або Python 3.10+, Postman.
* Акаунти/ключі: OPENAI\_API\_KEY, HF\_TOKEN (Hugging Face Inference API), GOOGLE\_API\_KEY (за бажанням).
* Базові вміння з HTTP/REST (GET/POST), робота з .env.

# 3. Короткі теоретичні відомості

* LLM (Large Language Models): моделі, що генерують/аналізують текст. Основні параметри запиту: model, messages/input, temperature, max\_tokens.
* Embeddings: числові вектори для пошуку за змістом, кластеризації й семантичної схожості.
* Inference endpoints: готові хости моделей (OpenAI/Google) або сторонні моделі через Hugging Face Inference API.
* Проксі через бекенд: ніколи не зберігати API-ключ на фронтенді; запити з браузера → ваш сервер → провайдер AI.
* Потокова передача (streaming): корисна для чат-інтерфейсу; фронтенд поступово виводить токени.
* Етика/безпека: обмеження вводу, фільтрація PII, логування, контроль вартості (rate limit), кешування відповідей.

# 4. Варіант A: Node.js / Express (проксі до AI)

**4.1. Створення проєкту і залежностей**

mkdir lr4-ai-express && cd lr4-ai-express  
npm init -y  
npm i express cors dotenv node-fetch  
# додатково для HF: npm i axios  
npm i -D nodemon

**4.2. Структура**

lr4-ai-express/  
 ├─ src/  
 │ ├─ index.js  
 │ ├─ providers/  
 │ │ ├─ openai.js  
 │ │ ├─ huggingface.js  
 │ │ └─ googleai.js  
 │ └─ routes.js  
 ├─ .env (.env.example)  
 └─ package.json

**4.3. .env (приклад)**

PORT=4000  
OPENAI\_API\_KEY=sk-...  
HF\_TOKEN=hf\_...  
GOOGLE\_API\_KEY=...

**4.4. Точка входу (src/index.js)**

const express = require('express'); const cors = require('cors'); require('dotenv').config();  
const routes = require('./routes'); const app = express();  
app.use(cors()); app.use(express.json());  
app.use('/api', routes);  
const port = process.env.PORT || 4000;  
app.listen(port, () => console.log(`API on http://localhost:${port}`));

**4.5. Провайдери (спрощено)**

// src/providers/openai.js  
const fetch = (...args) => import('node-fetch').then(({default: f}) => f(...args));  
exports.chat = async (messages, model = "gpt-4o-mini") => {  
 const r = await fetch("https://api.openai.com/v1/chat/completions", {  
 method:"POST",  
 headers:{  
 "Content-Type":"application/json",  
 "Authorization":`Bearer ${process.env.OPENAI\_API\_KEY}`  
 },  
 body: JSON.stringify({ model, messages, temperature: 0.2 })  
 });  
 const j = await r.json();  
 return j.choices?.[0]?.message?.content || JSON.stringify(j);  
};  
  
// src/providers/huggingface.js (тональність через Inference API)  
const axios = require('axios');  
exports.sentiment = async (text, model="finiteautomata/bertweet-base-sentiment-analysis") => {  
 const { data } = await axios.post(  
 `https://api-inference.huggingface.co/models/${model}`,  
 { inputs: text },  
 { headers: { Authorization: `Bearer ${process.env.HF\_TOKEN}` } }  
 );  
 return data;  
};

**4.6. Маршрути (src/routes.js)**

const router = require('express').Router();  
const openai = require('./providers/openai');  
const hf = require('./providers/huggingface');  
  
router.get('/health', (req,res)=>res.json({status:'ok'}));  
  
router.post('/chat', async (req,res) => {  
 const user = req.body?.message || "Hello";  
 const content = await openai.chat([  
 {role:"system", content:"You are a helpful assistant."},  
 {role:"user", content:user}  
 ]);  
 res.json({ reply: content });  
});  
  
router.post('/sentiment', async (req,res) => {  
 const text = req.body?.text || "";  
 const result = await hf.sentiment(text);  
 res.json({ result });  
});  
  
module.exports = router;

**4.7. Скрипти запуску (package.json – фрагмент)**

"scripts": { "dev": "nodemon src/index.js", "start": "node src/index.js" }

# 5. Варіант B: Python / Flask (проксі до AI)

**5.1. Віртуальне середовище і залежності**

mkdir lr4-ai-flask && cd lr4-ai-flask  
python3 -m venv venv  
source venv/bin/activate # Windows: venv\Scripts\activate  
pip install flask flask-cors requests python-dotenv

**5.2. Структура**

lr4-ai-flask/  
 ├─ app.py  
 ├─ providers.py  
 ├─ .env (.env.example)  
 └─ requirements.txt

**5.3. .env**

PORT=4001  
OPENAI\_API\_KEY=sk-...  
HF\_TOKEN=hf\_...

**5.4. Код провайдерів і застосунку**

# providers.py  
import os, requests  
from dotenv import load\_dotenv  
load\_dotenv()  
  
def openai\_chat(messages, model="gpt-4o-mini"):  
 url = "https://api.openai.com/v1/chat/completions"  
 r = requests.post(url, headers={  
 "Authorization": f"Bearer {os.getenv('OPENAI\_API\_KEY')}",  
 "Content-Type": "application/json"  
 }, json={"model": model, "messages": messages, "temperature": 0.2})  
 j = r.json()  
 return (j.get("choices",[{}])[0].get("message",{}).get("content")) or j  
  
def hf\_sentiment(text, model="finiteautomata/bertweet-base-sentiment-analysis"):  
 url = f"https://api-inference.huggingface.co/models/{model}"  
 r = requests.post(url, headers={"Authorization": f"Bearer {os.getenv('HF\_TOKEN')}"}, json={"inputs": text})  
 return r.json()  
  
# app.py  
import os  
from flask import Flask, request, jsonify  
from flask\_cors import CORS  
from providers import openai\_chat, hf\_sentiment  
app = Flask(\_\_name\_\_); CORS(app)  
  
@app.get("/api/health")  
def health(): return jsonify(status="ok")  
  
@app.post("/api/chat")  
def chat():  
 user = request.json.get("message","Hello")  
 reply = openai\_chat([{"role":"system","content":"You are a helpful assistant."},  
 {"role":"user","content":user}])  
 return jsonify(reply=reply)  
  
@app.post("/api/sentiment")  
def sentiment():  
 text = request.json.get("text","")  
 result = hf\_sentiment(text)  
 return jsonify(result=result)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(host="0.0.0.0", port=int(os.getenv("PORT",4001)), debug=True)

# 6. Клієнтська частина (мінімальний приклад)

*index.html (під'єднання до бекенду, показ відповіді):*

<!doctype html>  
<html>  
<head><meta charset="utf-8"><title>LR4 AI Demo</title></head>  
<body>  
 <h3>Чат з AI</h3>  
 <input id="msg" placeholder="Введіть повідомлення" />  
 <button onclick="send()">Надіслати</button>  
 <pre id="out"></pre>  
<script>  
async function send(){  
 const r = await fetch('http://localhost:4000/api/chat',{  
 method:'POST', headers:{'Content-Type':'application/json'},  
 body: JSON.stringify({message: document.getElementById('msg').value})  
 });  
 const j = await r.json();  
 document.getElementById('out').textContent = j.reply;  
}  
</script>  
</body>  
</html>

# 7. Тестування у Postman

* Створити колекцію «LR4 AI API» і зміну {{baseUrl}} (напр. http://localhost:4000/api).
* Додати запити: GET {{baseUrl}}/health; POST {{baseUrl}}/chat { message }; POST {{baseUrl}}/sentiment { text }.
* Додати тести на коди 200/400, наявність поля reply/result.

*Приклад тесту:*

pm.test("Status 200", ()=>pm.response.to.have.status(200));  
const json = pm.response.json();  
pm.expect(json).to.have.any.keys('reply','result');

# 8. Завдання до виконання

1. Обрати стек (Express або Flask).
2. Зробити два ендпоінти: /chat (LLM) і /sentiment (тональність).
3. Додати обмеження: max довжина введення, таймаути, обробка помилок.
4. Зробити просту клієнтську сторінку з формою та полем виводу.
5. Зібрати Postman-колекцію з тестами та експортувати її у звіт.

# 9. Вимоги до звіту

* Титульна, мета, інструменти, ключі (без розкриття значень).
* Схеми запитів (клієнт → бекенд → провайдер), діаграма послідовностей (за бажанням).
* Фрагменти коду бекенда/клієнта, опис моделей/ендпоінтів.
* Скріншоти Postman і приклад роботи інтерфейсу.
* Висновки: порівняння провайдерів, затримка, вартість, якість.

# 10. Питання до захисту

* Чому небезпечно зберігати API-ключ на фронтенді?
* Що таке temperature, max\_tokens і як вони впливають на результат?
* Що таке embeddings і де їх доречно використовувати?
* Як реалізувати стрімінг відповіді та чим він відрізняється від звичайного запиту?
* Які обмеження і політики безпеки слід ураховувати при роботі з AI API?

# Додаток A. Підказки Docker (за бажанням)

# Node.js API  
docker build -t lr4-ai-express .  
docker run -p 4000:4000 --env-file .env lr4-ai-express  
  
# Flask API  
docker build -t lr4-ai-flask .  
docker run -p 4001:4001 --env-file .env lr4-ai-flask

**Лабораторна робота №5**

«Методи обробки та управління даними користувача»

**Вступ.** П’ята лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена практиці роботи з даними користувача у веб-застосунках. У попередніх роботах основна увага приділялася клієнтській та серверній частині застосунків, однак справжня взаємодія з користувачами починається тоді, коли потрібно збирати, зберігати й обробляти персональні дані.

Основна ідея цієї роботи полягає у формуванні вмінь правильно організовувати процес збору та валідації даних, забезпечуючи при цьому належний рівень безпеки. Особливий акцент робиться на роботі з формами, локальним сховищем браузера (localStorage, sessionStorage) та сесіями на сервері.

У процесі виконання роботи студенти навчаться:

* створювати HTML-форми для збору різних типів даних (текст, email, число, коментар);
* передавати дані на сервер за допомогою HTTP-запитів;
* реалізовувати валідацію на клієнтському та серверному рівні;
* працювати з локальним сховищем браузера та серверними сесіями;
* впроваджувати базові механізми захисту від XSS та CSRF-атак.

Практична цінність цієї роботи полягає у формуванні критично важливих навичок для будь-якого веб-розробника, адже саме обробка даних користувачів є серцевиною більшості сучасних веб-застосунків. Вміння правильно будувати цей процес забезпечує надійність і безпечність роботи сервісів, а також довіру користувачів до системи.

# 1. Мета та результати навчання

Мета: навчитися збирати та обробляти дані користувача з веб-форм, використовувати локальне сховище та сесії, виконувати валідацію даних і реалізовувати базові заходи безпеки.

Після виконання роботи студент повинен уміти:

* збирати дані з HTML-форм і передавати їх на бекенд;
* обробляти дані на сервері та клієнті;
* використовувати локальне сховище (localStorage, sessionStorage);
* реалізовувати сесії на бекенді;
* виконувати валідацію та базовий захист даних.

# 2. Попередні вимоги

* Базові знання HTML, CSS, JavaScript.
* Вміння працювати з Node.js або Python (Flask).
* Встановлені інструменти: Postman, Git, PostgreSQL або MongoDB (за потреби).

# 3. Короткі теоретичні відомості

* HTML-форма: збір даних користувача за допомогою елементів input, textarea, select.
* HTTP-запити: передача даних на бекенд за допомогою POST-запитів.
* Валідація даних: на клієнті (HTML5-атрибути) та на сервері (повторна перевірка).
* Локальне сховище: збереження даних у браузері (localStorage — довгостроково, sessionStorage — на час сесії).
* Сесії на бекенді: збереження інформації про користувача на сервері.
* Безпека: захист від XSS, CSRF, валідація вхідних даних, маскування конфіденційних даних.

# 4. Варіант A: Node.js / Express

**4.1. Ініціалізація проєкту**

mkdir lr5-user-data && cd lr5-user-data  
npm init -y  
npm i express body-parser express-session cors dotenv  
npm i -D nodemon

**4.2. Структура**

lr5-user-data/  
 ├─ src/  
 │ ├─ index.js  
 │ ├─ routes.js  
 │ ├─ sessionStore.js  
 │ └─ validators.js  
 ├─ public/  
 │ └─ form.html  
 ├─ .env  
 └─ package.json

**4.3. HTML-форма (public/form.html)**

<!DOCTYPE html>  
<html>  
<head><meta charset="UTF-8"><title>Форма</title></head>  
<body>  
 <form id="dataForm">  
 <input type="text" name="name" placeholder="Ім'я" required><br>  
 <input type="email" name="email" placeholder="Email" required><br>  
 <button type="submit">Відправити</button>  
 </form>  
<script>  
document.getElementById('dataForm').addEventListener('submit', async e => {  
 e.preventDefault();  
 const data = Object.fromEntries(new FormData(e.target).entries());  
 localStorage.setItem('user', JSON.stringify(data));  
 const res = await fetch('/api/submit', {  
 method: 'POST', headers: {'Content-Type': 'application/json'}, body: JSON.stringify(data)  
 });  
 alert(await res.text());  
});  
</script>  
</body>  
</html>

**4.4. Сервер (src/index.js)**

const express = require('express');  
const session = require('express-session');  
const cors = require('cors');  
const bodyParser = require('body-parser');  
require('dotenv').config();  
  
const app = express();  
app.use(cors());  
app.use(bodyParser.json());  
app.use(session({ secret: process.env.SESSION\_SECRET, resave: false, saveUninitialized: true }));  
app.use(express.static('public'));  
  
app.post('/api/submit', (req, res) => {  
 const { name, email } = req.body;  
 if (!name || !email) return res.status(400).send('Дані не валідні');  
 req.session.user = { name, email };  
 res.send('Дані отримано та збережено у сесії');  
});  
  
app.get('/api/user', (req, res) => {  
 res.json(req.session.user || {});  
});  
  
app.listen(process.env.PORT || 3000, () => console.log('Server running'));

# 5. Варіант B: Python / Flask

**5.1. Ініціалізація середовища**

mkdir lr5-user-data-flask && cd lr5-user-data-flask  
python3 -m venv venv  
source venv/bin/activate # Windows: venv\Scripts\activate  
pip install flask flask-session python-dotenv

**5.2. Файлова структура**

lr5-user-data-flask/  
 ├─ app.py  
 ├─ templates/  
 │ └─ form.html  
 ├─ .env  
 └─ requirements.txt

**5.3. HTML-форма (templates/form.html)**

<!DOCTYPE html>  
<html>  
<head><meta charset="UTF-8"><title>Форма</title></head>  
<body>  
 <form method="POST" action="/submit">  
 <input type="text" name="name" placeholder="Ім'я" required><br>  
 <input type="email" name="email" placeholder="Email" required><br>  
 <button type="submit">Відправити</button>  
 </form>  
</body>  
</html>

**5.4. Flask-додаток (app.py)**

import os  
from flask import Flask, request, session, render\_template  
from flask\_session import Session  
from dotenv import load\_dotenv  
  
load\_dotenv()  
app = Flask(\_\_name\_\_)  
app.secret\_key = os.getenv('SESSION\_SECRET', 'default\_secret')  
app.config['SESSION\_TYPE'] = 'filesystem'  
Session(app)  
  
@app.route('/')  
def form():  
 return render\_template('form.html')  
  
@app.route('/submit', methods=['POST'])  
def submit():  
 name = request.form.get('name')  
 email = request.form.get('email')  
 if not name or not email:  
 return "Дані не валідні", 400  
 session['user'] = {'name': name, 'email': email}  
 return "Дані отримано та збережено у сесії"  
  
@app.route('/user')  
def user():  
 return session.get('user', {})  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True, port=int(os.getenv('PORT', 3000)))

# 6. Базові принципи безпеки

* Валідація даних на сервері навіть якщо вона є на клієнті.
* Не довіряти даним із localStorage/sessionStorage — перевіряти на бекенді.
* Захист від XSS — екранування HTML-спецсимволів.
* CSRF-захист — використання токенів при POST-запитах.
* Надійне зберігання сесій і обмеження терміну дії.

# 7. Завдання до виконання

1. Створити форму з кількома полями: ім’я, email, вік, коментар.
2. Зберігати дані у localStorage та на сервері.
3. Реалізувати валідацію: мінімальна довжина, правильний email.
4. Додати відображення введених даних у окремому вікні.
5. Підготувати Postman-колекцію для тестування API.

# 8. Вимоги до звіту

* Титульна сторінка.
* Мета роботи, інструменти.
* Код фронтенду та бекенду.
* Скріншоти роботи застосунку.
* Висновки про отримані навички.

# 9. Питання до захисту

* Яка різниця між localStorage та sessionStorage?
* Чому валідація на клієнті недостатня?
* Як реалізувати захист від XSS та CSRF?
* Як працюють серверні сесії?
* Чому важливо не зберігати паролі у відкритому вигляді?

**Лабораторна робота №6**

«Застосування базових моделей машинного навчання»

**Вступ.** Шоста лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена практичному застосуванню методів машинного навчання у веб-застосунках. Якщо попередні роботи були орієнтовані на інтеграцію сторонніх API та роботу з даними користувачів, то на цьому етапі студенти вперше створюють та навчають власні моделі штучного інтелекту.

Основна ідея роботи полягає у формуванні навичок побудови та інтеграції простих моделей класифікації в робоче середовище веб-додатка. Студенти ознайомляться з базовими бібліотеками для машинного навчання — **scikit-learn** (Python) та **TensorFlow.js** (JavaScript) — і навчаться виконувати повний цикл обробки даних: від підготовки до тестування та інтеграції.

У процесі виконання роботи студенти навчаться:

* налаштовувати середовище для використання ML-бібліотек;
* створювати та навчати базові моделі класифікації;
* проводити оцінку якості моделі та інтерпретувати результати;
* інтегрувати модель у веб-додаток (Node.js / Flask);
* готувати API для тестування моделей за допомогою Postman.

Практична цінність цієї роботи полягає у здобутті першого практичного досвіду у сфері машинного навчання, що дозволяє краще зрозуміти принципи роботи інтелектуальних систем. Цей досвід стане важливою базою для подальших лабораторних робіт, де передбачено використання більш складних моделей, інтеграцію з хмарними сервісами та оптимізацію продуктивності.

# 1. Мета та результати навчання

Мета: ознайомитися з бібліотеками TensorFlow.js та scikit-learn, навчитися будувати прості моделі машинного навчання, тренувати їх та інтегрувати у веб-додаток.

Після виконання роботи студент повинен уміти:

* налаштовувати середовище для роботи з ML-бібліотеками;
* створювати та навчати прості моделі;
* виконувати оцінку якості моделей;
* інтегрувати модель у веб-додаток (Node.js / Flask).

# 2. Попередні вимоги

* Знання Python або JavaScript.
* Встановлений Node.js та/або Python 3.10+.
* Встановлені бібліотеки: scikit-learn, pandas, numpy, TensorFlow.js (за потреби).

# 3. Короткі теоретичні відомості

* Машинне навчання (ML): створення моделей, які вчаться знаходити закономірності у даних.
* TensorFlow.js: використання моделей ML у браузері або Node.js.
* scikit-learn: бібліотека Python для побудови, навчання та тестування ML-моделей.
* Класифікація: задача передбачення мітки класу за вхідними даними.
* Пайплайн: підготовка даних → навчання → тестування → інтеграція.

# 4. Варіант A: scikit-learn (Python)

**4.1. Встановлення залежностей**

pip install scikit-learn pandas numpy flask

**4.2. Приклад: класифікація тексту**

import pandas as pd  
from sklearn.feature\_extraction.text import CountVectorizer  
from sklearn.naive\_bayes import MultinomialNB  
from sklearn.pipeline import make\_pipeline  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
  
# Дані  
data = pd.DataFrame({  
 "text": ["I love Python", "I hate bugs", "JavaScript is awesome", "I dislike errors"],  
 "label": ["positive", "negative", "positive", "negative"]  
})  
  
# Розбиття на навчальні і тестові  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(data["text"], data["label"], test\_size=0.25)  
  
# Модель  
model = make\_pipeline(CountVectorizer(), MultinomialNB())  
model.fit(X\_train, y\_train)  
  
# Передбачення  
pred = model.predict(X\_test)  
print("Accuracy:", accuracy\_score(y\_test, pred))  
  
# Збереження моделі  
import joblib  
joblib.dump(model, "text\_classifier.pkl")

**4.3. Інтеграція моделі у Flask**

from flask import Flask, request, jsonify  
import joblib  
  
app = Flask(\_\_name\_\_)  
model = joblib.load("text\_classifier.pkl")  
  
@app.route("/predict", methods=["POST"])  
def predict():  
 data = request.get\_json()  
 prediction = model.predict([data["text"]])[0]  
 return jsonify({"prediction": prediction})  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(port=5000, debug=True)

# 5. Варіант B: TensorFlow.js

**5.1. Встановлення середовища**

npm init -y  
npm install @tensorflow/tfjs express body-parser cors

**5.2. Приклад створення простої моделі**

const tf = require('@tensorflow/tfjs');  
  
// Створення моделі  
const model = tf.sequential();  
model.add(tf.layers.dense({ units: 8, activation: 'relu', inputShape: [2] }));  
model.add(tf.layers.dense({ units: 1, activation: 'sigmoid' }));  
  
model.compile({ optimizer: 'adam', loss: 'binaryCrossentropy', metrics: ['accuracy'] });  
  
// Навчання на штучних даних  
const xs = tf.tensor2d([[0,0],[0,1],[1,0],[1,1]]);  
const ys = tf.tensor2d([[0],[1],[1],[0]]);  
  
(async () => {  
 await model.fit(xs, ys, { epochs: 200 });  
 model.save('file://./model');  
 console.log('Модель навчена та збережена!');  
})();

**5.3. Інтеграція моделі у Node.js**

const express = require('express');  
const tf = require('@tensorflow/tfjs-node');  
const bodyParser = require('body-parser');  
  
const app = express();  
app.use(bodyParser.json());  
  
let model;  
(async () => {  
 model = await tf.loadLayersModel('file://./model/model.json');  
 console.log('Модель завантажена');  
})();  
  
app.post('/predict', async (req, res) => {  
 const input = tf.tensor2d([req.body.data]);  
 const prediction = model.predict(input);  
 res.json({ prediction: Array.from(prediction.dataSync()) });  
});  
  
app.listen(4000, () => console.log('Server running'));

# 6. Завдання до виконання

1. Оберіть стек: Python (scikit-learn) або TensorFlow.js.
2. Створіть просту модель для класифікації текстів або числових даних.
3. Навчіть модель на власному наборі даних.
4. Інтегруйте модель у веб-додаток.
5. Підготуйте Postman-колекцію для тестування API.

# 7. Вимоги до звіту

* Титульна сторінка.
* Мета, опис завдання та інструменти.
* Код моделі, результати навчання.
* Скріншоти тестування та інтеграції у веб-додаток.
* Висновки щодо точності та ефективності моделі.

# 9. Питання до захисту

* Що таке модель машинного навчання?
* У чому різниця між TensorFlow.js та scikit-learn?
* Як виконати навчання та перевірку якості моделі?
* Що таке перенавчання і як його уникнути?
* Як інтегрувати модель у веб-додаток?

**Лабораторна робота №7**

«Використання хмарних сервісів та деплоймент застосунків»

**Вступ.** Сьома лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена вивченню процесів розгортання веб-застосунків у хмарних середовищах. На цьому етапі студенти переходять від локальної розробки до публікації проєктів у продакшн-середовищі, де до застосунків можуть отримати доступ реальні користувачі.

Основна увага приділяється підготовці застосунку до деплою, налаштуванню змінних середовища та інтеграції базових механізмів моніторингу працездатності. Також студенти знайомляться з різними платформами для деплою (Heroku, Render, AWS), що дозволяє порівняти їхні можливості, переваги та обмеження.

У процесі виконання роботи студенти навчаться:

* готувати застосунки до розгортання за допомогою Dockerfile або buildpack-стратегій;
* налаштовувати змінні середовища та секрети для збереження конфіденційних даних;
* здійснювати деплой на хмарні платформи Heroku, Render або AWS;
* створювати health-check ендпоінти для перевірки працездатності сервісів;
* підключати журнали та налаштовувати uptime-моніторинг;
* використовувати CI/CD-підходи для автоматизації деплою з GitHub Actions.

Практична цінність цієї роботи полягає у формуванні ключових навичок продакшн-орієнтованої розробки. Вміння правильно налаштовувати середовище, публікувати застосунки та контролювати їхню працездатність є необхідними для будь-якого веб-розробника. Цей досвід стане фундаментом для подальших робіт, де розглядатиметься інтеграція більш складних сервісів і оптимізація роботи у хмарних середовищах.

# 1. Мета та результати навчання

Мета: навчитися готувати застосунок до продакшн-деплойменту, розгортати його на вибраній хмарній платформі (Heroku / AWS / Render), коректно налаштовувати змінні середовища та додати базовий моніторинг працездатності/продуктивності.

Після виконання роботи студент повинен уміти:

* контейнеризувати застосунок (Docker) або підготувати build-процес;
* налаштовувати секрети/ENV (API ключі, URI БД, PORT);
* здійснити деплой на одній з платформ (Heroku / AWS / Render);
* налаштувати логування та моніторинг (health-check, uptime, базові метрики).

# 2. Попередні вимоги

* Виконана ЛР3 (бекенд з REST API) — буде деплоїтись.
* Облікові записи на хмарних платформах (мінімум одна).
* GitHub-репозиторій з кодом. Бажано Docker встановлений локально.

# 3. Короткі теоретичні відомості

* Деплой: перенесення застосунку з локального середовища у хмару з можливістю доступу клієнтів.
* ENV: конфігурація через змінні середовища (.env локально, Secrets у хмарі).
* Стратегії: Docker-контейнер, buildpack (Heroku), нативний рантайм (Render).
* Моніторинг: Uptime/health-check, логування, базові APM/метрики (CPU/RAM/Latency).

# 4. Підготовка застосунку до деплою

**4.1. Dockerfile (Node.js приклад)**

FROM node:20-alpine  
WORKDIR /app  
COPY package\*.json ./  
RUN npm ci --only=production  
COPY . .  
ENV NODE\_ENV=production  
EXPOSE 3000  
CMD ["node", "src/index.js"]

**4.2. Dockerfile (Python/Flask приклад)**

FROM python:3.12-slim  
WORKDIR /app  
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt  
COPY . .  
ENV PYTHONUNBUFFERED=1  
EXPOSE 3000  
CMD ["python", "app.py"]

**.env.example**

PORT=3000  
DATABASE\_URL=postgresql://user:pass@host:5432/dbname  
OPENAI\_API\_KEY=sk-\*\*\*  
ALLOWED\_ORIGIN=https://your-frontend.example

**Health-check endpoint**

// Express  
app.get('/health', (req,res)=>res.json({status:'ok', time: Date.now()}));  
# Flask  
@app.get('/health')  
def health(): return {'status':'ok'}

# 5. Варіант A: Деплой на Heroku (buildpacks)

1. Встановіть Heroku CLI та увійдіть: heroku login.
2. Створіть застосунок: heroku create your-app-name.
3. Налаштуйте ENV: heroku config:set PORT=3000 DATABASE\_URL=...
4. Виконайте деплой: git push heroku main (або master).
5. Перевірте доступність (heroku open) і логи: heroku logs --tail.

*Procfile (Node.js):*

web: node src/index.js

# 6. Варіант B: Деплой на Render

* Render → New → Web Service → під’єднайте GitHub-репозиторій.
* Задайте Start command (наприклад, node src/index.js або gunicorn app:app).
* Налаштуйте Environment Variables (PORT, DATABASE\_URL, ін.).
* Health Check Path: /health; Auto Deploy: On.

# 7. Варіант C: AWS (Elastic Beanstalk Docker або ECS Fargate)

**7.1. Elastic Beanstalk (один контейнер)**

1. aws configure; eb init (Docker);
2. eb create для середовища; eb setenv PORT=3000 DATABASE\_URL=...;
3. eb deploy; eb open; eb logs.

**7.2. ECS Fargate (коротко)**

* Публікуйте Docker-образ у ECR;
* Task Definition з env, Service + Target Group з health-check /health;
* Application Load Balancer, авто-скейл за CPU/Latency.

# 8. CI/CD з GitHub Actions

**Heroku:**

name: Deploy to Heroku  
on: [push]  
jobs:  
 deploy:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
 - uses: akhileshns/heroku-deploy@v4.2.1  
 with:  
 heroku\_api\_key: ${{ secrets.HEROKU\_API\_KEY }}  
 heroku\_app\_name: your-app-name  
 heroku\_email: your@email.com

**Build & Push Docker до GHCR:**

name: Build & Push Docker Image  
on: [push]  
jobs:  
 docker:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
 - uses: docker/login-action@v3  
 with:  
 registry: ghcr.io  
 username: ${{ github.actor }}  
 password: ${{ secrets.GITHUB\_TOKEN }}  
 - uses: docker/build-push-action@v6  
 with:  
 push: true  
 tags: ghcr.io/${{ github.repository }}:latest

# 9. Моніторинг продуктивності та працездатності

* Health-check /health (200 OK).
* Uptime-моніторинг: UptimeRobot / BetterStack / Uptime Kuma.
* Логи: Heroku Logs, Render Logs, AWS CloudWatch Logs.
* Метрики: latency, помилки 5xx, CPU/RAM; опційно Prometheus/Grafana.

*Ендпойнт простих метрик (Prometheus):*

// Express  
app.get('/metrics', (req,res)=>res.type('text/plain').send('app\_up 1'))  
# Flask  
@app.get('/metrics')  
def metrics():   
 from flask import Response  
 return Response('app\_up 1', mimetype='text/plain')

# 10. Завдання до виконання

1. Підготуйте застосунок (health-check, .env.example, Dockerfile/Procfile).
2. Зробіть деплой на одну з платформ (Heroku/Render/AWS).
3. Налаштуйте ENV і перевірте /health; надішліть URL викладачу.
4. Підключіть журнали та додайте uptime-моніторинг (скрін).
5. Опціонально: налаштуйте GitHub Actions для авто-деплою.

# 11. Вимоги до звіту

* Титульна; мета; обрана платформа; посилання на сервіс/репозиторій.
* Скріншоти: ENV, логи, результат health-check та uptime.
* Фрагменти Dockerfile/Procfile та CI/CD конфігів.
* Висновки: проблеми/рішення, час розгортання, порівняння платформ (за наявності).

# 12. Питання до захисту

* Порівняйте buildpacks vs Docker vs нативний рантайм.
* Як безпечно зберігати секрети та змінні середовища?
* Що таке health-check і як його використовують балансувальники?
* Які метрики варто моніторити в продакшні?
* Як організувати безперервний деплой з GitHub Actions?

**Лабораторна робота №8**

«Розробка чат-бота або рекомендаційної системи»

**Вступ.** Восьма лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена створенню інтелектуальних сервісів, здатних взаємодіяти з користувачем у режимі діалогу або забезпечувати автоматичні рекомендації. Це один із ключових етапів курсу, оскільки студенти переходять від класичних веб-застосунків до застосунків, що реалізують елементи штучного інтелекту та персоналізації.

Основна увага приділяється двом напрямам:

* розробці **чат-ботів**, які імітують спілкування з користувачем природною мовою та здатні відповідати на запити;
* створенню **рекомендаційних систем**, що підбирають контент чи рішення на основі даних користувача.

У процесі виконання роботи студенти навчаться:

* налаштовувати інтеграцію з сучасними NLP-сервісами (OpenAI, Hugging Face, Google AI);
* реалізовувати чат-ботів на основі моделей обробки природної мови;
* створювати прості рекомендаційні системи з використанням алгоритмів подібності (TF-IDF, cosine similarity);
* будувати REST API для взаємодії клієнта з сервісом;
* тестувати чат-ботів і рекомендаційні системи за допомогою Postman та користувацьких сценаріїв.

Практична цінність цієї роботи полягає в отриманні досвіду створення інтелектуальних модулів, що здатні підвищувати зручність і якість взаємодії користувачів із системою. Такі навички є надзвичайно затребуваними в сучасній індустрії, адже чат-боти та системи рекомендацій використовуються у сфері електронної комерції, освіти, сервісного обслуговування та соціальних платформ.

# 1. Мета та результати навчання

Мета: ознайомитися з архітектурою інтелектуальних сервісів, інтеграцією NLP-моделей для обробки природної мови та реалізувати чат-бот або просту рекомендаційну систему.

Після виконання роботи студент повинен уміти:

* налаштовувати інтеграцію з NLP-сервісами (OpenAI, Hugging Face, Google AI);
* реалізовувати чат-бота або рекомендаційну систему;
* створювати API для взаємодії клієнта з сервісом;
* тестувати чат-бота або рекомендаційну систему за користувацькими сценаріями.

# 2. Попередні вимоги

* Виконана ЛР4 (інтеграція AI API).
* Базові знання Flask/Express для створення API.
* API-ключі для NLP-сервісів (OpenAI, Hugging Face).

# 3. Короткі теоретичні відомості

* Чат-бот: інтелектуальний сервіс для діалогу з користувачем у режимі реального часу.
* Рекомендаційна система: автоматичний підбір контенту чи рішень на основі даних користувача.
* NLP (Natural Language Processing): обробка природної мови для розуміння та генерації тексту.
* OpenAI API, Hugging Face та Google AI — приклади сучасних NLP-сервісів.

# 4. Варіант A: Чат-бот на Flask із інтеграцією OpenAI API

**4.1. Встановлення залежностей**

pip install flask openai python-dotenv

**4.2. .env приклад**

OPENAI\_API\_KEY=sk-\*\*\*  
PORT=5000

**4.3. Реалізація Flask-додатку**

import os  
from flask import Flask, request, jsonify  
from dotenv import load\_dotenv  
import openai  
  
load\_dotenv()  
app = Flask(\_\_name\_\_)  
openai.api\_key = os.getenv("OPENAI\_API\_KEY")  
  
@app.route("/chat", methods=["POST"])  
def chat():  
 data = request.get\_json()  
 user\_message = data.get("message")  
 response = openai.Completion.create(  
 engine="text-davinci-003",  
 prompt=user\_message,  
 max\_tokens=100  
 )  
 return jsonify({"reply": response.choices[0].text.strip()})  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(port=int(os.getenv("PORT", 5000)))

**4.4. Тестування API через Postman**

* POST → http://localhost:5000/chat
* Body → raw → JSON: { "message": "Привіт!" }
* Очікувано отримати відповідь у полі reply.

# 5. Варіант B: Рекомендаційна система (Python, scikit-learn)

**5.1. Встановлення залежностей**

pip install scikit-learn pandas flask

**5.2. Приклад побудови моделі**

import pandas as pd  
from sklearn.feature\_extraction.text import TfidfVectorizer  
from sklearn.metrics.pairwise import cosine\_similarity  
  
# Набір даних  
data = pd.DataFrame({  
 "title": ["Python Basics", "ML Intro", "Deep Learning", "Data Science"],  
 "tags": ["python, basics", "machine learning, intro", "deep learning, neural", "data science, statistics"]  
})  
  
# Векторизація тегів  
vectorizer = TfidfVectorizer()  
tfidf\_matrix = vectorizer.fit\_transform(data["tags"])  
  
# Функція рекомендацій  
def recommend(content\_title):  
 idx = data[data["title"] == content\_title].index[0]  
 cosine\_sim = cosine\_similarity(tfidf\_matrix[idx], tfidf\_matrix).flatten()  
 similar\_indices = cosine\_sim.argsort()[-3:][::-1]  
 return data["title"].iloc[similar\_indices].tolist()

# 6. Тестування чат-бота або рекомендаційної системи

* Тестування API через Postman або cURL.
* Реальні користувацькі сценарії: наприклад, бот відповідає на питання, або система пропонує подібні статті.
* Валідація результатів: коректність, релевантність, якість відповіді.

# 7. Завдання до виконання

1. Реалізуйте чат-бота або рекомендаційну систему (на вибір).
2. Налаштуйте інтеграцію з NLP API (OpenAI/Hugging Face).
3. Створіть REST API для взаємодії з сервісом.
4. Протестуйте роботу сервісу у Postman.
5. Зробіть звіт із результатами.

# 8. Вимоги до звіту

* Титульна сторінка.
* Мета роботи та опис реалізованого рішення.
* Код реалізації чат-бота або рекомендаційної системи.
* Скріншоти роботи API.
* Висновки щодо точності й функціональності.

# 9. Питання до захисту

* Що таке NLP і які його основні задачі?
* Чим відрізняється чат-бот від рекомендаційної системи?
* Які API для NLP ви знаєте і чим вони відрізняються?
* Як тестувати чат-бота або рекомендаційну систему?
* Які показники якості важливі для чат-бота?

**Лабораторна робота №9**

«Аналітика користувацької активності та візуалізація даних»

**Вступ.** Дев’ята лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена методам збору, аналізу та візуалізації даних користувацької активності. Якщо попередні роботи були зосереджені на створенні клієнтських і серверних частин застосунків та інтеграції з інтелектуальними сервісами, то ця робота акцентує увагу на побудові інструментів аналітики та дашбордів.

Основна ідея роботи полягає у формуванні вмінь збирати дані про поведінку користувачів, виявляти закономірності у їхніх діях і представляти результати у зручній та наочній формі. Студенти ознайомляться з популярними бібліотеками для візуалізації — **Chart.js** і **D3.js**, а також навчаться застосовувати методи аналізу трендів за допомогою **Pandas** у Python.

У процесі виконання роботи студенти навчаться:

* налаштовувати механізми збору логів користувацької активності у форматі JSON;
* створювати динамічні графіки та діаграми з використанням бібліотек Chart.js та D3.js;
* виконувати аналіз трендів у даних за допомогою Python/Pandas;
* будувати інтерактивні дашборди для відображення статистики;
* інтегрувати аналітичні сервіси у веб-застосунки.

Практична цінність цієї роботи полягає у розвитку навичок роботи з даними та побудови систем візуальної аналітики, що є невід’ємною частиною сучасних веб-сервісів. Ці знання стануть корисними як для створення власних проектів, так і для подальшої професійної діяльності у сфері веб-розробки, де аналіз користувацької поведінки є ключовим фактором для вдосконалення продуктів і сервісів.

# 1. Мета та результати навчання

Мета: навчитися збирати логи користувацької активності, аналізувати дані, будувати візуалізації за допомогою Chart.js або D3.js та створювати інтерактивні дашборди.

Після виконання роботи студент повинен уміти:

* збирати та зберігати логи користувацької активності;
* аналізувати запити та поведінку користувачів;
* будувати динамічні графіки, діаграми та тренди;
* створювати інтерактивні дашборди для відображення аналітики.

# 2. Попередні вимоги

* Базові знання JavaScript, Python та роботи з REST API.
* Виконані ЛР3–ЛР5 (створення бекенду та роботи з формами).
* Встановлені бібліотеки Chart.js або D3.js.

# 3. Короткі теоретичні відомості

* Аналітика користувачів: збір даних про запити, кліки, сторінки та інші події.
* Візуалізація даних: використання графіків, діаграм і дашбордів для представлення інформації.
* Chart.js: проста бібліотека для побудови інтерактивних графіків.
* D3.js: потужний інструмент для складних інтерактивних візуалізацій.
* Аналіз трендів: виявлення закономірностей і поведінкових патернів.

# 4. Приклад збору логів користувацької активності (Node.js + Express)

const express = require('express');  
const fs = require('fs');  
const app = express();  
  
app.use(express.json());  
  
app.post('/log', (req, res) => {  
 const logEntry = {  
 user: req.body.user,  
 action: req.body.action,  
 timestamp: new Date()  
 };  
 fs.appendFileSync('logs.json', JSON.stringify(logEntry) + '\n');  
 res.sendStatus(200);  
});  
  
app.listen(3000, () => console.log('Logger running on port 3000'));

# 5. Приклад побудови дашборду з Chart.js

**5.1. HTML-додаток**

<!DOCTYPE html>  
<html>  
<head>  
 <title>Аналітика користувачів</title>  
 <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>  
</head>  
<body>  
<canvas id="activityChart" width="400" height="200"></canvas>  
<script>  
const ctx = document.getElementById('activityChart').getContext('2d');  
const chart = new Chart(ctx, {  
 type: 'bar',  
 data: {  
 labels: ['Логін', 'Пошук', 'Перегляд сторінок', 'Вихід'],  
 datasets: [{  
 label: 'Кількість дій',  
 data: [12, 19, 3, 5],  
 backgroundColor: 'rgba(54, 162, 235, 0.5)',  
 borderColor: 'rgba(54, 162, 235, 1)',  
 borderWidth: 1  
 }]  
 },  
 options: {  
 responsive: true,  
 scales: {  
 y: { beginAtZero: true }  
 }  
 }  
});  
</script>  
</body>  
</html>

# 6. Аналіз трендів запитів користувачів (Python, Pandas)

import pandas as pd  
  
# Завантаження логів  
logs = pd.read\_json('logs.json', lines=True)  
  
# Групування за типом дії  
trend = logs.groupby('action').size().reset\_index(name='count')  
print(trend)  
  
# Виявлення найпопулярніших дій  
top\_actions = trend.sort\_values('count', ascending=False)  
print(top\_actions)

# 7. Завдання до виконання

1. Налаштувати збір логів користувацької активності у форматі JSON.
2. Розробити дашборд з використанням Chart.js або D3.js.
3. Виконати аналіз трендів та побудувати графіки.
4. Створити API для отримання статистики у форматі JSON.
5. Підготувати звіт із кодом, графіками та висновками.

# 8. Вимоги до звіту

* Титульна сторінка.
* Мета роботи та опис методів аналітики.
* Код збору логів та побудови дашбордів.
* Скріншоти роботи графіків та аналітики.
* Висновки та виявлені тренди.

# 9. Питання до захисту

* Як реалізується збір логів користувацької активності?
* Чим відрізняється Chart.js від D3.js?
* Як будуються дашборди для аналітики?
* Які методи використовуються для виявлення трендів?
* Як інтегрувати аналітику у веб-додаток?

**Лабораторна робота №10**

«Інтеграція веб-застосунків із зовнішніми сервісами»

**Вступ.** Десята лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена інтеграції веб-застосунків із зовнішніми сервісами та API. Якщо попередні роботи зосереджувалися на створенні власних функціональних модулів, то тепер студенти вчаться використовувати готові рішення, що надаються сторонніми провайдерами. Це дозволяє значно розширити можливості застосунку без необхідності розробляти всі компоненти «з нуля».

Основна увага приділяється роботі з **REST та GraphQL API**, інтеграції платіжних систем і використанню механізмів **webhook**, які забезпечують обробку подій у реальному часі.

У процесі виконання роботи студенти навчаться:

* підключати REST API для отримання й відправки даних;
* використовувати GraphQL API, формуючи гнучкі запити;
* налаштовувати інтеграцію з платіжними сервісами (Stripe, LiqPay, PayPal);
* створювати й обробляти webhook-и для відстеження подій у зовнішніх системах;
* тестувати роботу API за допомогою Postman.

Практична цінність цієї роботи полягає у розвитку навичок роботи з API та інтеграції сторонніх сервісів, що є критично важливими для сучасних веб-додатків. Студенти здобувають досвід побудови застосунків, які можуть працювати з великою кількістю зовнішніх даних, підтримувати платіжні операції та реагувати на події у реальному часі. Це наближає їхні проєкти до реальних умов роботи комерційних систем.

# 1. Мета та результати навчання

Мета: навчитися інтегрувати застосунки з REST та GraphQL API, використовувати платіжні сервіси та обробляти webhook-и.

Після виконання роботи студент повинен уміти:

* підключати REST та GraphQL API;
* інтегрувати платіжні системи (Stripe, LiqPay, PayPal);
* створювати та обробляти webhook-и;
* тестувати інтеграцію API за допомогою Postman.

# 2. Короткі теоретичні відомості

* REST API — обмін даними через HTTP-запити (GET, POST, PUT, DELETE).
* GraphQL — отримання даних через запити, де клієнт сам формує структуру відповіді.
* Webhooks — механізм зворотного виклику для отримання подій у реальному часі.
* Stripe, LiqPay, PayPal — приклади платіжних сервісів із REST/GraphQL API.

# 3. Приклад інтеграції REST API (Node.js + Express)

const express = require('express');  
const axios = require('axios');  
const app = express();  
  
app.get('/weather', async (req, res) => {  
 const city = req.query.city || 'Kyiv';  
 const response = await axios.get(  
 `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=YOUR\_API\_KEY`  
 );  
 res.json(response.data);  
});  
  
app.listen(3000, () => console.log('Server running'));

# 4. Приклад інтеграції GraphQL API (Apollo Client)

npm install @apollo/client graphql  
  
import { ApolloClient, InMemoryCache, gql } from "@apollo/client";  
  
const client = new ApolloClient({  
 uri: "https://countries.trevorblades.com/",  
 cache: new InMemoryCache()  
});  
  
client.query({  
 query: gql`  
 {  
 countries {  
 code  
 name  
 }  
 }  
 `  
}).then(result => console.log(result));

# 5. Обробка webhook від Stripe (Node.js)

const express = require('express');  
const bodyParser = require('body-parser');  
const app = express();  
const endpointSecret = "whsec\_...";  
  
app.post('/webhook', bodyParser.raw({type: 'application/json'}), (req, res) => {  
 const sig = req.headers['stripe-signature'];  
 let event;  
 try {  
 event = stripe.webhooks.constructEvent(req.body, sig, endpointSecret);  
 } catch (err) {  
 res.status(400).send(`Webhook Error: ${err.message}`);  
 return;  
 }  
 if (event.type === 'payment\_intent.succeeded') {  
 console.log('Payment succeeded:', event.data.object);  
 }  
 res.json({received: true});  
});  
  
app.listen(4242, () => console.log('Running webhook handler'));

# 6. Завдання до виконання

1. Інтегрувати будь-який REST API (публічний або власний).
2. Реалізувати роботу з GraphQL API.
3. Налаштувати інтеграцію платіжної системи (Stripe/LiqPay/PayPal).
4. Створити та протестувати webhook.
5. Підготувати звіт із кодом і результатами тестування.

# 7. Вимоги до звіту

* Титульна сторінка.
* Мета роботи та опис інтегрованих API.
* Код підключення REST/GraphQL API.
* Приклади роботи webhook-ів.
* Скріншоти тестування у Postman.
* Висновки.

# 8. Питання до захисту

* У чому різниця між REST та GraphQL API?
* Як працюють webhook-и та де їх застосовують?
* Як налаштовуються платіжні інтеграції?
* Як тестувати API у Postman?
* Що робити у випадку помилок інтеграції API?

**Лабораторна робота №11**

«Оптимізація продуктивності та забезпечення безпеки веб-застосунків»

**Вступ.**Одинадцята лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* присвячена питанням оптимізації продуктивності та безпеки веб-застосунків. Якщо попередні роботи зосереджувалися на функціональності та інтеграціях, то тепер студенти мають навчитися підвищувати швидкодію систем і захищати їх від потенційних загроз.

Основна увага приділяється оптимізації фронтенду й бекенду, застосуванню механізмів кешування, стиснення, захисту API від надмірних запитів, а також впровадженню аутентифікації та шифрування за допомогою JWT.

У процесі виконання роботи студенти навчаться:

* аналізувати продуктивність веб-застосунків та виявляти вузькі місця;
* оптимізовувати фронтенд через мінімізацію ресурсів, кешування й lazy loading;
* підвищувати ефективність бекенду за рахунок стиснення, кешування та шифрування;
* захищати API за допомогою rate limiting та валідації запитів;
* впроваджувати механізми аутентифікації з JWT і контролювати доступ до приватних маршрутів;
* використовувати алгоритми шифрування для забезпечення конфіденційності даних.

Практична цінність цієї роботи полягає в розвитку ключових компетенцій, які виходять за межі простої розробки функціоналу. Студенти оволодівають інструментами, що дозволяють робити веб-застосунки швидкими, стійкими до навантажень і безпечними для користувачів. Ці знання є фундаментом для подальшого професійного зростання, оскільки оптимізація та безпека є одними з найважливіших вимог у сучасній веб-розробці.

# 1. Мета та результати навчання

Мета: навчитися оптимізовувати продуктивність фронтенду та бекенду, налаштовувати захист API від DDoS та несанкціонованих запитів, а також реалізовувати шифрування даних та аутентифікацію за допомогою JWT.

Після виконання роботи студент повинен уміти:

* аналізувати продуктивність веб-застосунків;
* виконувати оптимізацію швидкодії фронтенду та бекенду;
* захищати API від DDoS і несанкціонованих запитів;
* шифрувати дані та працювати з JWT токенами.

# 2. Короткі теоретичні відомості

* Оптимізація фронтенду: мінімізація ресурсів, кешування, lazy loading.
* Оптимізація бекенду: зменшення часу відповіді API, використання кешу на стороні сервера.
* Захист API: використання rate limiting, валідація запитів, захист від brute-force атак.
* JWT (JSON Web Token): метод передачі зашифрованих даних між клієнтом та сервером.
* Шифрування даних: використання алгоритмів AES, RSA тощо.

# 3. Оптимізація бекенду (Node.js + Express)

const express = require('express');  
const compression = require('compression');  
const helmet = require('helmet');  
const app = express();  
  
// Увімкнення стиснення відповідей  
app.use(compression());  
  
// Захист заголовків HTTP  
app.use(helmet());  
  
app.get('/', (req, res) => {  
 res.send('Оптимізований сервер працює!');  
});  
  
app.listen(3000, () => console.log('Server running on port 3000'));

# 4. Захист API за допомогою rate limiting

const rateLimit = require('express-rate-limit');  
  
const limiter = rateLimit({  
 windowMs: 15 \* 60 \* 1000, // 15 хвилин  
 max: 100, // максимум 100 запитів  
 message: 'Занадто багато запитів з цього IP, спробуйте пізніше.'  
});  
  
app.use('/api/', limiter);

# 5. Приклад аутентифікації з JWT (Node.js)

const jwt = require('jsonwebtoken');  
const SECRET\_KEY = 'mysecret';  
  
// Створення токена  
app.post('/login', (req, res) => {  
 const { username } = req.body;  
 const token = jwt.sign({ username }, SECRET\_KEY, { expiresIn: '1h' });  
 res.json({ token });  
});  
  
// Перевірка токена  
function authenticateToken(req, res, next) {  
 const token = req.headers['authorization'];  
 if (!token) return res.sendStatus(403);  
 jwt.verify(token, SECRET\_KEY, (err, user) => {  
 if (err) return res.sendStatus(403);  
 req.user = user;  
 next();  
 });  
}  
  
app.get('/protected', authenticateToken, (req, res) => {  
 res.send(`Привіт, ${req.user.username}, ви авторизовані!`);  
});

# 6. Завдання до виконання

1. Проаналізувати продуктивність власного застосунку.
2. Оптимізувати фронтенд: мінімізація ресурсів, використання кешу.
3. Оптимізувати бекенд: додати стиснення, кешування, шифрування.
4. Захистити API від надмірних запитів за допомогою rate limiting.
5. Реалізувати аутентифікацію з JWT та захист приватних маршрутів.
6. Підготувати звіт із кодом і результатами тестування.

# 7. Вимоги до звіту

* Титульна сторінка.
* Мета роботи та використані методи оптимізації.
* Код оптимізації фронтенду та бекенду.
* Код реалізації JWT та захисту API.
* Скріншоти результатів тестування.
* Висновки.

# 8. Питання до захисту

* Які основні методи оптимізації фронтенду та бекенду?
* Що таке rate limiting і навіщо його застосовують?
* Що таке JWT та як він працює?
* Які алгоритми шифрування можна використовувати у веб-додатках?
* Як протестувати захищені маршрути API?

**Лабораторна робота №12**

«Підсумковий проєкт: розробка, документування та презентація»

**Вступ.** Дванадцята лабораторна робота дисципліни *«Web-базовані інтелектуальні додатки та сервіси»* є підсумковим проєктом курсу. Вона узагальнює всі знання й навички, набуті під час виконання попередніх робіт, і передбачає створення повноцінного веб-застосунку з інтеграцією сучасних технологій.

Основна мета роботи — сформувати у студентів комплексне розуміння процесу розробки від ідеї до деплою. Це включає вибір архітектури (моноліт або мікросервіси), побудову клієнтської та серверної частини, підключення бази даних, інтеграцію зовнішніх API та реалізацію безпеки й авторизації. Окрім технічної реалізації, важливим є документування API та підготовка презентації результатів.

У процесі виконання підсумкового проєкту студенти навчаться:

* розробляти архітектуру веб-застосунків різних типів (монолітні та мікросервісні);
* поєднувати фронтенд, бекенд, базу даних і зовнішні API в єдину систему;
* документувати REST або GraphQL API за допомогою Swagger/OpenAPI;
* автоматизувати деплой у хмарі за допомогою CI/CD;
* готувати презентаційні матеріали (звіт, діаграми, дашборди, демонстрацію роботи застосунку).

Практична цінність цієї роботи полягає у тому, що студенти отримують досвід, максимально наближений до реальної розробки у командних і промислових умовах. Виконання підсумкового проєкту формує компетенції, які дозволяють не лише створювати сучасні веб-сервіси, а й документувати та презентувати їх результати, що є важливим аспектом професійної діяльності.

# 1. Мета та результати навчання

Мета: створити повноцінний веб-базований інтелектуальний сервіс, який об'єднує всі вивчені технології, документувати архітектуру рішення, API та підготувати публікацію та презентацію результатів.

Після виконання роботи студент повинен уміти:

* розробляти архітектуру веб-застосунку;
* інтегрувати бекенд, фронтенд та базу даних;
* реалізовувати API та документувати його;
* готувати технічну документацію, презентацію та публікацію результатів.

# 2. Короткі теоретичні відомості

* Мікросервісна або монолітна архітектура застосунку — залежно від цілей проєкту.
* REST та GraphQL API — методи взаємодії фронтенду і бекенду.
* Документування API — використання Swagger/OpenAPI для опису маршрутів.
* CI/CD — автоматизація деплою на хмарні сервіси (Heroku, Render, AWS).
* Презентація результатів — підготовка звіту, діаграм, дашбордів та демонстрацій.

# 3. Приклад архітектури фінального проєкту

Frontend (React / Vue / Angular)  
 |  
 v  
API Gateway (REST / GraphQL) <---> Authentication (JWT)  
 |  
 v  
Backend (Node.js / Flask / Django)  
 |  
 v  
Database (PostgreSQL / MongoDB)  
 |  
 v  
External APIs (OpenAI, Stripe, Google AI)

# 4. Документування API за допомогою Swagger (Node.js + Express)

npm install swagger-ui-express swagger-jsdoc  
  
const express = require('express');  
const swaggerUi = require('swagger-ui-express');  
const swaggerJsdoc = require('swagger-jsdoc');  
  
const app = express();  
  
const swaggerSpec = swaggerJsdoc({  
 definition: {  
 openapi: "3.0.0",  
 info: { title: "Final Project API", version: "1.0.0" },  
 },  
 apis: ["./routes/\*.js"],  
});  
  
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(swaggerSpec));  
  
app.listen(3000, () => console.log("Swagger docs available at /api-docs"));

# 5. Завдання до виконання

1. Розробити веб-застосунок із використанням сучасних фреймворків.
2. Інтегрувати бекенд, фронтенд, базу даних і зовнішні API.
3. Створити REST або GraphQL API та задокументувати його.
4. Розгорнути проєкт на одній із хмарних платформ (Heroku, Render, AWS).
5. Підготувати презентацію та звіт з описом архітектури, API та отриманих результатів.

# 6. Вимоги до звіту

* Титульна сторінка.
* Мета проєкту та опис його архітектури.
* Діаграма архітектури (моноліт або мікросервіси).
* Документація API (Swagger/OpenAPI).
* Результати тестування та скріншоти роботи застосунку.
* Висновки та рекомендації.

# 7. Питання до захисту

* Опишіть архітектуру вашого застосунку.
* Які зовнішні API використовувались і як їх інтегрували?
* Як реалізовано авторизацію та аутентифікацію?
* Які інструменти використані для документування API?
* Як виконувалось тестування та деплой застосунку?

## Висновок

Виконання лабораторних робіт з дисципліни «Web-базовані інтелектуальні додатки та сервіси» дозволило студентам послідовно пройти всі ключові етапи створення сучасних веб-застосунків — від ознайомлення з основами HTML, CSS і JavaScript до реалізації повноцінних інтелектуальних сервісів, інтегрованих із зовнішніми API та хмарними платформами.

У процесі виконання робіт студенти здобули такі знання та навички:

* **базові**: розмітка та стилізація веб-сторінок, організація клієнтської логіки, використання Git і GitHub;
* **поглиблені**: реалізація клієнт–серверної взаємодії, побудова REST та GraphQL API, робота з базами даних, інтеграція зовнішніх сервісів;
* **інтелектуальні**: застосування методів машинного навчання, створення чат-ботів і рекомендаційних систем, використання NLP-моделей;
* **практичні**: деплой у хмарні середовища, організація CI/CD-процесів, моніторинг працездатності, забезпечення безпеки застосунків;
* **аналітичні**: збір, обробка та візуалізація даних користувачів, побудова інтерактивних дашбордів для прийняття рішень;
* **підсумкові**: реалізація комплексного проєкту з архітектурою «клієнт–сервер–хмара» та підготовка документації за стандартами Swagger/OpenAPI.

Практична цінність курсу полягає у тому, що студенти не лише ознайомилися з теоретичними аспектами веб-розробки та інтелектуальних технологій, а й отримали досвід практичного застосування цих знань у реальних сценаріях. Вони пройшли шлях від простих статичних сторінок до багатофункціональних веб-сервісів, що інтегрують сучасні інструменти штучного інтелекту та працюють у хмарних середовищах.

Таким чином, курс сформував у студентів **комплекс компетенцій**: технічних, аналітичних, практичних і дослідницьких. Ці знання та навички є основою для подальшої професійної діяльності у сфері веб-розробки та розробки інтелектуальних систем, а також відповідають актуальним вимогам ІТ-індустрії.

**Список використаної літератури**

1. Mozilla Developer Network. HTML Living Standard, CSS, JavaScript Docs. MDN Web Docs. URL: https://developer.mozilla.org/ (дата звернення: 12.09.2025).

2. W3C. Web Standards: HTML, CSS, JavaScript. World Wide Web Consortium. URL: https://www.w3.org/ (дата звернення: 12.09.2025).

3. Node.js Foundation. Node.js v20.x Documentation. 2023. URL: https://nodejs.org/ (дата звернення: 12.09.2025).

4. Flask. Flask Web Development Documentation. 2023. URL: https://flask.palletsprojects.com/ (дата звернення: 12.09.2025).

5. Docker. Docker Official Documentation. 2024. URL: https://docs.docker.com/ (дата звернення: 12.09.2025).

6. TensorFlow.js. Machine Learning for JavaScript Developers. 2023. URL: https://www.tensorflow.org/js (дата звернення: 12.09.2025).

7. OpenAI. API Reference. 2024. URL: https://platform.openai.com/docs/ (дата звернення: 12.09.2025).

8. Chart.js. Chart.js Documentation. 2023. URL: https://www.chartjs.org/docs/ (дата звернення: 12.09.2025).

9. Kubernetes Authors. Kubernetes Documentation. 2024. URL: https://kubernetes.io/docs/ (дата звернення: 12.09.2025).

10. Mujumdar A. S., Law C. L. Drying of Materials and Applications in Food Industry. Journal of Food Engineering, 2022. DOI: 10.1016/j.jfoodeng.2022.110985.

11. Kumar C., Karim M. A. Applications of Machine Learning in Food and Process Engineering. Applied Sciences, 2023. DOI: 10.3390/app132210111.

12. Farkas I. Smart Systems for Web-based Industrial Monitoring. Computers and Electronics in Agriculture, 2024. DOI: 10.1016/j.compag.2024.108512.