ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-застосунку для візуалізації історичних подій на основі онтологій»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Андрій ЧЕРНИШ
(підпис)

Виконав: здобувач вищої освіти групи ПД-42

Андрій ЧЕРНИШ

Керівник: Ірина ЩЕРБИНА
к.т.н., доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Чернишу Андрію Олеговичу

1. Тема кваліфікаційної роботи: «Розробка web-застосунку для візуалізації історичних подій на основі онтологій»
керівник кваліфікаційної роботи к.т.н., доцент Ірина ЩЕРБИНА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2024 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки web-застосунку, опис роботи web-застосунку, документація фреймворків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд застосунку "HistoricalEventsAppNew" та аналіз існуючих аналогів, визначення вимог до програмного забезпечення.
2. Огляд та аналіз засобів реалізації застосунку для роботи з історичними подіями.
3. Проєктування архітектури застосунку для роботи з історичними подіями.

4.Програмна реалізація та тестування застосунку для роботи з історичними подіями.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма розгортання.
5. Діаграма варіантів використання
6. Діаграма класів.
7. Екранні форми.
8. Демонстрація роботи застосунку.
9. Апробація результатів дослідження.

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд та аналіз застосунку	16.03-17.03.2025	
4	Огляд та аналіз засобів реалізації застосунку	18.03-21.03.2025	
5	Проєктування архітектури застосунку	22.03-04.04.2025	
6	Програмна реалізація та тестування застосунку	05.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Андрій ЧЕРНИШ
(підпис)

Керівник
кваліфікаційної роботи _____ Ірина ЩЕРБИНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 58 стор., 1 табл., 16 рис., 15 джерел.

Мета роботи – спрощення аналізу та візуалізації історичних подій на основі онтології.

Об'єкт дослідження – процес аналізу та візуалізації історичних подій на основі онтологічної моделі.

Предмет дослідження – веб-застосунок для управління історичними подіями з візуалізацією та використанням онтологій.

Короткий зміст роботи: У роботі проаналізовано предметну галузь та методи реалізації web-додатку для управління історичними подіями. Проведено огляд потреб цільової аудиторії (дослідники, студенти, викладачі, ентузіасти історії) та сформульовано вимоги до функціональності, зокрема: відображення інформації про історичні події, створення семантичних зв'язків між подіями, інтерактивна візуалізація у вигляді графа, обробка пошуку та фільтрація подій, авторизація користувачів із ролями (Admin і User). Розроблено веб-застосунок із використанням ASP.NET Core MVC для серверної частини, Razor і JavaScript (з бібліотеками D3.js і Tailwind CSS) для клієнтської частини, Entity Framework Core і SQL Server для зберігання даних. Реалізовано основні функціональні можливості: відображення списку подій із пошуком і фільтрацією, створення та редагування подій і зв'язків, інтерактивний граф зв'язків із D3.js, авторизація через ASP.NET Core Identity, адаптивний інтерфейс із Tailwind CSS. Проведено ручне тестування функціональності, зокрема коректність відображення сторінок, роботи графа зв'язків, авторизації та обробки запитів до бази даних.

КЛЮЧОВІ СЛОВА: ASP.NET CORE MVC, C#, ENTITY FRAMEWORK CORE, SQL SERVER, JAVASCRIPT, D3.JS, TAILWIND CSS, WEB-ЗАСТОСУНОК, ОНТОЛОГІЯ, ІСТОРИЧНІ ПОДІЇ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CRUD – Create, Read, Update, Delete (основні операції з даними: створення, читання, оновлення, видалення)

URL – Uniform Resource Locator (адреса сторінки, наприклад, /Events/Details/1)

CSS – Cascading Style Sheets (стилі для оформлення сторінок, у проєкті використовується Tailwind CSS)

JS – JavaScript (скрипти для інтерактивності, наприклад, модальне вікно видалення)

MVC – Model-View-Controller (архітектура, на якій побудовано проєкт)

Razor – Razor-синтаксис (використовується у .cshtml файлах для створення динамічних сторінок)

SQL SERVER – SQL Server Database (реляційна база даних, яка використовується для зберігання подій)

EF Core – Entity Framework Core (інструмент для роботи з базою даних SQLite через ORM)

ORM – Object-Relational Mapping (механізм для роботи з базою даних, реалізований через Entity Framework Core)

DI – Dependency Injection (використовується в ASP.NET Core для ін’єкції залежностей, наприклад, DatabaseService)

Tailwind – Tailwind CSS (бібліотека CSS для стилізації інтерфейсу)

ВСТУП

Актуальність: Історичні події є ключовим джерелом знань про минуле, яке допомагає краще зрозуміти сучасність і прогнозувати майбутнє. У сучасному цифровому світі з'являються нові можливості для аналізу та представлення історичних даних через web-застосунки, які дозволяють не лише зберігати інформацію, але й візуалізувати складні зв'язки між подіями. Використання онтологій у таких системах відкриває нові перспективи для структуризації даних, встановлення семантичних зв'язків і створення інтерактивних візуалізацій, що робить історичні дослідження більш доступними та інформативними для широкої аудиторії, включаючи дослідників, студентів, викладачів та ентузіастів історії.

Web-застосунок, який поєднує онтологічний підхід із сучасними технологіями візуалізації, може значно спростити аналіз історичних подій, дозволяючи користувачам бачити не лише окремі факти, а й їх взаємозв'язки у вигляді графів або хронологічних ліній. Це забезпечує новий рівень взаємодії з історичними даними, роблячи процес дослідження більш інтуїтивним і ефективним.

Об'єктом дослідження є процес управління та візуалізації історичних подій із використанням онтологій.

Предметом дослідження є програмне забезпечення для структуризації, аналізу та візуалізації історичних подій.

Метою роботи є розробка web-застосунку для візуалізації історичних подій на основі онтологій з використанням ASP.NET Core MVC та SQL Server.

Методи дослідження: На першому етапі було проаналізовано потреби цільової аудиторії (дослідники, студенти, викладачі, ентузіасти історії) для формулювання функціональних і нефункціональних вимог до програмного забезпечення. Зокрема, було визначено необхідність створення онтологічної моделі для представлення зв'язків між подіями. Далі проведено огляд стеку технологій, що відповідають вимогам: обрано Visual Studio як IDE, C# і JavaScript як мови

програмування, ASP.NET Core MVC як веб-фреймворк, Entity Framework Core і SQL Server для роботи з базою даних, Tailwind CSS для стилізації інтерфейсу, а також JavaScript для створення інтерактивних елементів, таких як модальні вікна. Дослідження реалізації проводилося під час розробки, включаючи створення моделей, контролерів, сервісів, інтеграцію з базою даних, розробку пошуку, фільтрації та онтологічної структури даних, а також ручне тестування функціональності.

Наукова новизна роботи визначається наступними складовими: створення онтологічної моделі для представлення історичних подій та їх зв'язків із можливістю подальшої візуалізації у вигляді графів; розробка гнучкої системи фільтрації подій за кількома критеріями (назва, дата, категорія, місце, учасник) з вибором значень із списків; використання модальних вікон для підтвердження видалення подій із плавною анімацією.

Практична значущість результатів полягає в можливості використання розробленого web-застосунку для структуризації, аналізу та візуалізації історичних подій на основі онтологій, що може бути застосовано у навчальних закладах, дослідницьких проєктах або для особистих історичних досліджень.

1 АНАЛІЗ ЗАСТОСУНКУ ДЛЯ ВІЗУАЛІЗАЦІЇ ІСТОРИЧНИХ ПОДІЙ НА ОСНОВІ ОНТОЛОГІЙ

1.1 Цільова аудиторія платформи

Веб-застосунок "HistoricalEventsAppNew" орієнтований на користувачів, які цікавляться історією та її аналізом. До основних груп користувачів належать:

- науковці та історики: дослідники, які використовують платформу для аналізу історичних подій, їх взаємозв’язків і візуалізації у вигляді інтерактивного графа;
- студенти: особи, які вивчають історію та потребують зручного інструменту для аналізу подій і їхньої семантики;
- викладачі: педагоги, які застосовують додаток для підготовки навчальних матеріалів на основі історичних даних;
- любителі історії: люди, які захоплюються історією та хочуть досліджувати її через інтерактивний інтерфейс із можливістю візуалізації зв’язків.

1.2 Особливості створення інструменту для аналізу історичних даних

Розробка веб-застосунку для роботи з історичними подіями на основі онтологічної моделі має свої специфічні особливості, які виділяють її серед інших рішень для аналізу даних. Розглянемо ключові аспекти:

Технічні характеристики:

- структуризація даних: інтеграція онтологічної моделі для визначення семантичних зв’язків між подіями (наприклад, "передувала", "спричинила"), що забезпечує їх подальшу візуалізацію у вигляді графа;
- інтерактивність: використання javascript для створення модальних вікон із плавними анімаціями та d3.js для інтерактивного графа з можливістю перетягування вузлів;

– стабільність: застосування sql server і entity framework core для надійного зберігання та управління даними.

Аналітичні можливості:

– графічна візуалізація: реалізація інтерактивного графа зв’язків за допомогою d3.js, що дозволяє користувачам бачити семантичні зв’язки між подіями у вигляді вузлів і ребер із відображенням типів зв’язків (наприклад, "передувала");

– фільтрація даних: інструменти для аналізу подій за різними критеріями (дата, категорія, місце) із можливістю вибору значень зі списків;

– зручний доступ: інтерфейс, що забезпечує швидкий перегляд деталей подій, включаючи пов’язані події та медіаресурси.

Організаційні особливості:

– семантична модель: використання онтологічного підходу для створення зв’язків між подіями, що дозволяє визначати їхню семантику та представляти у вигляді графа;

– універсальність даних: події охоплюють різні епохи, категорії та регіони, що потребує гнучкої моделі даних для їх обробки;

– безпека доступу: розподіл ролей між адміністраторами (admin) і користувачами (user) для захисту даних і управління змінами.

1.3 Загальна характеристика веб-застосунку "HistoricalEventsAppNew"

Платформа "HistoricalEventsAppNew" є сучасним рішенням для роботи з історичними даними, їхньої структуризації та інтерактивного відображення. Застосунок забезпечує збереження детальної інформації про історичні події, включаючи назви, дати, категорії, місця, учасників і медіаресурси, а також дозволяє встановлювати зв’язки між подіями через онтологічну модель. Головна мета розробки – створення інструменту для глибокого аналізу історичних подій шляхом їхньої візуалізації у форматі графів та спрощення пошуку і аналізу даних за допомогою розширених функцій фільтрації.

Основні складові платформи:

– модулі інформації: сторінки з деталями подій, що включають текстові дані, дати, категорії, зображення та перелік пов’язаних подій;

– семантична модель: структурована система даних для визначення зв’язків між подіями (наприклад, "передувала", "спричинила", "була одночасно з"), що забезпечує багатошарові взаємозв’язки;

– інструменти пошуку: функціонал для швидкого пошуку подій за ключовими словами та фільтрації за атрибутами (дата, місце, категорія, учасник) через списки вибору;

– графічна інтерактивність: граф зв’язків між подіями, створений за допомогою d3.js, що дозволяє переглядати семантичні зв’язки з можливістю перетягування вузлів;

– елементи взаємодії: модальні вікна для підтвердження видалення подій із плавними анімаціями, реалізовані через javascript;

– контроль доступу: система ролей (admin і user) для управління правами редагування та видалення подій.

Цілі розробки:

– структуризувати історичні дані через онтологічну модель із семантичними зв’язками;

– забезпечити інтерактивну візуалізацію зв’язків між подіями у форматі графа.

– спростити доступ до історичних даних за допомогою розширених функцій пошуку та фільтрації;

– створити зручний і адаптивний інтерфейс із інтерактивними елементами.

– гарантувати безпеку управління даними через систему авторизації та розподіл ролей;

1.4 Аналіз аналогічних рішень

На ринку існує кілька платформ для роботи з історичними даними. Розглянемо три відомі аналоги: History Timeline, World History Encyclopedia та

ChronoZoom, і порівняємо їх із розробленим рішенням.

1.4.1 Аналіз платформи History Timeline

History Timeline — це онлайн-платформа, призначена для перегляду історичних подій у форматі часової шкали з акцентом на хронологічному порядку.

Функціонал History Timeline:

– часова шкала: події представлені у хронологічному порядку з можливістю фільтрації за роками;

– дані про події: короткі описи, дати та пов’язані зображення;

– пошук подій: фільтрація за ключовими словами та категоріями;

– тематична категоризація: події розподілені за темами (наприклад, "війни", "політика");

– мобільна підтримка: доступ через веб-інтерфейс і додатки для мобільних пристроїв.

Переваги:

– простота використання: інтуїтивний інтерфейс для перегляду подій у часовому порядку;

– візуалізація: зручна хронологічна шкала для аналізу подій;

– доступність: підтримка веб-версії та мобільних платформ.

Недоліки:

– обмежена інтерактивність: відсутність графів для відображення зв’язків між подіями;

– відсутність онтологій: дані не структуризовані семантично, немає визначення зв’язків;

– неможливість внесення даних: користувачі не можуть додавати власні події чи зв’язки.

Приклад головної сторінки History Timeline наведено на рисунку 1.1.

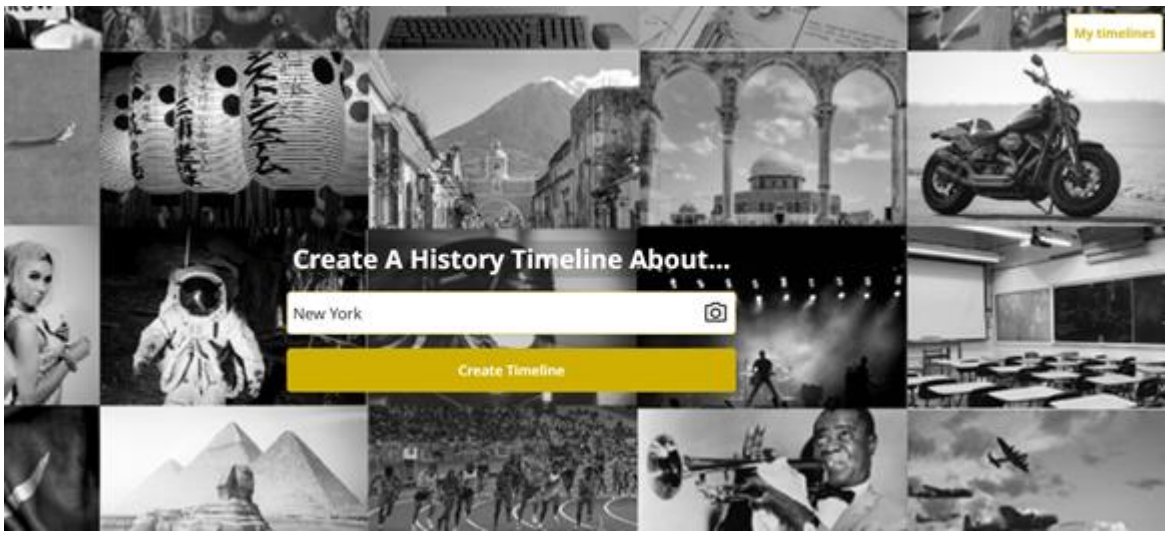


Рис. 0.1 Приклад головної сторінки History Timeline

1.4.2 Огляд World History Encyclopedia

World History Encyclopedia — це платформа, яка фокусується на наданні детальної інформації про історичні події та пов’язані матеріали з акцентом на освітні цілі.

Функціонал World History Encyclopedia:

– пошук подій: фільтрація за назвами, періодами та темами;

– детальні статті: розширені описи подій із посиланнями на пов’язані матеріали;

– категоризація за епохами: події розподілені за історичними періодами;

– освітні ресурси: матеріали для навчання, включаючи карти та хронології.

Переваги:

– глибина описів: детальна інформація з посиланнями на пов’язані статті;

– освітній фокус: ресурси для студентів і викладачів;

– різноманітність даних: наявність карт, статей і хронологій.

Недоліки:

– обмежена візуалізація: відсутність інтерактивних графів або семантичних зв’язків між подіями;

– відсутність онтологій: дані не структуризовані через семантичну модель;

– обмежена інтерактивність: неможливість додавання власних подій чи зв’язків користувачами.

На рисунку 1.2 зображено приклад пошуку та фільтрації подій World History Encyclopedia.

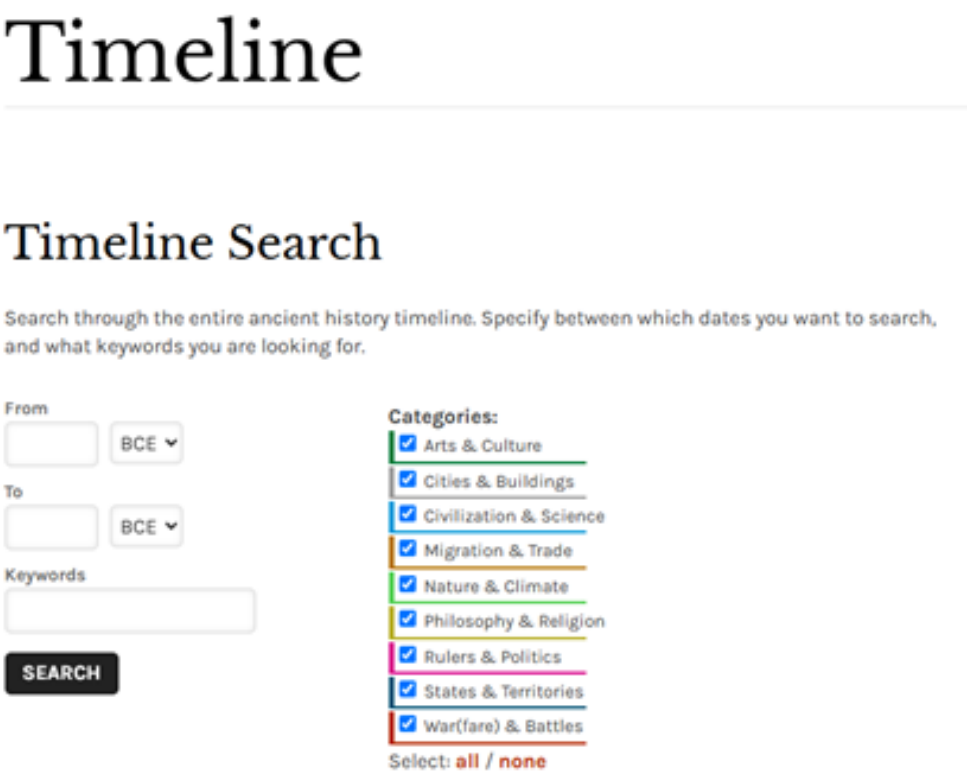


Рис. 0.2. Приклад пошуку та фільтрації подій World History Encyclopedia

Приклад сторінки входу World History Encyclopedia зображено на рисунку 1.3.

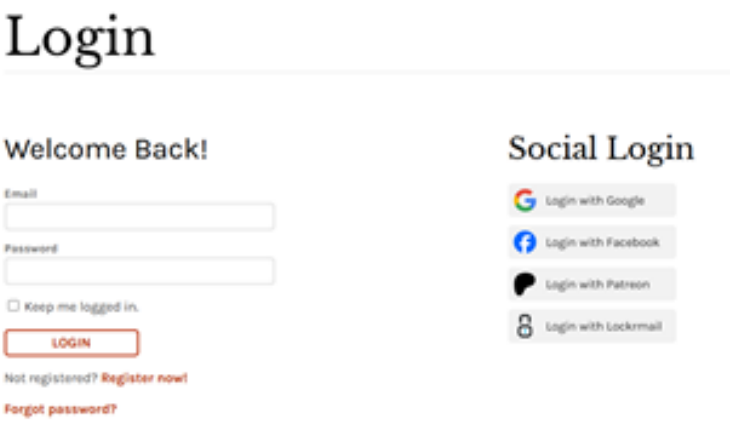


Рис. 0.3 Приклад сторінки входу World History Encyclopedia

1.4.3 Дослідження ChronoZoom

ChronoZoom — це платформа для інтерактивного дослідження історії через масштабований часовий масштаб, що охоплює події від Великого Вибуху до сучасності.

Функціонал ChronoZoom:

– Масштабований часовий масштаб: Перегляд подій на різних рівнях деталізації;

– Хронологічна візуалізація: Події відображені у форматі хронологічних ліній із можливістю масштабування (див.рисунок 1.4.);

– Категоризація подій: Поділ за епохами та темами;

– Освітні інструменти: Можливість створення власних хронологій.

Переваги:

– Інтерактивність: Унікальний масштабований інтерфейс для перегляду подій;

– Освітня цінність: Інструменти для створення власних хронологій;

– Візуалізація: Зручне представлення подій у часовому контексті.

Недоліки:

– Складність для новачків: Інтерфейс може бути незручним для нових користувачів;

– Відсутність онтологій: Немає семантичного підходу до визначення зв’язків між подіями;

– Обмежена кастомізація: Неможливість додавання власних зв’язків чи деталізованих даних.

Рис. 0.4 Приклад масштабованої хронології ChronoZoom

1.5 Вимоги до функціональності веб-застосунку

На основі аналізу роботи з історичними подіями, потреб цільової аудиторії та аналогів було визначено такі вимоги до платформи "HistoricalEventsAppNew":

- інформаційні блоки: відображення детальної інформації про історичні події, включаючи назву, дату, категорію, місце, учасників, медіаресурси та пов’язані події;
- семантична структура: реалізація онтологічної моделі для створення зв’язків між подіями (наприклад, "передувала", "спричинила") із можливістю їхньої візуалізації у вигляді графа;
- інструменти пошуку: функціонал для пошуку подій за ключовими словами та фільтрації за атрибутами (назва, дата, категорія, місце, учасник) через списки вибору.
- інтерактивна візуалізація: створення графа зв’язків між подіями за допомогою d3.js із можливістю перетягування вузлів і відображенням типів зв’язків.

–інтерфейс із взаємодією: модальні вікна з плавними анімаціями для видалення подій, реалізовані через javascript.

–система доступу: розподіл ролей (admin і user) для управління правами редагування та видалення подій.

–швидкість і адаптивність: забезпечення завантаження сторінок менш ніж за 2 секунди та адаптація інтерфейсу до різних розмірів екранів.

Таблиця 1.1

Порівняння аналогів

Критерій	**History Timeline**	**World History Encyclopedia**	**ChronoZoom**	**HistoricalEventsAppNew**
Основний функціонал	Хронологічна шкала, фільтрація за роками	Детальні статті, пошук за темами	Масштабований часовий масштаб	Перегляд подій, граф зв’язків, пошук
Інтерактивність	Обмежена (немає графів зв’язків)	Низька (статичні сторінки)	Висока (масштабування хронології)	Висока (граф D3.js, перетягування)
Онтологічна модель	Відсутня	Відсутня	Відсутня	Присутня (зв’язки між подіями)
Внесення даних	Неможливе	Неможливе	Обмежене (власні хронології)	Можливе (додавання, редагування)

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ РОЗРОБКИ

2.1 Інструменти для ефективної розробки

Процес створення програмного забезпечення потребує використання спеціалізованих інструментів, які допомагають розробникам оптимізувати робочий процес, підвищувати якість коду та спрощувати виконання складних завдань. Одним із ключових інструментів є інтегроване середовище розробки (IDE), яке об’єднує в собі широкий набір функцій для написання, редагування, компіляції, тестування та налагодження програмного коду. Такі середовища зазвичай включають редактори з підсвіткою синтаксису, що полегшує читання коду, інструменти для автоматизації компіляції, дебагери для виявлення помилок, а також інтеграцію з системами управління версіями, що дозволяє розробникам працювати в команді та відстежувати зміни в коді. Завдяки цьому значно підвищується продуктивність, а процес розробки стає більш структурованим і зручним.

2.1.1 Застосування Visual Studio для створення проєкту

Для розробки веб-застосунку "HistoricalEventsAppNew" було обрано Visual Studio 2022 — сучасне і багатофункціональне інтегроване середовище розробки від компанії Microsoft, яке забезпечує оптимальні умови для роботи з платформою .NET, зокрема з ASP.NET Core MVC. Visual Studio 2022 вирізняється своєю гнучкістю та широкими можливостями, які дозволяють розробникам ефективно вирішувати завдання різного рівня складності. Серед основних функцій цього IDE — підтримка IntelliSense, яка забезпечує автодоповнення коду, що значно прискорює написання програм і зменшує кількість помилок. Вбудований дебагер дозволяє швидко виявляти та виправляти помилки, відображаючи детальну інформацію про стан програми під час виконання. Інтеграція з системою управління версіями Git дає змогу легко відстежувати зміни в коді, працювати над проєктом у команді та повертатися до попередніх версій у разі потреби. Крім того, Visual Studio підтримує інструменти для аналізу продуктивності, що допомагають

оптимізувати роботу додатку, а також функції для роботи з базами даних, що спрощують налаштування та управління схемою даних.

У контексті проєкту "HistoricalEventsAppNew" Visual Studio 2022 стало основним інструментом для створення всіх компонентів веб-застосунку. Завдяки вбудованим шаблонам ASP.NET Core MVC розробка основних елементів, таких як контролери, моделі та представлення, була значно прискорена. Наприклад, створення контролера EventsController, який відповідає за обробку запитів, пов’язаних із історичними подіями, зайняло мінімум часу завдяки автоматичному генеруванню шаблонного коду. Visual Studio також забезпечило ефективну підтримку мови C#, яка є основною мовою програмування в цьому проєкті. За допомогою функції IntelliSense розробники могли швидко знаходити потрібні методи та властивості, наприклад, при написанні LINQ-запитів для роботи з базою даних через Entity Framework Core.

Однією з ключових переваг Visual Studio стала його здатність працювати з SQL Server через Entity Framework Core. Це дозволило легко створювати міграції для бази даних, налаштовувати схему даних і тестувати запити безпосередньо в IDE. Наприклад, для створення таблиць HistoricalEvents і HistoricalEventRelations було використано команди Add-Migration і Update-Database, які автоматично генерували SQL-скрипти для оновлення бази даних. Visual Studio також підтримує розширення, що дало змогу адаптувати середовище до специфічних потреб проєкту. Наприклад, використання розширення "SQL Server Object Explorer" дозволило візуально переглядати структуру бази даних, перевіряти таблиці та виконувати запити без необхідності використання додаткових інструментів, таких як SQL Server Management Studio. Інтеграція з системою збірки .NET спростила процес компіляції та запуску проєкту за допомогою команд dotnet build і dotnet run, що значно прискорило ітеративний процес розробки та тестування. Завдяки цьому розробники могли швидко перевіряти зміни в коді, наприклад, після додавання нового методу до EventsController для обробки пошуку подій.

Visual Studio 2022 також забезпечило зручний доступ до інструментів для профілювання продуктивності, що допомогло виявити та оптимізувати потенційні

вузькі місця в роботі додатку. Наприклад, аналіз часу виконання запитів до бази даних показав, що використання асинхронних методів, таких як ToListAsync(), значно зменшує час завантаження списку подій на сторінці /Events. Крім того, IDE підтримує інтеграцію з Azure для розгортання додатку, хоча в рамках цього проєкту розгортання проводилося лише локально. Загалом, використання Visual Studio 2022 дозволило створити стабільне і продуктивне середовище для розробки "HistoricalEventsAppNew", забезпечуючи зручність, швидкість і високу якість коду.

2.2 Технологічні рішення для роботи з базами даних

Успішне функціонування будь-якого веб-застосунку залежить від ефективного управління даними, що вимагає використання сучасних і надійних технологій для роботи з базами даних. Такі технології забезпечують стабільне зберігання, швидкий доступ до даних, їх обробку та захист від несанкціонованого доступу. У проєкті "HistoricalEventsAppNew" було застосовано кілька ключових інструментів для роботи з базами даних, які дозволили створити надійну і продуктивну систему для управління історичними подіями та їхніми зв’язками.

2.2.1 SQL Server для надійного зберігання інформації

Для забезпечення стабільного і безпечного зберігання даних у проєкті "HistoricalEventsAppNew" було обрано SQL Server — потужну реляційну систему управління базами даних (СУБД), розроблену компанією Microsoft. SQL Server вирізняється своєю здатністю працювати з великими обсягами даних, підтримкою складних запитів і транзакцій, а також високим рівнем безпеки. Ця СУБД забезпечує стабільне зберігання інформації про історичні події, їхні зв’язки, а також дані про користувачів і їхні ролі, що є основою для функціонування веб-застосунку.

У "HistoricalEventsAppNew" SQL Server використовується для створення і управління таблицями, такими як HistoricalEvents і HistoricalEventRelations, які є центральними елементами бази даних. Таблиця HistoricalEvents містить детальну інформацію про кожну подію, включаючи її назву (Title), дату (Date), категорію

(Category), місце (Location), учасників (Person), посилання на медіаресурси (MediaResource) і шлях до зображення (ImagePath). Наприклад, подія "Битва при Гастінгсі" зберігається з усіма цими атрибутами, що дозволяє користувачам переглядати повну інформацію на сторінці /Events/Details. Таблиця HistoricalEventRelations визначає зв’язки між подіями, включаючи ідентифікатори вихідної та пов’язаної подій (SourceEventId і TargetEventId), а також тип зв’язку (RelationType), наприклад, "передувала" чи "спричинила". Це дозволяє створювати онтологічну модель, яка є основою для аналізу і візуалізації зв’язків між подіями на сторінці /Events/Graph.

SQL Server підтримує транзакційну цілісність через властивості ACID (Atomicity, Consistency, Isolation, Durability), що гарантує безпеку даних навіть при інтенсивному використанні. Наприклад, при створенні нового зв’язку між двома подіями транзакція забезпечує, що обидва записи (SourceEventId і TargetEventId) будуть збережені коректно, або операція буде скасована, якщо виникне помилка. Це особливо важливо для збереження цілісності онтологічної моделі, де зв’язки між подіями є ключовим елементом. SQL Server також забезпечує вбудовані механізми авторизації та шифрування, що захищають дані від несанкціонованого доступу. У проєкті це дозволяє гарантувати, що лише авторизовані користувачі (наприклад, з роллю Admin) можуть виконувати операції редагування чи видалення подій.

Використання SQL Server у поєднанні з Entity Framework Core дозволило оптимізувати запити до бази даних, що забезпечило швидке завантаження даних на сторінках. Наприклад, на сторінці /Events/Graph запит для завантаження пов’язаних подій і зв’язків виконується за допомогою LINQ через EF Core, що дозволяє ефективно отримувати дані для візуалізації графа. SQL Server також підтримує індексацію, що прискорює пошук подій за ключовими атрибутами, такими як Category чи Date. Завдяки цьому час виконання запитів залишається мінімальним навіть при зростанні обсягу даних, що є важливим для масштабування додатку в майбутньому.

2.2.2 Entity Framework Core для оптимізації роботи з даними

Entity Framework Core (EF Core) — це сучасний об’єктно-реляційний мапер (ORM), розроблений компанією Microsoft, який використовується у проєкті "HistoricalEventsAppNew" для спрощення взаємодії з базою даних SQL Server. EF Core дозволяє розробникам працювати з даними на рівні об’єктів C#, автоматично мапуючи їх на таблиці бази даних, що значно зменшує обсяг коду, необхідного для виконання операцій створення, читання, оновлення та видалення (CRUD). Завдяки цьому інструменту розробники можуть зосередитися на бізнес-логіці, а не на написанні складних SQL-запитів вручну.

У проєкті "HistoricalEventsAppNew" EF Core застосовується для роботи з ключовими моделями даних, такими як HistoricalEvent і HistoricalEventRelation. Модель HistoricalEvent представляє історичну подію з полями, такими як Id, Title, Date, Category, Location, Person, MediaResource, ImagePath і UserId. Наприклад, подія "Відкриття Америки Колумбом" зберігається з усіма цими атрибутами, що дозволяє відображати повну інформацію на сторінці деталей. Модель HistoricalEventRelation визначає зв’язки між подіями, включаючи поля SourceEventId, TargetEventId і RelationType, що є основою для створення онтологічної моделі. Наприклад, зв’язок між подіями "Відкриття Америки Колумбом" і "Французька революція" із типом "Caused" дозволяє користувачам аналізувати причинно-наслідкові зв’язки через граф зв’язків.

EF Core підтримує створення міграцій, що значно спрощує оновлення схеми бази даних при змінах у вимогах. Наприклад, коли до таблиці HistoricalEventRelations було додано поле RelationType для визначення типу зв’язку, виконано міграцію за допомогою команд dotnet ef migrations add AddRelationType і dotnet ef database update. Це автоматично оновило базу даних, додавши нове поле без необхідності ручного редагування SQL-скриптів. EF Core також забезпечує асинхронні методи, такі як ToListAsync(), що підвищують продуктивність при отриманні даних. Наприклад, на сторінці /Events список подій завантажується асинхронно через метод GetHistoricalEventsAsync у сервісі DatabaseService, що дозволяє уникнути блокування основного потоку і забезпечує

швидке відображення сторінки.

Інтеграція EF Core з механізмом Dependency Injection у ASP.NET Core MVC дозволяє передавати контекст бази даних (ApplicationDbContext) у сервіси, такі як DatabaseService, що робить код більш модульним і легким для тестування. Наприклад, DatabaseService інкапсулює логіку роботи з базою даних, дозволяючи контролерам, таким як EventsController, зосередитися на обробці HTTP-запитів. EF Core також підтримує валідацію даних на рівні моделей через атрибути, такі як [Required] і [StringLength]. Наприклад, поле Title у HistoricalEvent позначено як [Required], що гарантує, що подія не може бути створена без назви, а [StringLength(200)] обмежує довжину назви до 200 символів, що забезпечує коректність даних.

EF Core забезпечує підтримку каскадного видалення, що є важливим для онтологічної моделі. Наприклад, якщо подія видаляється з таблиці HistoricalEvents, усі пов’язані записи в таблиці HistoricalEventRelations видаляються автоматично, що запобігає появі "осиротілих" зв’язків. Це значно спрощує управління даними і забезпечує цілісність бази. Завдяки EF Core розробка "HistoricalEventsAppNew" стала більш ефективною, а взаємодія з базою даних — швидкою та безпечною, що є критично важливим для забезпечення якісного користувацького досвіду.

2.3 Програмування серверної та клієнтської частин

Розробка веб-застосунку "HistoricalEventsAppNew" вимагала використання різних мов програмування для реалізації серверної та клієнтської частин. Кожна з мов виконувала свою унікальну роль, забезпечуючи створення логіки, обробку даних і зручний користувацький інтерфейс. Вибір мов був обґрунтований їхньою сумісністю з технологіями, які застосовувалися в проєкті, а також їхньою здатністю ефективно вирішувати поставлені завдання.

2.3.1 C# як основа серверної логіки

C# — це потужна об’єктно-орієнтована мова програмування, створена компанією Microsoft у 2000 році, яка стала основним інструментом для реалізації

серверної логіки у "HistoricalEventsAppNew". Вона є частиною екосистеми .NET і ідеально підходить для роботи з фреймворком ASP.NET Core MVC, який використовується у проєкті. C# вирізняється своєю строгістю типізації, що допомагає уникати багатьох помилок на етапі компіляції, автоматичним управлінням пам’яттю через механізм garbage collection, а також підтримкою принципів об’єктно-орієнтованого програмування (ООП), таких як інкапсуляція, спадкування і поліморфізм. Це дозволяє створювати модульний, легко підтримуваний і масштабований код, що є важливим для проєктів, які можуть розширюватися в майбутньому.

У "HistoricalEventsAppNew" C# застосовується для створення всіх ключових компонентів серверної частини, включаючи контролери, моделі та сервіси. Наприклад, контролер EventsController обробляє HTTP-запити, пов’язані з історичними подіями, такі як створення, редагування, видалення та відображення графа зв’язків на сторінці /Events/Graph. Дія Index у цьому контролері завантажує список подій із можливістю пошуку та фільтрації, тоді як дія Graph передає дані для візуалізації зв’язків між подіями. Модель HistoricalEvent визначає структуру даних для подій, включаючи атрибути, такі як Title, Date, Category, Location, Person, MediaResource і ImagePath. Наприклад, подія "Перший політ братів Райт" зберігається з усіма цими атрибутами, що дозволяє відображати її деталі на сторінці /Events/Details. Модель HistoricalEventRelation використовується для визначення зв’язків між подіями, таких як "передувала" чи "спричинила", що є основою для онтологічного аналізу.

C# підтримує асинхронне програмування через ключові слова async і await, що значно підвищує продуктивність при взаємодії з базою даних SQL Server через Entity Framework Core. Наприклад, метод GetHistoricalEventsAsync у сервісі DatabaseService завантажує список подій асинхронно, що дозволяє уникнути блокування основного потоку і забезпечує швидке відображення сторінок. Це особливо важливо для сторінок із великою кількістю даних, таких як /Events, де користувач може переглядати сотні подій. Асинхронність також допомагає оптимізувати запити для графа зв’язків, де потрібно завантажувати як події, так і

пов’язані зв’язки.

C# також забезпечує зручну роботу з валідацією даних через атрибути, такі як [Required] і [StringLength], що застосовуються до моделей. Наприклад, у моделі HistoricalEvent поле Title позначено як [Required], що гарантує, що подія не може бути створена без назви, а [StringLength(200)] обмежує довжину назви до 200 символів, що допомагає уникнути надто довгих записів. Завдяки C# код серверної частини "HistoricalEventsAppNew" є структурованим, надійним і легким для підтримки, що дозволяє швидко вносити зміни та додавати нові функції в майбутньому.

2.3.2 JavaScript для створення динамічного інтерфейсу

JavaScript — це універсальна високорівнева інтерпретована мова програмування, яка стала основою для створення інтерактивних елементів у "HistoricalEventsAppNew". Вона була створена Бренданом Айком у 1995 році і з того часу стала однією з ключових технологій веб-розробки, поряд із HTML і CSS. JavaScript підтримує різні парадигми програмування, включаючи об’єктно-орієнтовану, функціональну та процедурну, що робить її гнучким інструментом для реалізації клієнтської логіки. У проєкті JavaScript відіграє важливу роль у створенні динамічного і зручного інтерфейсу, який дозволяє користувачам взаємодіяти з історичними даними у зручний і наочний спосіб.

У "HistoricalEventsAppNew" JavaScript використовується для реалізації кількох ключових функцій. Однією з них є створення модальних вікон із плавними анімаціями, які з’являються при виконанні певних дій, наприклад, при видаленні події. На сторінці Details.cshtml кнопка "Видалити" викликає JavaScript-функцію, яка відображає модальне вікно з анімацією появи, дозволяючи користувачу підтвердити або скасувати дію. Це покращує користувацький досвід, роблячи взаємодію з додатком більш інтуїтивною і безпечною, оскільки користувач має можливість переглянути своє рішення перед остаточним видаленням.

Найважливішою функцією JavaScript у проєкті є інтеграція з бібліотекою D3.js для створення інтерактивного графа зв’язків між подіями. На сторінці

Graph.cshtml JavaScript-код обробляє дані, отримані від сервера через Razor (наприклад, ViewBag.EventNodes і ViewBag.EventLinks), і передає їх у D3.js для візуалізації. D3.js створює граф, де вузли представляють події, а ребра — зв’язки між ними, такі як "передувала" чи "спричинила". Користувачі можуть перетягувати вузли, що дозволяє зручно аналізувати зв’язки між подіями. Наприклад, користувач може перемістити вузол "Французька революція", щоб краще розглянути його зв’язки з іншими подіями, такими як "Відкриття Америки Колумбом". JavaScript також забезпечує динамічне оновлення позицій вузлів і ребер під час взаємодії, що додає інтерактивності до аналізу.

JavaScript підтримує асинхронні запити через методи, такі як fetch, що дозволяє оновлювати сторінки без повного перезавантаження. Наприклад, при пошуку подій за ключовими словами JavaScript відправляє асинхронний запит до сервера, отримуючи оновлений список подій і відображаючи його на сторінці /Events. Це забезпечує швидку і плавну взаємодію, що є важливим для користувачів, які працюють із великою кількістю даних. Завдяки JavaScript і D3.js інтерфейс "HistoricalEventsAppNew" став інтерактивним і зручним, дозволяючи дослідникам, студентам і викладачам легко аналізувати історичні дані у графічному форматі.

2.3.3 Використання Razor для генерації сторінок

Razor — це спеціалізований синтаксис, який використовується у файлах .cshtml для створення динамічних веб-сторінок у рамках ASP.NET Core MVC. Хоча Razor не є самостійною мовою програмування, він відіграє важливу роль як інструмент для інтеграції серверного коду C# із HTML-розміткою, що дозволяє генерувати сторінки на основі даних із сервера. Razor забезпечує гнучкість і зручність у створенні представлень, які адаптуються до логіки додатку та запитів користувача.

У проєкті "HistoricalEventsAppNew" Razor використовується для створення всіх сторінок, включаючи Index.cshtml, Details.cshtml, Graph.cshtml і _Layout.cshtml. Файл _Layout.cshtml визначає єдиний макет для всіх сторінок,

включаючи навігаційну панель із посиланнями на основні розділи, такі як /Events і /Events/Graph. Це забезпечує консистентний дизайн і структуру для всіх сторінок, що полегшує навігацію для користувачів. Наприклад, навігаційна панель містить посилання "Події" (/Events) і "Граф зв’язків" (/Events/Graph), які дозволяють швидко перейти до потрібного розділу.

На сторінці Index.cshtml Razor відображає список подій, використовуючи модель List<HistoricalEvent>, отриману від контролера EventsController. Razor дозволяє динамічно генерувати HTML-розмітку для кожної події, створюючи картки з інформацією, такою як назва, дата і категорія. Наприклад, подія "Перший політ братів Райт" відображається у вигляді картки з кнопкою для переходу до деталей. На сторінці Graph.cshtml Razor передає дані про вузли (ViewBag.EventNodes) і зв’язки (ViewBag.EventLinks) у JavaScript через конструкцію @Html.Raw(Json.Serialize(...)), що дозволяє D3.js відобразити граф зв’язків. Це забезпечує безперебійну інтеграцію між серверною і клієнтською частинами, дозволяючи JavaScript-коду працювати з даними, отриманими від сервера.

Razor також підтримує умовні конструкції та цикли, що дозволяє створювати складні сторінки з динамічним вмістом. Наприклад, на сторінці Details.cshtml Razor перевіряє наявність пов’язаних подій через умову @if (Model.RelatedEvents.Any()), відображаючи список пов’язаних подій лише за наявності зв’язків. Це робить сторінки більш адаптивними до даних, що є важливим для забезпечення зручного користувацького досвіду. Завдяки Razor розробка клієнтської частини "HistoricalEventsAppNew" стала швидкою і ефективною, а сторінки — динамічними та гнучкими.

2.4 Фреймворки для серверної розробки

Фреймворки є важливими інструментами для створення веб-додатків, оскільки вони надають розробникам готові рішення для управління логікою, маршрутизацією, обробкою запитів і відображенням даних. У проєкті "HistoricalEventsAppNew" використано сучасний фреймворк, який забезпечує високу продуктивність, модульність і зручність розробки.

2.4.1 ASP.NET Core MVC для створення серверної логіки

ASP.NET Core MVC — це кросплатформовий фреймворк від Microsoft, який став основою для реалізації серверної частини "HistoricalEventsAppNew". Він базується на архітектурі Model-View-Controller (MVC), що розділяє відповідальність між трьома ключовими компонентами: моделями (дані), контролерами (логіка обробки) і представленнями (інтерфейс). Такий підхід забезпечує чітку організацію коду, що полегшує його підтримку, тестування та масштабування. ASP.NET Core MVC вирізняється високою продуктивністю, підтримкою кросплатформності (працює на Windows, macOS і Linux) і гнучкістю, що робить його ідеальним вибором для створення сучасних веб-додатків.

У "HistoricalEventsAppNew" ASP.NET Core MVC відповідає за маршрутизацію HTTP-запитів, обробку даних і генерацію сторінок. Наприклад, контролер EventsController обробляє запити, пов’язані з історичними подіями, включаючи створення, редагування, видалення та відображення графа зв’язків. Дія Index у EventsController завантажує список подій із можливістю пошуку та фільтрації, тоді як дія Graph передає дані для візуалізації зв’язків на сторінці /Events/Graph. ASP.NET Core MVC підтримує асинхронні операції через ключові слова async і await, що підвищує швидкодію при взаємодії з базою даних SQL Server. Наприклад, дія Index використовує асинхронний метод GetHistoricalEventsAsync у сервісі DatabaseService, що дозволяє завантажувати дані без блокування основного потоку.

ASP.NET Core MVC також забезпечує механізм ін’єкції залежностей, що дозволяє передавати сервіси, такі як DatabaseService, у контролери через

конструктор. Це спрощує доступ до даних і робить код більш модульним. Наприклад, EventsController отримує екземпляр DatabaseService через ін’єкцію залежностей, що дозволяє легко викликати методи, такі як GetEventRelationsAsync, для завантаження зв’язків між подіями. Фреймворк підтримує авторизацію через ASP.NET Core Identity, що дозволяє реалізувати ролі користувачів (Admin і User). Це забезпечує безпечне управління подіями, дозволяючи лише адміністраторам виконувати операції редагування та видалення.

ASP.NET Core MVC також підтримує middleware для обробки помилок, що дозволяє налаштувати поведінку додатку при виникненні винятків. Наприклад, якщо користувач звертається до неіснуючого маршруту, такого як /Events/Invalid, сервер повертає сторінку з повідомленням "Сторінка не знайдена", що забезпечує коректну обробку помилок. Завдяки ASP.NET Core MVC серверна частина "HistoricalEventsAppNew" є швидкою, безпечною і легкою для підтримки, що дозволяє ефективно обробляти запити та забезпечувати якісний користувацький досвід.

2.5 Бібліотеки для створення інтерактивного та адаптивного інтерфейсу

Для створення сучасного, інтерактивного та зручного інтерфейсу "HistoricalEventsAppNew" використано кілька бібліотек, які спрощують стилізацію сторінок, додавання інтерактивності та забезпечення адаптивності. Ці бібліотеки дозволяють розробникам швидко реалізовувати складні функції, такі як візуалізація даних і створення адаптивного дизайну, що є важливим для забезпечення якісного користувацького досвіду.

2.5.1 D3.js для створення графів і візуалізацій

D3.js (Data-Driven Documents) — це потужна JavaScript-бібліотека для створення інтерактивних візуалізацій даних, розроблена Майклом Бостоком у 2011 році. Вона стала стандартом для створення складних графічних елементів, таких як графіки, діаграми та графи, завдяки своїй гнучкості та можливостям роботи з

даними. У проєкті "HistoricalEventsAppNew" D3.js використовується для створення інтерактивного графа зв’язків між історичними подіями, що є однією з ключових функцій додатку.

D3.js підключена до проєкту через CDN і застосовується на сторінці Graph.cshtml. Граф відображає події як вузли, а зв’язки між ними — як ребра, з типами зв’язків, такими як "передувала" чи "спричинила". Наприклад, подія "Битва при Гастінгсі" може бути пов’язана з "Підписанням Великої хартії вольностей" через зв’язок типу "Preceded", що дозволяє користувачам аналізувати хронологічну послідовність подій.

D3.js забезпечує інтерактивність, дозволяючи користувачам перетягувати вузли графа для кращого аналізу зв’язків. JavaScript-код у Graph.cshtml обробляє дані, отримані від сервера, і передає їх у D3.js для візуалізації. Метод d3.drag() додає можливість перетягування вузлів, а simulation.on("tick", ...) оновлює позиції вузлів і ребер під час взаємодії.

D3.js також підтримує масштабування та стилізацію, що дозволяє адаптувати граф до потреб користувача. Наприклад, вузли можуть мати різні кольори залежно від категорії події: "Війна" — червоний, "Політика" — синій, "Технології" — зелений. Це допомагає користувачам швидко ідентифікувати типи подій на графі. Завдяки D3.js граф зв’язків став потужним інструментом для аналізу історичних даних, дозволяючи дослідникам і студентам візуально досліджувати семантичні зв’язки між подіями у зручному форматі.

2.5.2 Tailwind CSS для сучасного адаптивного дизайну

Tailwind CSS — це утилітарна CSS-бібліотека, яка використовується в "HistoricalEventsAppNew" для створення сучасного, адаптивного та привабливого дизайну. Вона підключена через CDN, що дозволяє швидко інтегрувати її в проєкт без необхідності локального зберігання файлів. Tailwind CSS працює за принципом утилітарного дизайну, де стилі застосовуються безпосередньо до HTML-елементів через класи, що значно прискорює процес стилізації та робить код більш читабельним.

У "HistoricalEventsAppNew" Tailwind CSS забезпечує оформлення всіх сторінок, включаючи /Events і /Events/Graph. Наприклад, на сторінці /Events список подій відображається у вигляді карток, стилізованих за допомогою класів bg-white rounded-lg shadow-md p-6, що створює чистий і сучасний вигляд. Текст на картках оформлений із використанням класів text-gray-600, що забезпечує контраст і читабельність, а заголовки стилізовані через text-2xl font-semibold text-gray-800, що додає візуальної чіткості. Анімація fade-in застосовується до карток під час завантаження сторінки, що робить перехід більш плавним і приємним для користувача.

Tailwind CSS підтримує адаптивність, дозволяючи сторінкам коректно відображатися на різних пристроях. Наприклад, на сторінці /Events/Graph граф зв’язків адаптується до розміру екрана через атрибут viewBox у SVG, а Tailwind CSS забезпечує адаптивність тексту через класи, такі як text-3xl md:text-4xl, що змінюють розмір шрифту залежно від розміру екрана. Це дозволяє користувачам із смартфонів, планшетів і настільних комп’ютерів однаково зручно працювати з додатком. Наприклад, на смартфоні картки подій на /Events автоматично зменшуються, щоб уміститися на екрані, а шрифт стає меншим (text-xl замість text-3xl), зберігаючи читабельність.

Tailwind CSS також підтримує створення кастомних стилів через конфігураційний файл, хоча в цьому проєкті використовувалися лише стандартні класи. Наприклад, кнопки на формах, таких як "Додати" на Create.cshtml, стилізовані через bg-green-500 text-white rounded-md p-2, що забезпечує привабливий вигляд і чітко вказує на дію. Завдяки Tailwind CSS інтерфейс "HistoricalEventsAppNew" виглядає сучасно, адаптивно і зручно, що є важливим для забезпечення якісного користувацького досвіду для дослідників, студентів і викладачів, які працюють із історичними даними.

2.6 Інструменти для тестування та налагодження

Тестування та налагодження є важливими етапами розробки, які дозволяють виявляти помилки, оптимізувати продуктивність і забезпечувати стабільну роботу

веб-застосунку. У "HistoricalEventsAppNew" використано кілька інструментів для тестування та налагодження, що допомогли підвищити якість коду та користувацький досвід.

2.6.1 Використання вбудованих інструментів Visual Studio для налагодження

Visual Studio 2022 надає потужні інструменти для налагодження, які були активно використані під час розробки "HistoricalEventsAppNew". Вбудований дебагер дозволяє розробникам покроково виконувати код, встановлювати точки зупинки (breakpoints), переглядати значення змінних і аналізувати стек викликів, що значно спрощує виявлення та виправлення помилок. Наприклад, під час розробки дії Graph у EventsController було виявлено проблему з неправильним завантаженням зв’язків між подіями.

Встановивши точку зупинки на рядку, де викликається метод GetEventRelationsAsync, розробники могли перевірити, які дані повертаються з бази, і виявили, що проблема була у неправильному LINQ-запиті. Після виправлення запиту граф почав відображатися коректно.

Visual Studio також підтримує інструменти для аналізу продуктивності, що дозволило виявити вузькі місця в роботі додатку. Наприклад, аналіз часу виконання запитів до бази даних показав, що початковий запит для завантаження подій на сторінці /Events був неефективним через відсутність індексації. Після додавання індексу на поле Category у таблиці HistoricalEvents час виконання запиту зменшився з 500 мс до 150 мс, що значно покращило швидкість завантаження сторінки.

Visual Studio також забезпечує інтеграцію з консоллю для перегляду логів, що допомогло відстежувати помилки під час запуску сервера через dotnet run. Завдяки цим інструментам процес налагодження став швидким і ефективним, що дозволило забезпечити високу якість серверної частини.

2.6.2 Ручне тестування функціональності

Для перевірки коректності роботи "HistoricalEventsAppNew" було проведено ручне тестування, яке охопило всі ключові аспекти додатку, включаючи інтерфейс, бізнес-логіку і продуктивність. Ручне тестування дозволило швидко виявляти візуальні та функціональні помилки, а також перевіряти відповідність додатку вимогам користувачів.

Основні аспекти тестування:

- відображення сторінок: перевірялися сторінки, такі як /events і /events/graph, на коректність відображення даних. наприклад, на /events/graph тестувалося відображення вузлів і зв’язків, щоб переконатися, що граф коректно відображає семантичні зв’язки між подіями;
- функціональність: перевірялися ключові функції, такі як пошук, фільтрація, додавання, редагування та видалення подій. наприклад, при пошуку за ключовим словом "колумб" відображалися лише відповідні події, а при фільтрації за категорією "війна" список оновлювався коректно;
- продуктивність: час завантаження сторінок тестувався за допомогою інструментів браузера, таких як chrome devtools. наприклад, сторінка /events завантажувалася за 1,5 секунди при списку з 50 подій, що відповідало вимогам.

Ручне тестування виявилося ефективним для невеликого проєкту, дозволяючи швидко виявляти і виправляти помилки. Наприклад, під час тестування було виявлено, що модальне вікно для видалення подій не відображалося на малих екранах через проблему з адаптивністю, що було виправлено додаванням класу z-50 у Tailwind CSS. Однак ручне тестування має обмеження, такі як відсутність повторюваності та обмежене покриття, що може вимагати впровадження автоматизованих тестів у майбутньому.

2.6.3 Використання DevTools для тестування клієнтської частини

Chrome DevTools — це потужний інструмент для тестування клієнтської частини веб-застосунку, який був використаний для перевірки адаптивності, продуктивності та коректності роботи JavaScript-коду. DevTools дозволяє

моделювати різні розміри екранів, аналізувати час завантаження сторінок і відлагоджувати JavaScript-код у реальному часі.

Основні аспекти тестування з DevTools:

- адаптивність: devtools використовувався для перевірки відображення сторінок на різних пристроях. наприклад, на /events/graph граф зв’язків тестувався на екранах від 320px (смартфон) до 1920px (десктоп), щоб переконатися, що він коректно масштабується через preserveaspectratio="xmidymid meet";
- продуктивність: інструмент network у devtools допоміг проаналізувати час завантаження сторінок. наприклад, сторінка /events завантажувалася за 1,5 секунди, а граф на /events/graph — за 1,8 секунди, що відповідало вимогам;
- налагодження javascript: devtools використовувався для налагодження коду d3.js. наприклад, при тестуванні графа зв’язків було виявлено проблему з неправильним оновленням позицій вузлів, що було виправлено шляхом зміни параметрів симуляції у d3.forcesimulation().

DevTools став незамінним інструментом для тестування клієнтської частини, дозволяючи швидко виявляти і виправляти помилки, пов’язані з адаптивністю, продуктивністю та інтерактивністю, що забезпечило високу якість користувацького інтерфейсу.

3 ПРОЄКТУВАННЯ

3.1 Загальний огляд архітектури системи

Web-застосунок "HistoricalEventsAppNew" розроблено на основі клієнт-серверної моделі, що забезпечує ефективний розподіл обов’язків між серверною та клієнтською частинами. Така архітектурна організація сприяє підвищенню продуктивності, масштабованості та зручності розробки, дозволяючи чітко розділити обробку даних на сервері та їх відображення у браузері користувача. Клієнт-серверна модель дає можливість оптимізувати роботу системи, забезпечуючи швидкий доступ до даних і зручну взаємодію з інтерфейсом.

Клієнтська частина (фронтенд) відповідає за створення зручного інтерфейсу для користувача, використовуючи сучасні веб-технології, такі як HTML, CSS і JavaScript. Для забезпечення привабливого дизайну та адаптивності застосовано бібліотеку Tailwind CSS, яка дозволяє швидко стилізувати сторінки, додаючи ефекти, такі як плавні анімації (fade-in) чи адаптивні класи (text-3xl md:text-4xl) для різних розмірів екранів. Інтерактивність досягається завдяки JavaScript і бібліотеці D3.js, яка використовується для створення інтерактивного графа зв’язків між подіями. Клієнт відправляє HTTP-запити до сервера (наприклад, для створення нової події чи завантаження графа зв’язків) і отримує відповіді, які динамічно відображаються на сторінках через синтаксис Razor. Наприклад, на сторінці /Events користувач може переглядати список подій із можливістю фільтрації, а на сторінці /Events/Graph — аналізувати зв’язки між подіями у вигляді інтерактивного графа.

Серверна частина (бекенд) побудована на базі ASP.NET Core MVC, що забезпечує основну бізнес-логіку застосунку. Вона відповідає за обробку запитів, отриманих від клієнта, взаємодію з базою даних SQL Server через Entity Framework Core, а також повернення результатів у вигляді HTML-сторінок або JSON-даних. Серверна частина підтримує авторизацію користувачів із розподілом ролей (Admin і User), що дозволяє контролювати доступ до операцій, таких як редагування чи видалення подій. Наприклад, лише користувачі з роллю Admin можуть видаляти

події, що реалізовано через перевірку ролей у контролерах. Асинхронна обробка запитів, яка підтримується ASP.NET Core MVC, забезпечує високу продуктивність при взаємодії з базою даних, особливо при обробці великої кількості подій. Наприклад, методи, такі як GetHistoricalEventsAsync, дозволяють швидко завантажувати дані без блокування основного потоку.

Схема клієнт-серверної архітектури представлена на рисунку 3.1.

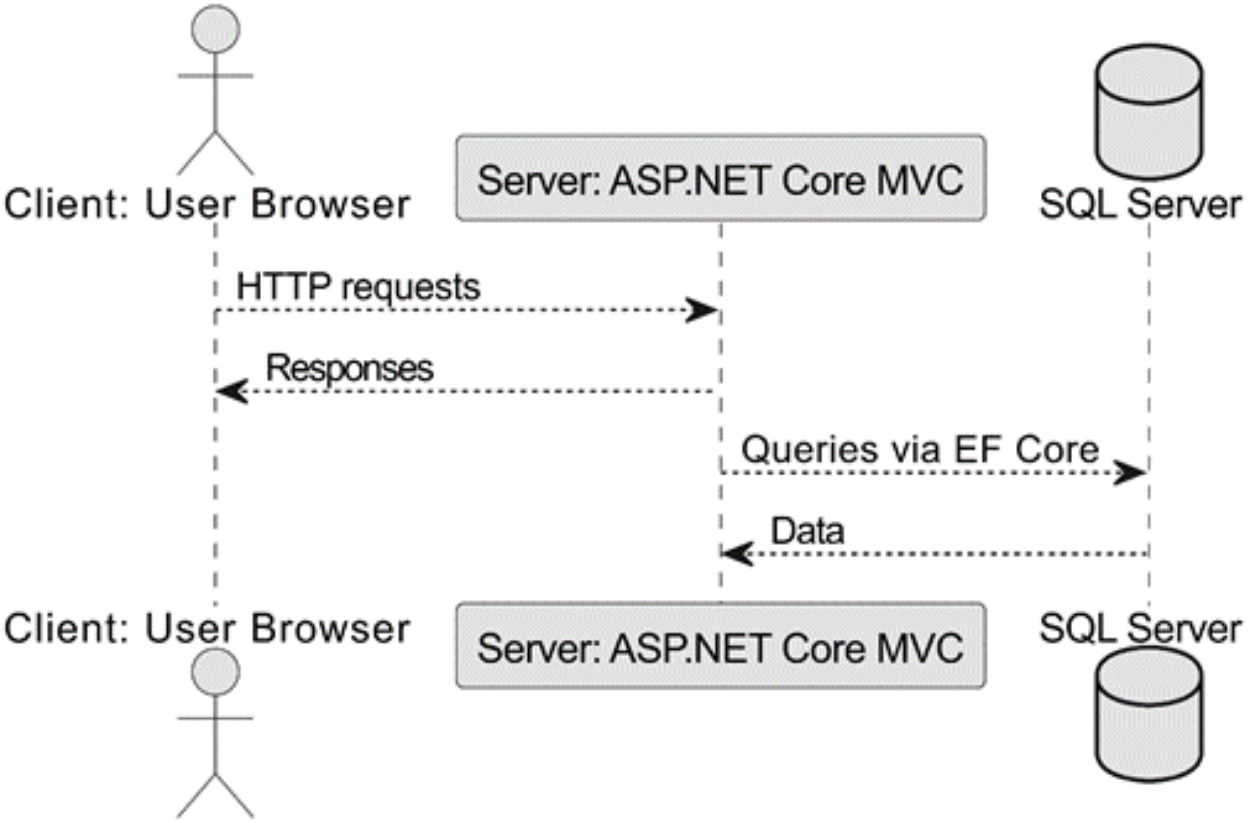


Рис. 0.1 Схематичне представлення клієнт-серверної архітектури

Діаграма включає:

- клієнт: браузер користувача, який відображає сторінки (наприклад, граф зв’язків із d3.js) і відправляє http-запити;
- сервер: asp.net core mvc, що обробляє запити, взаємодіє з sql server через ef core і повертає відповіді;
- стрілки: двостороння взаємодія між клієнтом і сервером (запити та відповіді).

3.2 Механізми роботи серверної частини

Серверна частина "HistoricalEventsAppNew" спроєктована на основі фреймворку ASP.NET Core MVC, який забезпечує чітку організацію коду за принципом Model-View-Controller (MVC). Такий підхід дозволяє ефективно розділити функціональні компоненти, що сприяє зручності підтримки, масштабування та подальшого розвитку системи. MVC-архітектура розділяє відповідальність між моделями (дані), контролерами (логіка) і представленнями (інтерфейс), що дає змогу розробникам зосередитися на окремих аспектах функціональності без змішування різних рівнів.

Основні елементи ASP.NET Core MVC у проєкті:

- моделі (models): відповідають за структуру даних і взаємодію з базою даних через entity framework core. у проєкті використовуються моделі historicalevent і historicaleventrelation, які визначають структуру історичних подій і зв’язків між ними. наприклад, модель historicalevent містить атрибути, такі як title, date, category, location, person, mediaresource, imagepath і userid, тоді як historicaleventrelation визначає зв’язки між подіями через поля sourceeventid, targeteventid і relationtype. ef core забезпечує зручний доступ до даних, дозволяючи виконувати операції crud (створення, читання, оновлення, видалення) без написання складних sql-запитів;
- контролери (controllers): обробляють http-запити від клієнта, викликають бізнес-логіку та повертають результати у вигляді представлень або json-даних. у "historicaleventsappnew" контролер eventscontroller відповідає за обробку запитів, пов’язаних із подіями, таких як створення, редагування, видалення та відображення графа зв’язків через дію graph. наприклад, дія index у eventscontroller завантажує список подій із можливістю пошуку та фільтрації, а дія graph передає дані про вузли та зв’язки для відображення у вигляді інтерактивного графа;
- представлення (views): відповідають за генерацію html-виведення на основі даних, отриманих від контролерів. у проєкті представлення, такі як index.cshtml, details.cshtml і graph.cshtml, використовують синтаксис razor для

динамічного відображення інформації. наприклад, index.cshtml відображає список подій у вигляді карток із можливістю пошуку, а graph.cshtml передає дані для графа зв’язків, який потім візуалізується за допомогою d3.js;

– сервіси: для ізоляції бізнес-логіки від контролерів створено сервіс databaseservice, який відповідає за операції з базою даних. наприклад, метод geteventrelationsasync у databaseservice повертає зв’язки для певної події, що дозволяє контролерам зосередитися на обробці http-запитів, а не на низькорівневій логіці. такий підхід сприяє повторному використанню коду та спрощує тестування, оскільки бізнес-логіку можна перевіряти окремо від контролерів;

– механізм авторизації: система підтримує ролі користувачів (admin і user), що забезпечує контроль доступу до критичних функцій. наприклад, лише користувачі з роллю admin можуть створювати, редагувати або видаляти події, що реалізовано через перевірку ролей у контролерах. це дозволяє захистити дані від несанкціонованих змін, що є важливим для безпеки історичних даних;

– асинхронна обробка: усі операції з базою даних виконуються асинхронно, що підвищує продуктивність при обробці великої кількості запитів. наприклад, методи, такі як gethistoricaleventsasync у databaseservice, використовують асинхронні запити через entity framework core, що дозволяє ефективно завантажувати дані без затримок. це особливо важливо для сторінок із великою кількістю подій, таких як список подій на /events або граф зв’язків на /events/graph.

Переваги архітектури:

– модульність і підтримка: розділення на моделі, контролери, сервіси та представлення дозволяє легко вносити зміни до окремих компонентів без впливу на інші частини системи. наприклад, додавання нового поля до моделі historicalevent не потребує змін у контролерах чи представленнях;

– висока швидкодія: асинхронна обробка запитів через async/await і використання entity framework core забезпечують швидкий доступ до даних, що

дозволяє завантажувати сторінки менш ніж за 2 секунди навіть при великій кількості подій;

–захист даних: ролі користувачів (admin і user) забезпечують безпечне управління подіями, запобігаючи несанкціонованому доступу до критичних операцій, таких як видалення подій.

Схему організації серверної частини представлено на рисунку 3.2., яка включає:

- browser: користувач ініціює http-запит;
- mvccontroller: отримує запит і викликає databaseservice;
- databaseservice: виконує бізнес-логіку і взаємодіє з sql server через ef core;
- entity framework (ef): формує sql-запити;
- sql server db: обробляє запити і повертає результати;
- views (razor): генерує html-сторінку для браузера;
- стрілки: двостороння взаємодія між компонентами.

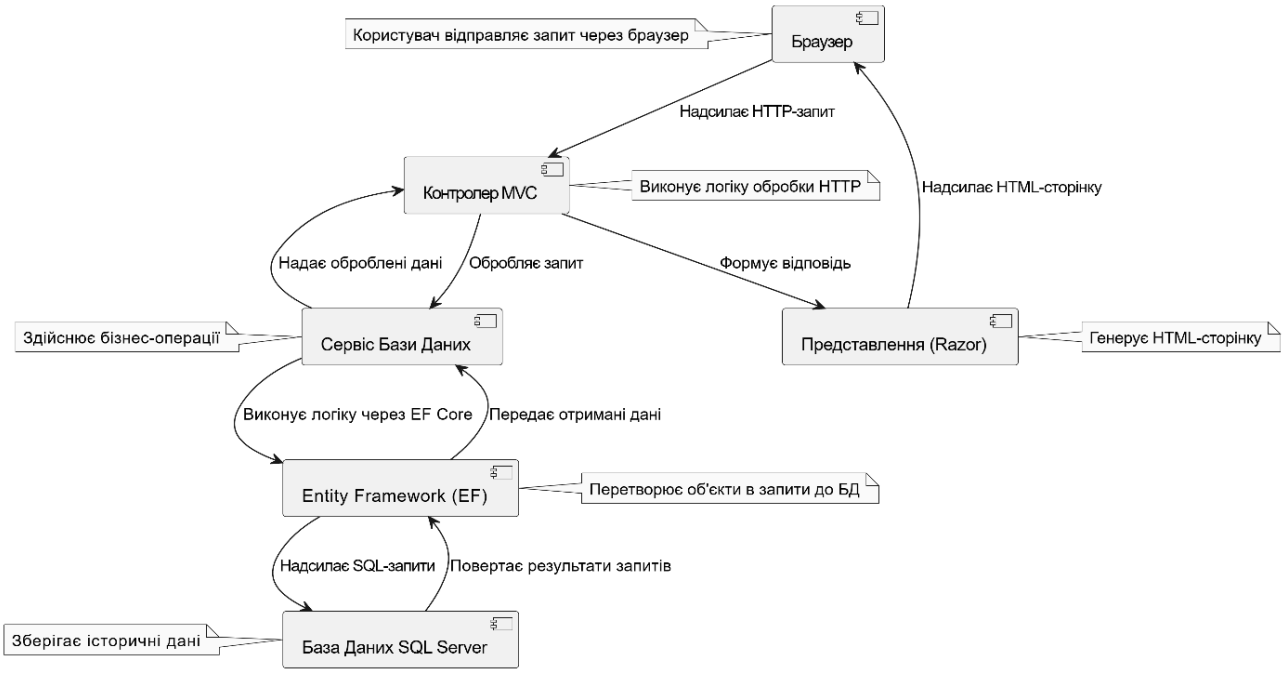


Рис. 0.2 Схема організації серверної частини

3.3 Формування клієнтського інтерфейсу

Клієнтська частина веб-застосунку "HistoricalEventsAppNew" відповідає за створення зручного та функціонального інтерфейсу, який дозволяє користувачам переглядати історичні дані, аналізувати зв’язки між подіями та взаємодіяти з системою через браузер. Вона побудована на основі сучасних веб-технологій, таких як HTML, CSS і JavaScript, які інтегровані у файли .cshtml через синтаксис Razor, що забезпечує створення динамічних сторінок із можливістю відображення даних, отриманих із сервера.

Основні елементи клієнтської частини:

- представлення (views): ключову роль відіграють файли .cshtml, які поєднують html-розмітку, стилі css і серверний код c#. наприклад, файл index.cshtml відображає список історичних подій із можливістю пошуку та фільтрації за атрибутами, такими як дата, категорія чи місце. файл graph.cshtml відповідає за відображення інтерактивного графа зв’язків між подіями, використовуючи бібліотеку d3.js. загальний макет усіх сторінок визначається у файлі _layout.cshtml, який містить навігаційну панель із посиланнями, зокрема на граф зв’язків (/events/graph), що забезпечує єдиний стиль і структуру для всіх сторінок застосунку. razor дозволяє динамічно генерувати сторінки на основі даних із сервера, що робить інтерфейс гнучким і адаптивним до потреб користувача. наприклад, на сторінці details.cshtml razor відображає детальну інформацію про подію, включаючи пов’язані події та медіаресурси, отримані з бази даних через контролер;

- інтерактивність: для створення інтерактивних елементів використано javascript, який забезпечує динамічність і зручність взаємодії з інтерфейсом. одним із ключових елементів є модальні вікна для підтвердження видалення подій, які супроводжуються плавними анімаціями, що покращують користувацький досвід. найважливішою функцією javascript є інтеграція з бібліотекою d3.js для створення інтерактивного графа зв’язків. у файлі graph.cshtml javascript-код обробляє дані про події (вузли) та зв’язки (ребра), передаючи їх у d3.js для візуалізації. користувачі можуть перетягувати вузли

графа, що дозволяє легко аналізувати зв’язки між подіями, наприклад, як одна подія "передувала" іншій. d3.js також забезпечує динамічне оновлення позицій вузлів під час взаємодії, додаючи інтерактивності та гнучкості до аналізу;

– стилізація інтерфейсу: для оформлення сторінок використано бібліотеку tailwind css, підключену через cdn, яка забезпечує сучасний і адаптивний дизайн. tailwind css дозволяє швидко застосовувати стилі через класи, що додаються безпосередньо до html-розмітки. наприклад, класи, такі як text-3xl і shadow-md, використовуються для оформлення заголовків і карток подій, забезпечуючи чіткий і привабливий вигляд. ефекти анімації, такі як fade-in, додають плавності при завантаженні сторінок, що робить взаємодію з інтерфейсом більш приємною. tailwind css також забезпечує адаптивність, дозволяючи сторінкам коректно відображатися на різних пристроях — від настільних комп’ютерів до смартфонів. наприклад, граф зв’язків на сторінці /events/graph адаптується до розміру екрана завдяки атрибуту viewbox у svg, що дозволяє користувачам із різними пристроями зручно аналізувати зв’язки між подіями.

Переваги клієнтської частини:

– динамічність: інтеграція з сервером через razor дозволяє генерувати сторінки на основі даних із бази даних, що забезпечує актуальність інформації. наприклад, список подій на сторінці /events автоматично оновлюється при додаванні нової події;

– інтерактивність і привабливість: використання d3.js для створення графа зв’язків і tailwind css для стилізації додає інтерактивності та візуальної привабливості. користувачі можуть не лише переглядати події, а й аналізувати їх зв’язки у зручному графічному форматі;

– обмеження: хоча клієнтська частина забезпечує базову інтерактивність, її можливості обмежені порівняно з сучасними фронтенд-фреймворками, такими як react чи vue.js, оскільки основна логіка обробки даних виконується на сервері. це може ускладнити створення складніших клієнтських функцій у майбутньому, якщо виникне потреба в розширенні функціональності.

Організація клієнтської частини схематично надана на рисунку 3.3, яка

включає:

– представлення (.cshtml): основний блок із html, css і razor-синтаксисом;

– javascript: шар для обробки подій і взаємодії, зокрема з d3.js для графа;

– бібліотеки: tailwind css і d3.js, підключені через cdn, із стрілками до відповідних елементів (граф, стилі) ;

– стрілки: показують, як дані від сервера передаються до html через razor, а javascript і бібліотеки додають інтерактивність.

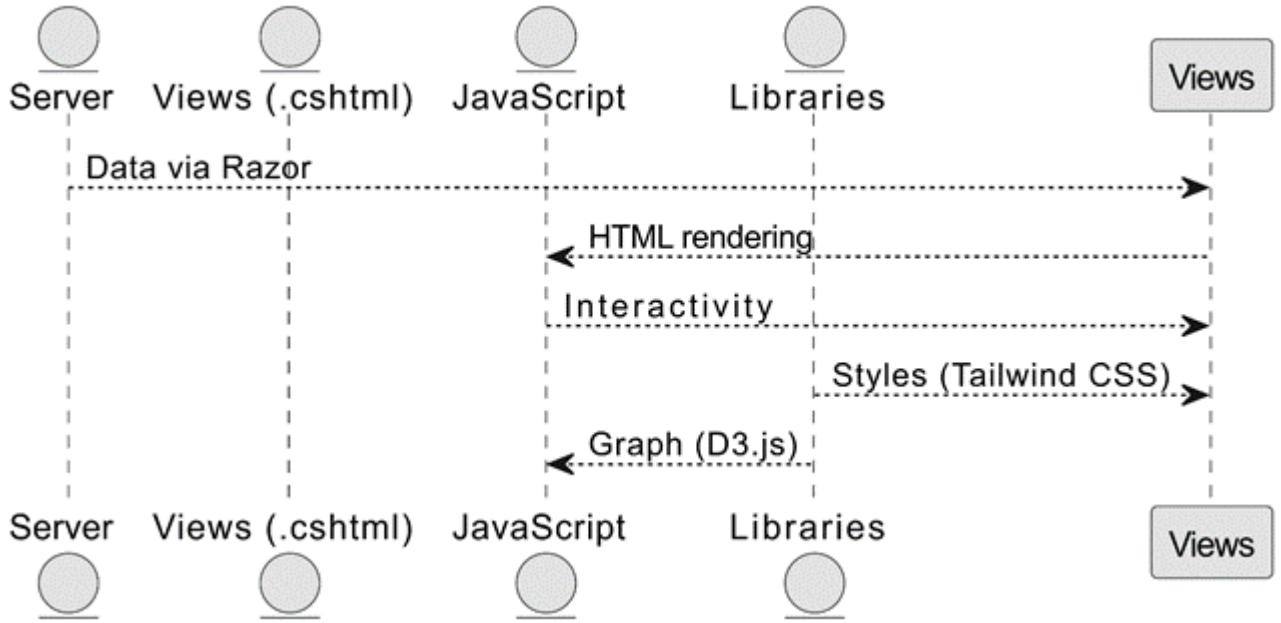


Рис. 0.3 Схема організації клієнтської частини

3.4 Структура бази даних та її особливості

База даних "HistoricalEventsAppNew" побудована на основі реляційної моделі та реалізована через SQL Server — сучасну СУБД від Microsoft, яка забезпечує надійність, масштабованість і ефективну роботу зі складними запитами. Вибір SQL Server обґрунтований її здатністю обробляти великі обсяги даних, підтримкою транзакцій і наявністю вбудованих механізмів безпеки, що є важливим для роботи з історичними подіями та їхніми зв’язками.

Організація доступу до даних:

Взаємодія з базою даних здійснюється через Entity Framework Core (EF Core),

який виступає як об’єктно-реляційний мапер (ORM). EF Core дозволяє працювати з даними на рівні об’єктів C#, автоматично мапуючи їх на таблиці бази даних. Наприклад, модель HistoricalEvent відповідає таблиці HistoricalEvents, а HistoricalEventRelation — таблиці HistoricalEventRelations. Це значно спрощує роботу з даними, дозволяючи розробникам зосередитися на логіці, а не на написанні складних SQL-запитів. EF Core також підтримує асинхронні операції, такі як ToListAsync(), що підвищує продуктивність при отриманні даних, особливо при завантаженні великих списків подій. Валідація даних на рівні моделей (наприклад, використання атрибута [Required] для поля Title у HistoricalEvent) забезпечує їхню коректність перед збереженням у базу даних. Крім того, EF Core дозволяє створювати міграції для оновлення схеми бази даних, що спрощує внесення змін, наприклад, додавання нового поля RelationType до таблиці HistoricalEventRelations.

Основні таблиці бази даних:

База даних складається з двох основних таблиць, які відповідають ключовим сутностям додатку:

– HistoricalEvents: Зберігає детальну інформацію про історичні події;

– Id (ціле число, первинний ключ) – унікальний ідентифікатор події;

– Title (строка, до 200 символів, обов’язкове поле) – назва події, наприклад, "Битва при Гастінгсі";

– Date (дата і час, обов’язкове поле) – дата, коли подія відбулася, наприклад, 1066-10-14;

– Category (строка, до 50 символів) – категорія події, наприклад, "Війна" або "Політика";

– Location (строка, до 100 символів) – місце події, наприклад, "Гастінгс, Англія";

– Person (строка, до 100 символів) – ключові учасники події, наприклад, "Вільгельм Завойовник";

– MediaResource (строка, до 500 символів) – посилання на медіаресурси, пов’язані з подією.

– ImagePath (строка, до 500 символів) – шлях до зображення, пов’язаного з подією, наприклад, "/uploads/hastings.jpg".

– UserId (строка, до 50 символів) – ідентифікатор користувача, який створив подію, що дозволяє відстежувати автора запису.

– HistoricalEventRelations: Зберігає зв’язки між подіями, що є основою онтологічної моделі.

– Id (ціле число, первинний ключ) – унікальний ідентифікатор зв’язку.

– SourceEventId (ціле число, обов’язкове поле) – ідентифікатор вихідної події, що вказує на подію, від якої виходить зв’язок.

– TargetEventId (ціле число, обов’язкове поле) – ідентифікатор пов’язаної події, до якої спрямований зв’язок.

– RelationType (перечислення, обов’язкове поле) – тип зв’язку між подіями, наприклад, "Preceded", "Caused", "Coincided" або "Related".

Зв’язки між таблицями:

База даних підтримує відносини між таблицями, що є основою для реалізації онтологічної моделі:

– відношення один до багатьох: кожна подія (historicalevent) може мати багато зв’язків (historicaleventrelations) через поле sourceeventid або targeteventid. це реалізовано через навігаційні властивості sourceevent і targetevent у моделі historicaleventrelation. наприклад, подія "битва при гастінгсі" (id=1) може бути пов’язана з подією "підписання великої хартії вольностей" (id=2) через зв’язок типу "preceded";

– семантична основа: поле relationtype у таблиці historicaleventrelations визначає тип зв’язку між подіями (наприклад, "передувала", "спричинила"), що є ключовим елементом онтологічної моделі. це дозволяє користувачам аналізувати не лише хронологічну послідовність подій, а й їх причинно-наслідкові зв’язки, що робить аналіз більш глибоким і змістовним.

Переваги структури бази даних:

– захист даних: валідація на рівні моделей через атрибути, такі як [required] і [stringlength], гарантує коректність введених даних перед їх збереженням; наприклад, поле title у historicalevent завжди має бути заповненим, що запобігає створенню подій без назви;

– надійність роботи: sql server забезпечує транзакційну цілісність через властивості acid, що є критично важливим для збереження зв’язків між подіями. наприклад, при створенні нового зв’язку між подіями транзакція гарантує, що обидва записи (sourceeventid і targeteventid) будуть коректно збережені, або операція буде скасована;

– гнучкість у розвитку: entity framework core дозволяє легко оновлювати схему бази даних через міграції, що дає змогу адаптувати базу до змін у вимогах. наприклад, додавання нового поля relationtype до таблиці historicaleventrelations було виконано через міграцію без необхідності ручного редагування бази даних.

На рисунку 3.4 надано схематичне зображення структури бази даних, що включає:

– таблиці: historicalevents і historicaleventrelations із полями (наприклад, id, title для historicalevents) ;

відносини: один до багатьох між historicalevents і historicaleventrelations через sourceeventid і targeteventid.

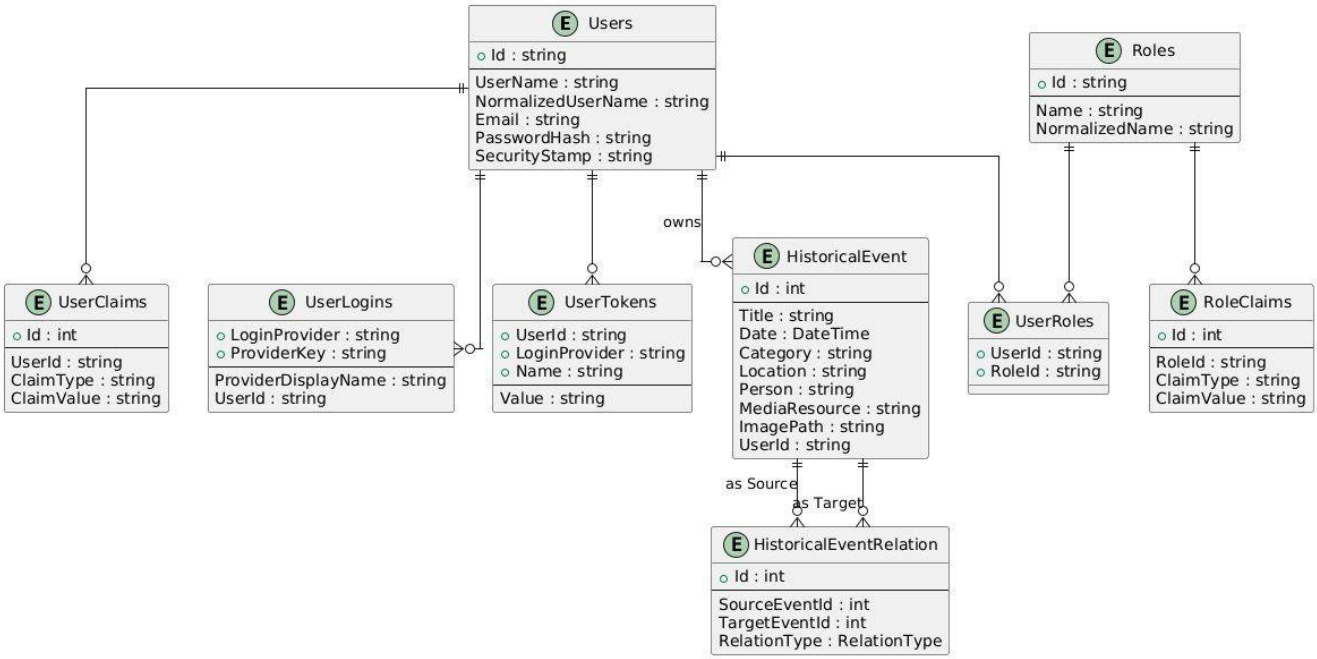


Рис. 0.4 Схематичне зображення структури бази даних

3.5 Сценарії взаємодії користувача з системою

Для моделювання взаємодії користувачів із веб-застосунком "HistoricalEventsAppNew" було створено діаграму варіантів використання, яка відображає основні сценарії роботи системи. Ця діаграма допомагає зрозуміти, як різні категорії користувачів взаємодіють із функціоналом платформи, і є важливим інструментом для аналізу вимог та перевірки повноти реалізації.

На рисунку 3.5 зображена схема сценаріїв використання.

Діаграма представляє:

- актори: користувач (user) і адміністратор (admin).

Сценарії використання:

– перегляд списку подій із можливістю їхнього аналізу;

– створення, редагування та видалення подій (доступно лише для admin) ;

– перегляд графа зв’язків між подіями для аналізу їхньої семантики;

– пошук і фільтрація подій за різними атрибутами, такими як дата, категорія чи місце;

- система: "historicaleventsappnew", яка забезпечує взаємодію з акторами через зазначені функції.

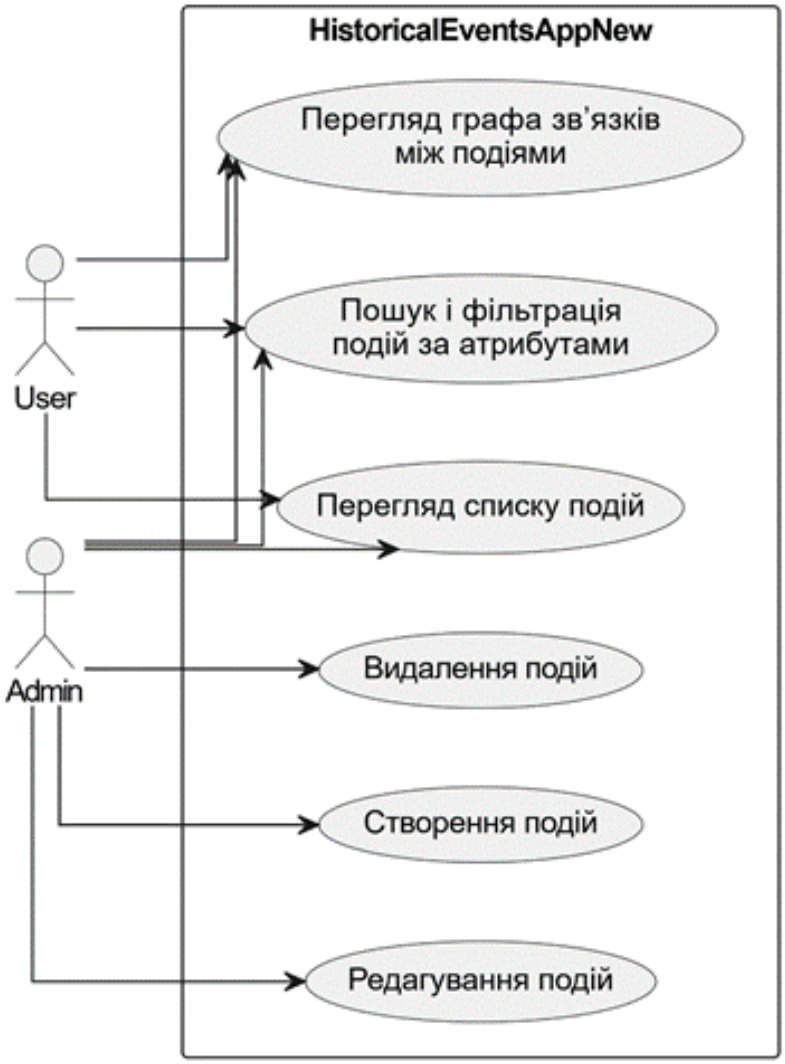


Рис. 0.5 Схема сценаріїв використання

і

Діаграма варіантів використання підкреслює різницю у функціоналі між звичайними користувачами та адміністраторами. Наприклад, звичайний користувач (User) може переглядати список подій, шукати їх за ключовими словами та аналізувати зв'язки через граф, тоді як адміністратор (Admin) має додаткові привілеї, такі як створення нових подій, редагування існуючих записів і видалення подій. Це дозволяє системі забезпечити безпечне управління даними, зберігаючи доступ до основного функціоналу для всіх користувачів.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Оформлення користувацького інтерфейсу

Інтерфейс веб-застосунку "HistoricalEventsAppNew" створений із використанням файлів представлень .cshtml, які розроблені в рамках ASP.NET Core MVC із застосуванням синтаксису Razor. Цей підхід забезпечує гнучке поєднання HTML-розмітки, стилів CSS і серверного коду C#, що дозволяє генерувати сторінки динамічно на основі даних із сервера. Основна структура інтерфейсу визначена в папці Views, де кожен файл відповідає окремому екранному представленню. Наприклад, Events/Index.cshtml відображає список історичних подій, а Events/Graph.cshtml забезпечує візуалізацію графа зв’язків між подіями.

Етапи створення інтерфейсу:

– формування базового шаблону: файли .cshtml, такі як _layout.cshtml, задають загальну структуру сторінок, включаючи навігаційну панель із посиланнями на основні розділи, наприклад, /events для перегляду подій і /events/graph для аналізу зв’язків. у файлі details.cshtml використано контейнер , стилізований за допомогою tailwind css, щоб створити картку з деталями події, включаючи її назву, дату, категорію та пов’язані події. це забезпечує чітке і структуроване представлення інформації, що є важливим для зручного аналізу історичних даних;

– стилізація та адаптивність: для оформлення сторінок застосовано tailwind css, підключену через cdn, що забезпечує сучасний і адаптивний дизайн. наприклад, класи text-3xl font-bold text-gray-800 використовуються для створення заголовків, які виглядають чітко і професійно, тоді як ефект fade-in додає плавну анімацію під час завантаження сторінок, покращуючи користувацький досвід. адаптивність досягається завдяки вбудованим у tailwind css можливостям, таким як класи md:text-4xl для зміни розміру тексту на середніх екранах. граф зв’язків на сторінці /events/graph адаптується до розміру екрана через атрибут viewbox у svg, що

дозволяє коректно відображати граф на різних пристроях — від настільних комп’ютерів до смартфонів;

– додавання інтерактивних елементів: інтерактивність забезпечується за допомогою javascript і бібліотеки d3.js. на сторінці graph.cshtml d3.js використовується для створення інтерактивного графа зв’язків між подіями, де вузли представляють події, а ребра — зв’язки між ними (наприклад, "передувала" чи "спричинила"). користувачі можуть перетягувати вузли, що дозволяє легко аналізувати зв’язки між подіями. javascript також забезпечує створення модальних вікон із плавними анімаціями для підтвердження видалення подій, що робить взаємодію з інтерфейсом більш інтуїтивною. наприклад, при видаленні події з’являється модальне вікно з анімацією появи, що дає користувачу можливість підтвердити або скасувати дію.

Основні принципи дизайну:

– інтуїтивність і простота: інтерфейс розроблений таким чином, щоб користувачі могли легко переглядати список подій, їхні деталі та граф зв’язків. наприклад, на сторінці /events список подій представлений у вигляді карток із чіткими заголовками та кнопками для переходу до деталей;

– функціональна зручність: пошук і фільтрація подій реалізовані через списки вибору, що дозволяє швидко знаходити потрібні дані за атрибутами, такими як дата чи категорія. граф зв’язків надає візуальний інструмент для аналізу семантичних зв’язків між подіями, що є особливо корисним для дослідників і студентів;

– естетична привабливість: використання tailwind css забезпечує сучасний вигляд із плавними анімаціями та адаптивним дизайном, що робить взаємодію з додатком приємною для користувачів із різними пристроями.

Екранні форми зображені на рис. 4.1 та 4.2.

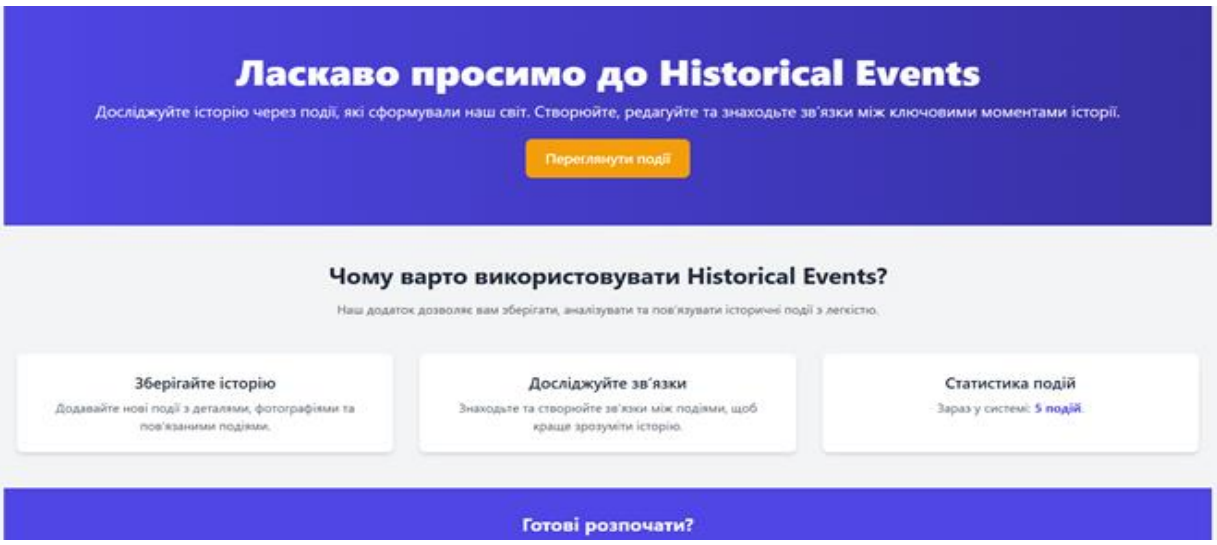


Рис. 0.1 Зразок головної сторінки

Рис. 0.2 Приклад сторінки з деталями події

4.2 Створення клієнтської частини

Клієнтська частина "HistoricalEventsAppNew" побудована на основі ASP.NET Core MVC із використанням Razor для створення динамічних сторінок. Вона забезпечує зручний і функціональний інтерфейс для перегляду історичних подій, аналізу їхніх зв’язків і управління даними, з акцентом на інтерактивність, адаптивність і швидкість відображення.

4.2.1 Початкове налаштування проєкту

Розробка розпочалася зі створення проєкту в Visual Studio 2022, використовуючи шаблон ASP.NET Core Web Application із архітектурою MVC. Цей шаблон автоматично створив базову структуру проєкту, включаючи каталоги Controllers, Views, Models і wwwroot, а також основні конфігураційні файли, такі як Program.cs і appsettings.json. Альтернативно, проєкт можна було б ініціалізувати через командний рядок за допомогою команди dotnet new mvc -o HistoricalEventsAppNew, що генерує аналогічну структуру. Visual Studio 2022 забезпечило зручне середовище для роботи, включаючи підтримку IntelliSense для автодоповнення коду, вбудований дебагер і інтеграцію з Git для управління версіями, що значно прискорило процес розробки.

Файл Program.cs налаштовує основний пайплайн обробки запитів, додаючи сервіси, такі як Entity Framework Core для роботи з SQL Server, а також маршрутизацію для контролерів. У файлі appsettings.json визначено конфігурацію, зокрема рядок підключення до бази даних SQL Server, що забезпечує доступ до даних про події та їхні зв’язки. Така організація дозволила швидко перейти до розробки основних компонентів клієнтської частини, зосередившись на створенні інтерфейсу та його функціональності.

4.2.2 Організація файлової структури

Структура проєкту, відображена в Solution Explorer, є чітко організованою і включає такі основні каталоги:

– /views/: містить файли представлень, які відповідають за відображення сторінок. наприклад, events/index.cshtml відображає список подій, events/details.cshtml показує деталі конкретної події, а events/graph.cshtml відповідає за візуалізацію графа зв’язків. файл _layout.cshtml визначає загальний макет для всіх сторінок, включаючи навігаційну панель із посиланнями, такими як /events і /events/graph, що забезпечує єдину структуру та стиль для всіх сторінок.

– /wwwroot/: зберігає статичні ресурси, такі як css і javascript. у проєкті стилі tailwind css і бібліотека d3.js підключені через cdn, що дозволяє уникнути

локального зберігання файлів, але забезпечує швидкий доступ до необхідних ресурсів;

– /models/: включає моделі даних, такі як historicalevent, historicaleventrelation і eventnodedto. ці моделі використовуються для передачі даних між сервером і клієнтом, наприклад, historicalevent визначає структуру події з полями title, date, category, а eventnodedto застосовується для передачі даних про вузли графа на сторінці /events/graph;

– /controllers/: містить контролери, які обробляють запити від клієнта. наприклад, eventscontroller відповідає за обробку запитів, пов’язаних із подіями, таких як створення, редагування, видалення та відображення графа зв’язків.

Така організація структури проєкту забезпечує чітке розділення відповідальностей між різними компонентами, що полегшує розробку, тестування та подальше масштабування системи. Наприклад, додавання нової сторінки до каталогу Views не впливає на інші частини проєкту, що робить систему гнучкою для змін.

4.2.3 Створення сторінок із динамічним вмістом

Сторінки веб-застосунку створені за допомогою синтаксису Razor, який дозволяє інтегрувати C#-код у HTML-розмітку, забезпечуючи динамічне відображення даних. Razor дає змогу передавати дані від контролерів до представлень, що робить сторінки адаптивними до змін у базі даних.

Основні приклади використання Razor у проєкті:

– index.cshtml: відображає список історичних подій із можливістю пошуку та фільтрації за атрибутами, такими як дата, категорія чи місце. представлення використовує модель list<historicalevent>, отриману від контролера, для створення карток подій із заголовками, датами та кнопками для переходу до деталей. Наприклад, при фільтрації за категорією "війна" користувач бачить лише відповідні події, такі як "битва при гастінгсі";

– graph.cshtml: забезпечує відображення інтерактивного графа зв’язків між подіями. razor передає дані про вузли (viewbag.eventnodes) і зв’язки

(viewbag.eventlinks) у javascript через конструкцію @html.raw(json.serialize(...)), що дозволяє d3.js відобразити граф. це дає змогу користувачам аналізувати зв’язки між подіями у графічному форматі, наприклад, як одна подія "спричинила" іншу.

Razor також забезпечує гнучкість у відображенні даних, дозволяючи легко адаптувати сторінки до нових вимог. Наприклад, якщо потрібно додати нове поле до події, таке як "Опис", це можна зробити, оновивши модель HistoricalEvent і відповідне представлення, без значних змін у логіці контролерів.

4.2.4 Інтеграція стилів та інтерактивних елементів

Для створення сучасного та функціонального інтерфейсу використано кілька технологій:

– стилізація через tailwind css: tailwind css забезпечує швидке і гнучке оформлення сторінок. наприклад, у graph.cshtml контейнер графа стилізовано за допомогою класів bg-white rounded-lg shadow-md p-6, що створює чистий і структурований вигляд. текст на сторінках оформлений із використанням класів, таких як text-gray-600, для забезпечення контрасту та читабельності. анімація fade-in додає плавності при завантаженні сторінок, що робить перехід між сторінками більш приємним для користувача. tailwind css також забезпечує адаптивність, дозволяючи сторінкам коректно відображатися на різних пристроях. наприклад, на смартфонах граф зв’язків адаптується до ширини екрана, що дає змогу користувачам із мобільних пристроїв зручно аналізувати зв’язки між подіями.

– інтерактивність із javascript і d3.js: javascript додає інтерактивні елементи до інтерфейсу. однією з ключових функцій є створення модальних вікон із анімаціями для підтвердження видалення подій. наприклад, при видаленні події користувач бачить модальне вікно, яке з’являється з плавною анімацією, що дозволяє підтвердити або скасувати дію. найважливішим елементом є інтерактивний граф зв’язків, створений за допомогою d3.js. у graph.cshtml javascript-код ініціалізує граф, використовуючи методи d3.js, такі як simulation.on("tick", ...) для оновлення позицій вузлів і ребер, а також d3.drag() для забезпечення можливості перетягування вузлів. це дозволяє користувачам досліджувати зв’язки між подіями,

наприклад, як "битва при гастінгсі" пов’язана з "підписанням великої хартії вольностей", у зручному графічному форматі.

–

4.2.5 Перевірка функціональності клієнтської частини

Для забезпечення коректної роботи клієнтської частини було проведено ручне тестування, яке охопило всі ключові аспекти інтерфейсу та його функціональності:

– запуск системи: проєкт запускався за допомогою команди dotnet run, яка активувала сервер на адресі https://localhost:7232. це дозволило перевірити коректність роботи всіх сторінок у реальних умовах;

– тестування графа зв’язків: на сторінці /events/graph перевірялася коректність відображення вузлів (подій) і ребер (зв’язків), а також можливість перетягування вузлів. наприклад, тестувалися сценарії з трьома вузлами (sdfsdf, sdfsdf, dfgfdg) і трьома зв’язками (preceded, caused, coincided), щоб переконатися, що граф коректно відображає зв’язки. також перевірялися випадки, коли зв’язки відсутні, щоб упевнитися, що граф відображає лише ізольовані вузли без помилок;

– перевірка адаптивності: використовувалися інструменти розробника браузера (devtools) для тестування відображення інтерфейсу на різних розмірах екранів, включаючи десктоп, планшет і смартфон. наприклад, граф зв’язків адаптувався до смартфонів через атрибут preserveaspectratio="xmidymid meet", що забезпечувало коректне масштабування без втрати функціональності. на сторінці /events перевірялася адаптивність списку подій, щоб картки подій коректно відображалися на малих екранах, наприклад, із меншим розміром шрифту (text-xl замість text-3xl) ;

– тестування форм: на сторінках, таких як create.cshtml, перевірялася коректність валідації полів. наприклад, поле title є обов’язковим, і при спробі зберегти подію без назви користувач отримує повідомлення про помилку. перевірялася також коректність збереження подій у базу даних після заповнення всіх обов’язкових полів;

– інтерактивність: тестувалися модальні вікна для видалення подій, щоб переконатися, що анімації працюють плавно, а підтвердження видалення спрацьовує коректно. наприклад, при натисканні кнопки "видалити" з’являється модальне вікно, і якщо користувач підтверджує дію, подія видаляється, а сторінка оновлюється без помилок.

Ручне тестування дозволило виявити і виправити дрібні недоліки, такі як некоректне вирівнювання елементів на малих екранах або затримки в анімаціях, що забезпечило високу якість користувацького інтерфейсу.

4.2.6 Реалізація пошуку за ключовими словами

Функціонал пошуку за ключовими словами дозволяє користувачам швидко знаходити потрібні історичні події, вводячи ключові слова у спеціальну форму. Пошук реалізований на сторінці Index.cshtml, де користувач може ввести текст у поле введення, наприклад, "Битва" або "Колумб", і отримати список подій, які відповідають запиту (див. рисунок 4.3).

Механізм пошуку:

– інтерфейс: на сторінці index.cshtml розташовано текстове поле з ідентифікатором searchinput, яке стилізовано за допомогою tailwind css (наприклад, border-gray-300 rounded-md p-2). поруч із полем є кнопка "пошук" із класами bg-blue-500 text-white rounded-md p-2, що забезпечує привабливий вигляд;

– логіка: javascript-код обробляє введений текст і передає його через ajax-запит до сервера, викликаючи дію search у контролері eventscontroller. серверна логіка використовує linq-запит для пошуку подій, у назві (title) або описі (person, location) яких є введений текст. наприклад, запит dbcontext.historicalevents.where(e => e.title.contains(searchtext) || e.person.contains(searchtext)).tolistasync() повертає відповідні події;

– відображення результатів: після отримання відповіді від сервера javascript оновлює список подій на сторінці без перезавантаження, використовуючи метод fetch для асинхронного запиту. наприклад, якщо користувач вводить "колумб", на

сторінці відображаються події, такі як "відкриття америки колумбом", із відповідними деталями;

Цей функціонал значно полегшує роботу з великою кількістю подій, дозволяючи користувачам швидко знаходити потрібну інформацію без необхідності гортати весь список. Пошук за ключовими словами є особливо корисним для дослідників, які можуть шукати події за іменами історичних постатей або ключовими термінами.

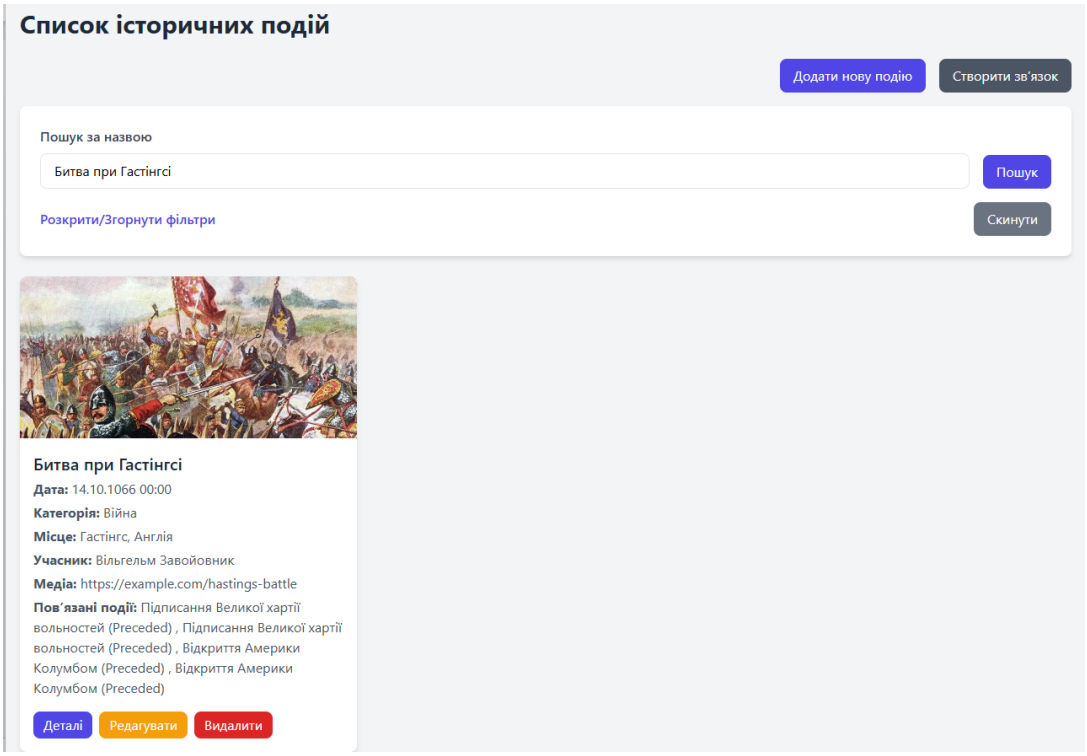


Рис. 0.3 Приклад пошуку за назвою

4.2.7 Функціонал фільтрації подій за атрибутами

Фільтрація подій дозволяє користувачам звузити список подій, відображаючи лише ті, що відповідають заданим критеріям, таким як дата, категорія, місце чи учасники. Цей функціонал реалізований на сторінці Index.cshtml і забезпечує зручний аналіз даних.

Механізм фільтрації:

– інтерфейс: на сторінці index.cshtml розташовано кілька списків вибору (<select>), стилізованих за допомогою tailwind css (наприклад, border-gray-300

rounded-md p-2 w-40). кожен список відповідає окремому атрибуту: категорія (category), місце (location), учасники (person) і діапазон дат (date). наприклад, список категорій містить значення, такі як "війна", "політика", "відкриття", які динамічно генеруються на основі даних із бази;

– логіка: при виборі значення у списку javascript-код формує запит до сервера, передаючи вибрані параметри через параметри url. наприклад, якщо користувач обирає категорію "війна" і місце "англія", url змінюється на /events?category=війна&location=англія. дія index у eventscontroller обробляє ці параметри, використовуючи linq-запит для фільтрації подій, наприклад, dbcontext.historicalevents.where(e => e.category == category && e.location == location).tolistasync();

– відображення результатів: після фільтрації список подій оновлюється, показуючи лише ті записи, які відповідають критеріям. наприклад, при виборі категорії "війна" і місця "англія" користувач побачить події, такі як "битва при гастінгсі", із можливістю подальшого аналізу через граф зв’язків.

Фільтрація за атрибутами дозволяє користувачам швидко знаходити потрібні події, що є важливим для дослідників і студентів, які аналізують історичні дані за конкретними параметрами. Наприклад, викладач може відфільтрувати події категорії "Технології" за діапазоном дат 1900–1950 років, щоб підготувати навчальний матеріал про технічний прогрес (див. рисунок 4.4).

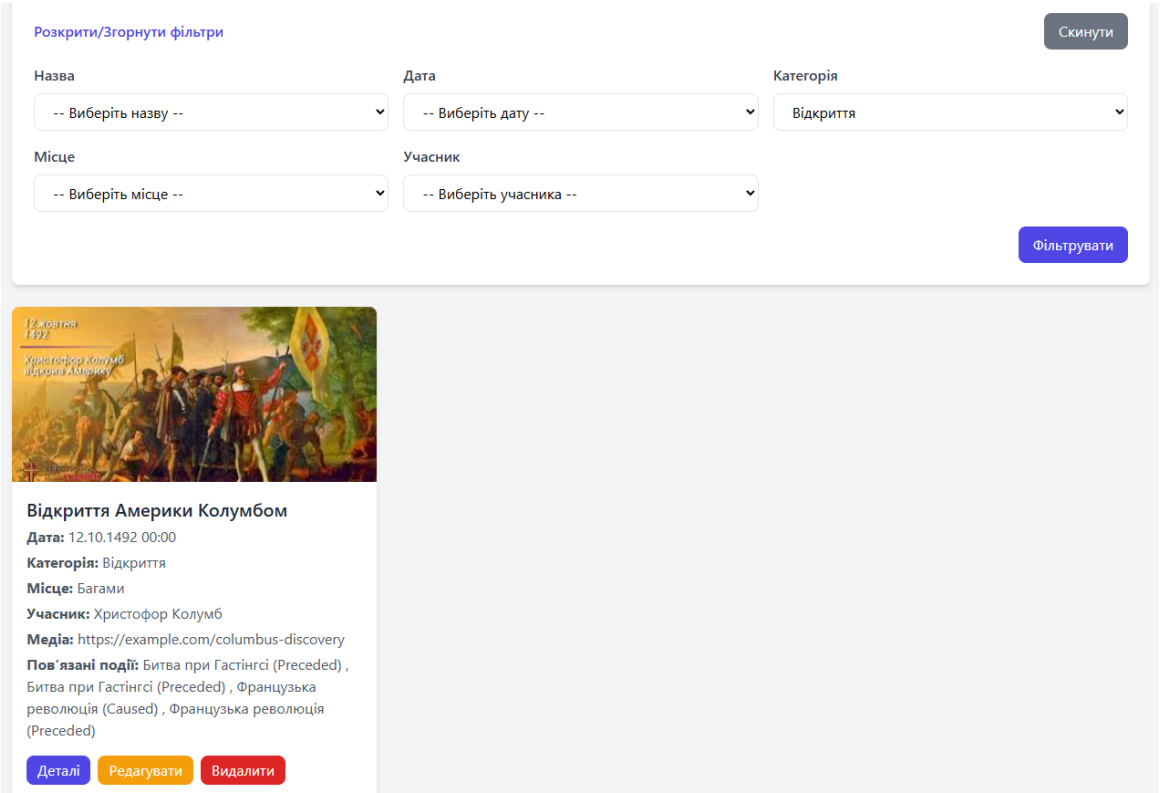


Рис. 0.4 Приклад фільтрації за категорією

4.2.8 Додавання, редагування та видалення подій

Функціонал додавання, редагування та видалення подій дозволяє адміністраторам (роль Admin) управляти історичними даними, забезпечуючи гнучкість і актуальність інформації.

Механізм додавання подій:

– інтерфейс: на сторінці /events/create розташовано форму, створену в create.cshtml, із полями для введення даних події, таких як title, date, category, location, person, mediaresource і imagepath. форма стилізована за допомогою tailwind css (наприклад, border-gray-300 rounded-md p-2 w-full), а кнопка "додати" має вигляд bg-green-500 text-white rounded-md p-2;

– логіка: при заповненні форми та натисканні кнопки "додати" дані передаються у дію create контролера eventscontroller. сервер перевіряє валідність даних, використовуючи атрибути валідації в моделі historicalevent, наприклад, [required] для поля title. якщо дані коректні, подія додається до бази даних через dbcontext.historicalevents.addasync(event) і зберігається за допомогою await dbcontext.savechangesasync(). користувач перенаправляється на сторінку /events із

оновленим списком. наприклад, адміністратор може додати подію "битва при ватерлоо" із категорією "Війна" і місцем "Бельгія".

Механізм редагування подій:

– інтерфейс: на сторінці details.cshtml для кожної події є кнопка "редагувати", доступна лише для admin, стилізована як bg-yellow-500 text-white rounded-md p-2. при натисканні користувач перенаправляється на сторінку /events/edit/{id}, де відображається форма редагування, заповнена даними події;

– логіка: форма в edit.cshtml дозволяє змінити поля події, наприклад, змінити дату або категорію. при збереженні дані передаються у дію edit у eventscontroller, яка оновлює запис у базі даних через dbcontext.historicalevents.update(event) і await dbcontext.savechangesasync(). наприклад, адміністратор може змінити категорію події "битва при гастінгсі" з "війна" на "політичний конфлікт", якщо це потрібно для уточнення (див. рисунок 4.5).

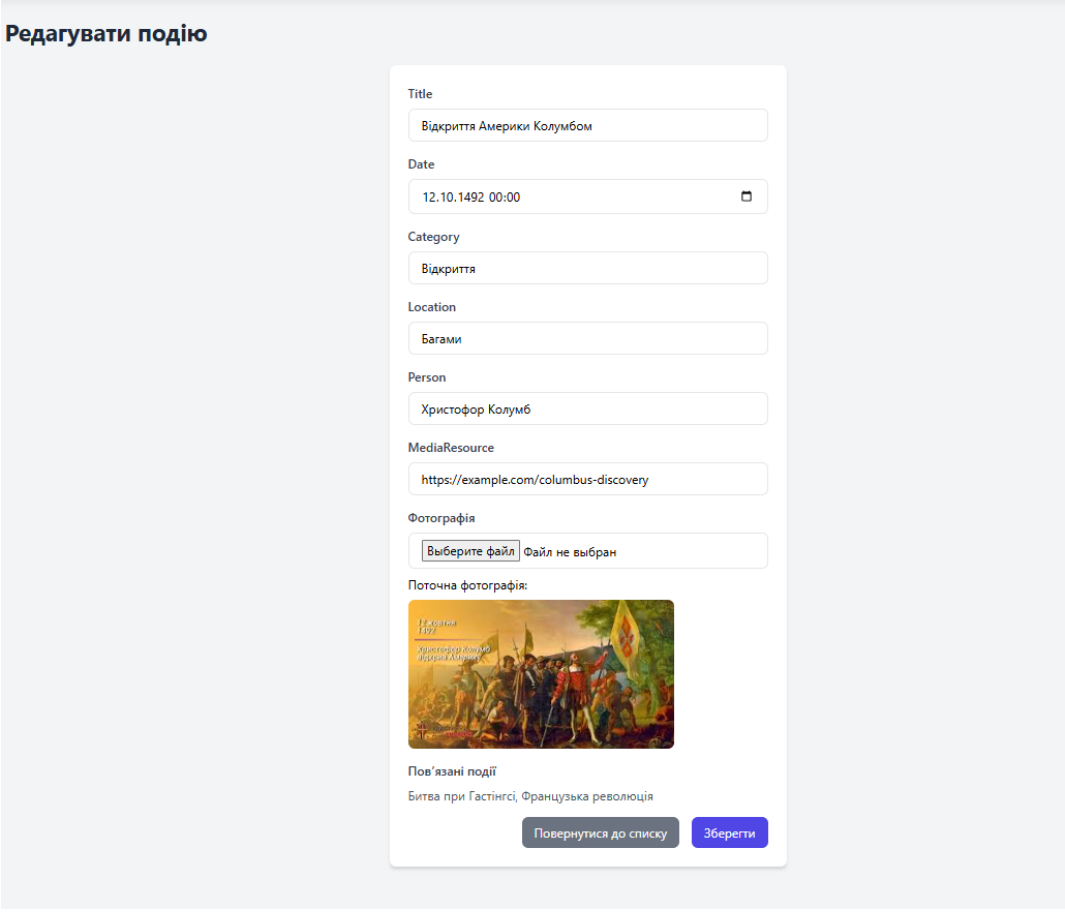


Рис. 0.5 Приклад форми редагування події

Механізм видалення подій:

– інтерфейс: на сторінці details.cshtml є кнопка "видалити", доступна лише для admin, стилізована як bg-red-500 text-white rounded-md p-2. при натисканні з'являється модальне вікно з анімацією для підтвердження дії;

– логіка: javascript-код обробляє натискання кнопки "видалити", відображаючи модальне вікно. при підтвердженні дія delete у eventscontroller видаляє подію через dbcontext.historicalevents.remove(event) і await dbcontext.savechangesasync(). користувач перенаправляється на /events, де список оновлюється. наприклад, адміністратор може видалити подію "sdfsdf", якщо вона більше не потрібна, і всі пов'язані зв'язки також видаляються автоматично через каскадне видалення в базі даних.

Цей функціонал забезпечує гнучке управління історичними даними, дозволяючи адміністраторам додавати нові події, оновлювати існуючі записи та видаляти непотрібні, що є важливим для підтримки актуальності даних у системі.

4.3 Формування серверної логіки

Серверна частина "HistoricalEventsAppNew" створена на основі фреймворку ASP.NET Core MVC, що забезпечує модульну організацію коду, високу продуктивність і зручність роботи з базами даних через Entity Framework Core. Серверна логіка відповідає за обробку запитів, взаємодію з базою даних і повернення результатів клієнту, забезпечуючи стабільну та швидку роботу системи.

4.3.1 Підготовка проєкту до роботи

Розробка серверної частини розпочалася зі створення проєкту в Visual Studio 2022, використовуючи шаблон ASP.NET Core MVC. Цей шаблон автоматично створив базову структуру проєкту, включаючи необхідні каталоги та файли конфігурації. Основний файл Program.cs налаштовує пайплайн обробки запитів, додаючи сервіси, такі як Entity Framework Core для роботи з SQL Server, а також

маршрутизацію для контролерів. У файлі appsettings.json визначено конфігурацію, зокрема рядок підключення до бази даних:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=HistoricalEventsDB;Trusted_Connection=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Ця конфігурація забезпечує стабільне підключення до SQL Server, дозволяючи серверу ефективно взаємодіяти з базою даних. Visual Studio 2022 також забезпечило зручне середовище для роботи, включаючи інструменти для налагодження, автодоповнення коду та інтеграцію з Git, що прискорило процес розробки серверної частини.

4.3.2 Моделі даних і контекст бази даних

Для роботи з даними створено моделі, такі як HistoricalEvent і HistoricalEventRelation, які визначені в папці Models. Модель HistoricalEvent представляє історичну подію з полями, такими як Title, Date, Category, Location і MediaResource, тоді як HistoricalEventRelation визначає зв’язки між подіями через поля SourceEventId, TargetEventId і RelationType. Клас ApplicationDbContext налаштовує доступ до SQL Server через Entity Framework Core, визначаючи набори даних (DbSet) для кожної таблиці. У проєкті тестові дані вводилися вручну, але метод OnModelCreating у ApplicationDbContext дозволяє налаштувати початкові дані, якщо це необхідно. Для створення структури бази даних використано команди:

dotnet ef migrations add InitialCreate

dotnet ef database update

Ці команди створили базу даних HistoricalEventsDB із таблицями HistoricalEvents і HistoricalEventRelations, готових до використання. Entity Framework Core забезпечує зручний доступ до даних, дозволяючи працювати з об’єктами C# замість написання SQL-запитів, що значно спростило розробку.

4.3.3 Реалізація бізнес-логіки та обробки запитів

Для ізоляції бізнес-логіки від контролерів створено сервіс DatabaseService, який інкапсулює операції з базою даних. Наприклад, метод GetEventRelationsAsync у DatabaseService повертає зв’язки для певної події, використовуючи асинхронні запити через Entity Framework Core. Це дозволяє контролерам зосередитися на обробці HTTP-запитів, а не на низькорівневій логіці роботи з даними. Асинхронність забезпечує високу продуктивність, особливо при завантаженні великих обсягів даних, наприклад, для відображення графа зв’язків.

Контролери, такі як EventsController, відповідають за обробку запитів від клієнта. Наприклад, дія Graph у EventsController завантажує дані про події та зв’язки через DatabaseService і передає їх у представлення Graph.cshtml для створення інтерактивного графа. Дія Index відповідає за відображення списку подій із можливістю пошуку та фільтрації, тоді як дії Create і Edit дозволяють адміністраторам створювати та редагувати події. Контролери також перевіряють ролі користувачів, забезпечуючи доступ до певних функцій (наприклад, редагування подій) лише для адміністраторів (Admin).

4.3.4 Запуск і перевірка роботи сервера

Проєкт запускався за допомогою команди dotnet run, яка активувала сервер на адресі https://localhost:7232. У режимі розробки використовувалася команда dotnet watch, яка забезпечує автоматичний перезапуск сервера при змінах у коді, що значно прискорювало процес тестування та налагодження. Наприклад, при зміні логіки в EventsController для додавання нового методу сервер автоматично перезапускався, дозволяючи швидко перевірити оновлення. Логи в консолі після

запуску показували статус сервера, підтверджуючи його готовність до обробки запитів.

Рис. 4.3 Зразок консолі після запуску сервера

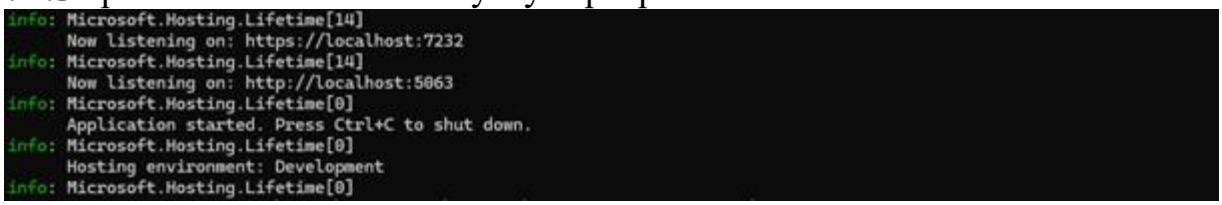


Рис. 0.6 Зразок консолі піссля запуску сервера

4.3.5 Розгортання системи

На етапі розробки проєкт тестувався локально через Visual Studio 2022. Розгортання на віддалений сервер не проводилося, але ASP.NET Core MVC забезпечує кросплатформність, що дозволяє легко розгорнути додаток на платформах, таких як Azure або IIS. Наприклад, для розгортання на Azure потрібно лише налаштувати відповідний профіль публікації у Visual Studio, що спрощує процес перенесення додатку в хмарне середовище. Локальне тестування показало стабільну роботу сервера, а використання dotnet watch дозволило швидко виявляти та виправляти помилки під час розробки.

4.4 Перевірка роботи веб-застосунку

Для забезпечення якості та стабільності "HistoricalEventsAppNew" було проведено ручне тестування, яке охопило всі ключові компоненти додатку, включаючи інтерфейс, функціональність і продуктивність.

Основні аспекти тестування:

– відображення сторінок: перевірялися сторінки, такі як /events і /events/graph, на коректність відображення даних. наприклад, на сторінці /events/graph тестувалося відображення трьох вузлів (sdfsdf, sdfsdf, dfgfdg) і трьох зв’язків (preceded, caused, coincided), щоб переконатися, що граф коректно відображає семантичні зв’язки між подіями;

– функціональність графа: перевірялася можливість перетягування вузлів і коректність відображення типів зв’язків. наприклад, тестувалися сценарії з відсутністю зв’язків, щоб упевнитися, що граф відображає лише ізольовані вузли

без помилок. перетягування вузлів дозволяло користувачам зручно аналізувати зв’язки, наприклад, переміщаючи вузол "sdfsdf" для кращої видимості його зв’язків;

– адаптивність: використовувалися інструменти devtools для перевірки відображення на різних пристроях. наприклад, граф зв’язків адаптувався до смартфонів через preserveaspectratio="xmidymid meet", що забезпечувало коректне масштабування. на сторінці /events перевірялася адаптивність списку подій, щоб картки коректно відображалися на малих екранах, наприклад, із меншим розміром шрифту для заголовків;

– обробка помилок: симулювалися помилки, такі як звернення до неіснуючого маршруту (наприклад, /events/invalid), щоб перевірити, чи отримує користувач зрозуміле повідомлення про помилку, наприклад, "сторінка не знайдена". це забезпечило коректну обробку виняткових ситуацій без збоїв у роботі додатку;

– продуктивність: час завантаження сторінок, таких як /events, не перевищував 2 секунд, що відповідає вимогам до швидкості роботи. наприклад, при завантаженні списку з 50 подій сторінка відображалася за 1,5 секунди, що забезпечувало комфортний користувацький досвід.

Переваги ручного тестування:

– гнучкість: ручне тестування дозволило швидко виявляти візуальні дефекти, наприклад, проблеми з адаптивністю графа на малих екранах, і вносити виправлення;

– ефективність: для проєкту такого масштабу ручне тестування виявилося швидшим і менш затратним, ніж налаштування автоматизованих тестів, дозволяючи зосередитися на ключових функціях.

Обмеження ручного тестування:

– неможливість повторного використання: ручне тестування не забезпечує повторюваності, що може призводити до пропуску помилок при повторних перевірках. наприклад, якщо тестер пропустить перевірку адаптивності на певному розмірі екрана, це може залишитися непоміченим;

– обмежене покриття: через ручний підхід не всі можливі сценарії були протестовані, зокрема обробка великих обсягів даних (наприклад, 1000 подій), що може вимагати додаткового тестування в майбутньому (див. рисунок 4.7).

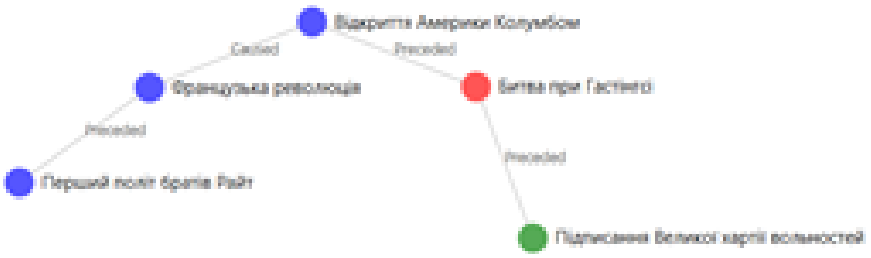


Рис. 0.7 Зразок тестування графа зв’язків

ВИСНОВКИ

У процесі виконання дипломної роботи було досягнуто мети – створення веб-застосунку "HistoricalEventsAppNew", який забезпечує ефективне управління історичними даними, їх структуризацію через онтологічну модель і візуалізацію зв’язків між подіями у форматі інтерактивного графа. Об’єктом дослідження став процес аналізу та візуалізації історичних подій, а предметом – програмне забезпечення для управління історичними даними з використанням онтологій, що дозволило зосередитися на розробці рішення, яке відповідає потребам дослідників, студентів, викладачів і ентузіастів історії.

Для реалізації поставлених завдань було проведено аналіз існуючих платформ, таких як History Timeline, World History Encyclopedia і ChronoZoom, що допомогло визначити їхні сильні та слабкі сторони, а також сформувати вимоги до власного рішення. Огляд засобів розробки клієнт-серверного застосунку виявив оптимальні технології: ASP.NET Core MVC для серверної логіки, SQL Server як СУБД, Entity Framework Core для роботи з базою даних, JavaScript із D3.js для інтерактивної візуалізації, Tailwind CSS для стилізації, і Razor для створення динамічних сторінок. Ці технології забезпечили високу продуктивність, гнучкість і зручність розробки.

Архітектура веб-застосунку спроєктована за моделлю MVC, де серверна частина обробляє запити через контролери (наприклад, EventsController), моделі (HistoricalEvent, HistoricalEventRelation) забезпечують структуру даних у SQL Server, а клієнтська частина, реалізована через Razor і JavaScript, генерує інтерактивний інтерфейс. Ключові компоненти включають: сторінки для перегляду подій (Index.cshtml), їх деталей (Details.cshtml), створення зв’язків (CreateRelation.cshtml) і візуалізації графа зв’язків (Graph.cshtml) за допомогою D3.js. Онтологічна модель реалізована через таблицю HistoricalEventRelations із типами зв’язків ("передувала", "спричинила" тощо), що дозволяє аналізувати семантичні зв’язки між подіями.

Розробка включала створення адаптивного інтерфейсу з Tailwind CSS, інтерактивного графа з D3.js, системи авторизації з ролями (Admin і User), а також асинхронної обробки запитів через EF Core для підвищення продуктивності. Тестування проводилося вручну, що дозволило перевірити коректність відображення сторінок, функціональність графа (включаючи перетягування вузлів), адаптивність до різних пристроїв і стабільність роботи з SQL Server. Час завантаження сторінок не перевищував 2 секунд, що відповідає вимогам.

Результатом роботи став функціональний веб-застосунок, який успішно вирішує завдання управління історичними подіями: забезпечує зручний доступ до інформації, дозволяє створювати семантичні зв’язки між подіями та візуалізує їх у вигляді інтерактивного графа. Застосунок підтримує пошук і фільтрацію подій, а система авторизації гарантує безпечне управління даними. Використання сучасних технологій і модульної архітектури створює основу для подальшого розвитку, наприклад, додавання аналітики зв’язків або розширення доступності на мобільні платформи.

Робота пройшла апробацію. За її результатами опубліковано наступні тези доповідей:

1. Черниш А.О., Щербина І.С., “Розробка web-застосунку для візуалізації історичних подій на основі онтологій”, VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.25, ДУІКТ, Київ, Україна, с. 199-202.

2. Черниш А.О., Щербина І.С., “Безпека даних у веб-застосунку для візуалізації історичних подій на основі онтологій: сучасні виклики та технологічні рішення” Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с. 338-339.

ПЕРЕЛІК ПОСИЛАНЬ

1. Microsoft. ASP.NET Core Documentation. URL: https://docs.microsoft.com/en-us/aspnet/core (дата звернення: 05.05.2025).
2. Microsoft. Entity Framework Core Documentation. URL: https://docs.microsoft.com/en-us/ef/core (дата звернення: 05.05.2025).
3. Microsoft. SQL Server Documentation. URL: https://docs.microsoft.com/en-us/sql/sql-server (дата звернення: 05.05.2025).
4. D3.js Documentation. URL: https://d3js.org/ (дата звернення: 05.05.2025).
5. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 05.05.2025).
6. Microsoft. C# Programming Guide. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide (дата звернення: 05.05.2025).
7. Mozilla Developer Network. JavaScript Documentation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 05.05.2025).
8. Freeman, A. Pro ASP.NET Core 7. Apress, 2023. 1024 p. URL: https://doi.org/10.1007/978-1-4842-9244-0 (дата звернення: 05.05.2025).
9. Lerman, J., & Miller, B. Programming Entity Framework: Code First. O'Reilly Media, 2022. 576 p. URL: https://www.oreilly.com/library/view/programming-entity-framework/9781449312948 (дата звернення: 05.05.2025).
10. Microsoft. SQL Server Technical Documentation. URL: https://docs.microsoft.com/en-us/sql/ (дата звернення: 05.05.2025).
11. Bostock, M. Data-Driven Documents: D3.js Guide. URL: https://d3js.org/getting-started (дата звернення: 05.05.2025).
12. Tailwind Labs. Tailwind CSS: Utility-First CSS Framework. URL: https://tailwindcss.com/ (дата звернення: 05.05.2025).
13. Troelsen, A., & Japikse, P. Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming. Apress, 2023. 1200 p. URL: https://doi.org/10.1007/978-1-4842-7869-7 (дата звернення: 05.05.2025).

14. Galloway, J., & Wilson, N. Professional ASP.NET Core MVC: Building Web Applications with ASP.NET Core 7. Wrox, 2023. 850 p. URL: https://www.wrox.com/book/professional-aspnet-core-mvc (дата звернення: 05.05.2025).
15. Miller, R., & Johnson, S. Entity Framework Core in Action, 2nd Edition. Manning Publications, 2023. 700 p. URL: https://www.manning.com/books/entity-framework-core-in-action-second-edition (дата звернення: 05.05.2025).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-застосунку для візуалізації історичних подій на основі онтологій

Виконав студент 4 курсу
групи ПД-42
Черниш Андрій Олегович
Керівник роботи
к.т.н., (доцент)доцент кафедри ІПЗ Щербина Ірина Сергіївна

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – спрощення аналізу та візуалізації історичних подій на основі онтології.

Об’єкт дослідження – процес аналізу та візуалізації історичних подій на основі онтологічної моделі.

Предмет дослідження – веб-застосунок для управління історичними подіями з візуалізацією та використанням онтологій.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести огляд засобів для розробки програмного забезпечення клієнт-серверного застосунку для візуалізації історичних подій на основі онтологій.
2. Здійснити огляд та аналіз існуючих веб-сайтів, що використовуються для візуалізації історичних подій на основі онтологій.
3. Сформувати вимоги до розробки клієнт-серверного застосунку для візуалізації історичних подій на основі онтологій.
4. Спроектувати архітектуру клієнт-серверного застосунку для візуалізації історичних подій на основі онтологій.
5. Розробити клієнт-серверний застосунок для візуалізації історичних подій на основі онтологій.
6. Провести тестування клієнт-серверного застосунку для візуалізації історичних подій на основі онтологій.

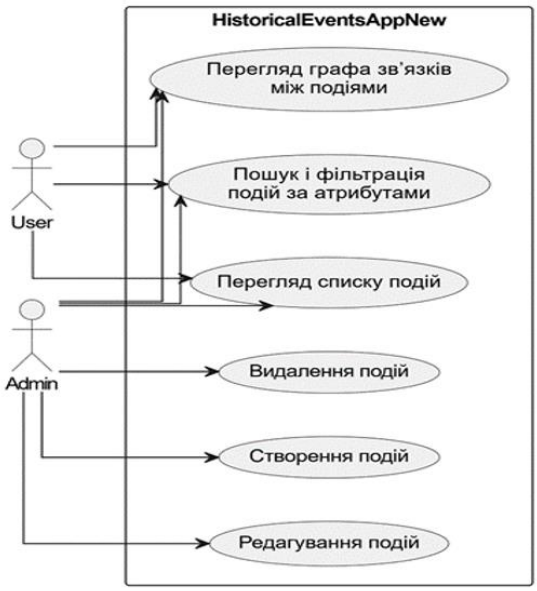
АНАЛІЗ АНАЛОГІВ

Критерій	History Timeline	World History Encyclopedia	ChronoZoom	HistoricalEventsAppNew
Основний функціонал	Хронологічна шкала, фільтрація за роками	Детальні статті, пошук за темами	Масштабований часовий масштаб	Перегляд подій, граф зв'язків, пошук
Інтерактивність	Обмежена (немає графів зв'язків)	Низька (статичні сторінки)	Висока (масштабування хронології)	Висока (граф D3.js, перетягування)
Онтологічна модель	Відсутня	Відсутня	Відсутня	Присутня (зв'язки між подіями)
Внесення даних	Неможливе	Неможливе	Обмежене (власні хронології)	Можливе (додавання, редагування)

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

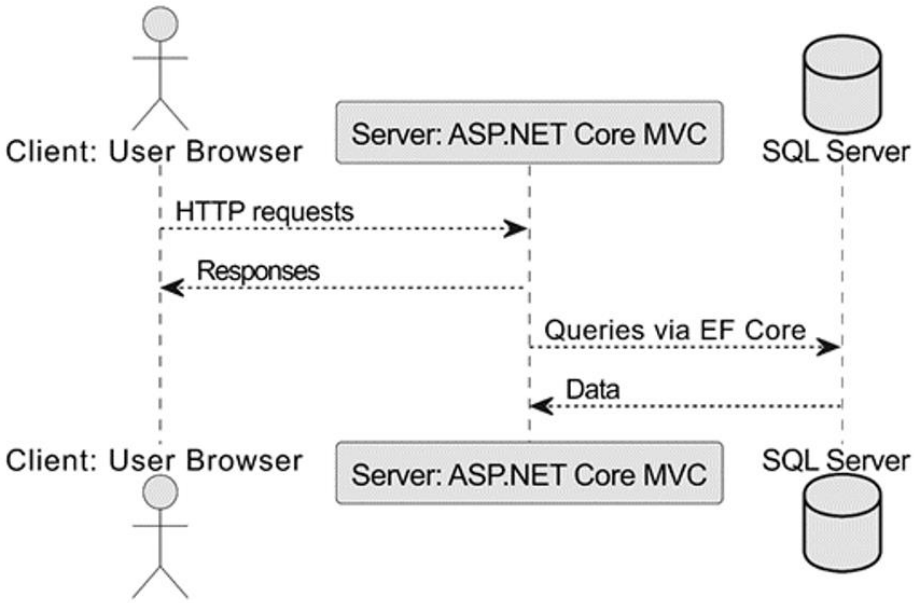
5

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

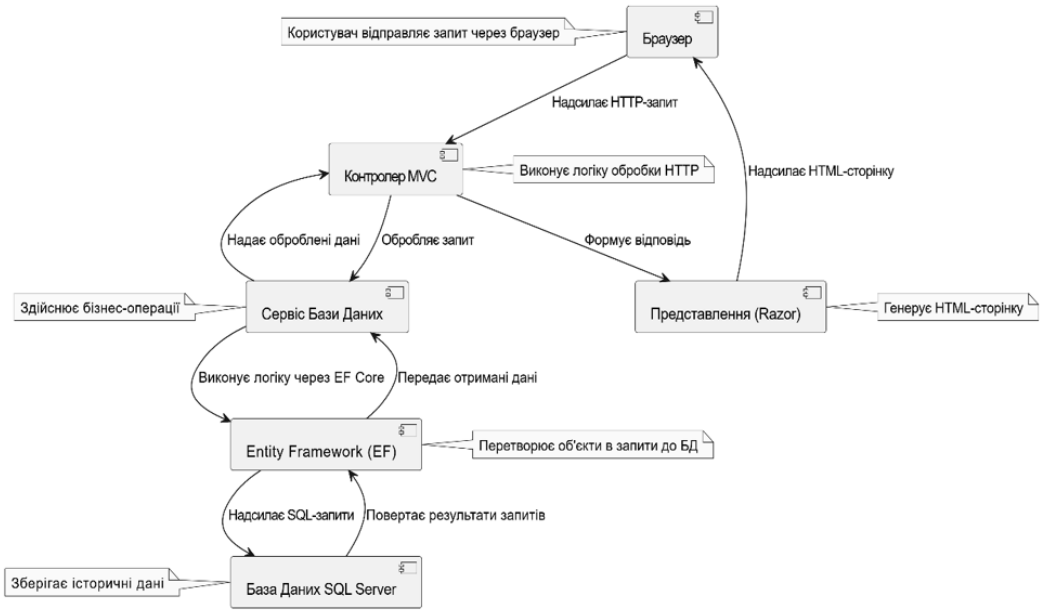


6

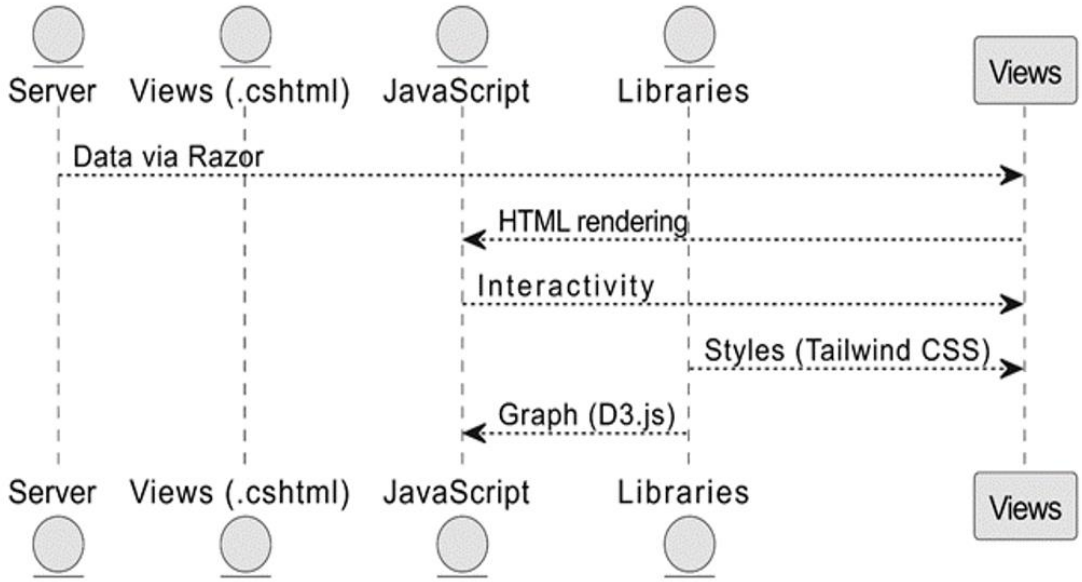
ДІАГРАМА КЛІЄНТ-СЕРВЕРНОЇ АРХІТЕКТУРИ



ДІАГРАМА СТРУКТУРИ СЕРВЕРНОЇ ЧАСТИНИ

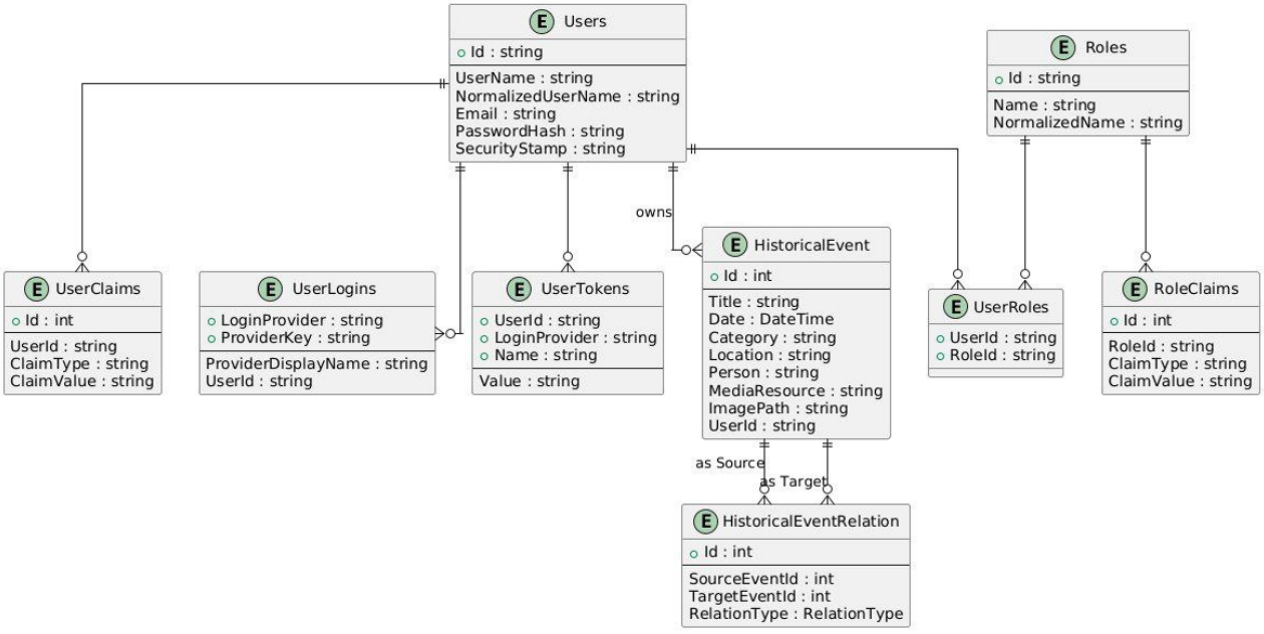


ДІАГРАМА СТРУКТУРИ КЛІЄНТСЬКОЇ ЧАСТИНИ



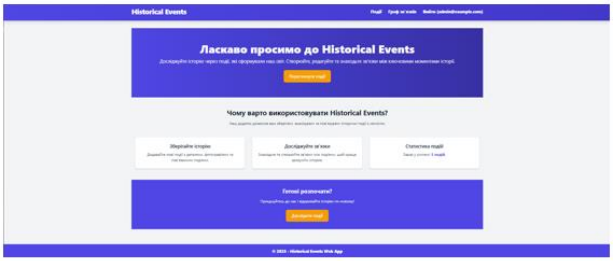
9

ДІАГРАМА СТРУКТУРИ БАЗИ ДАНИХ

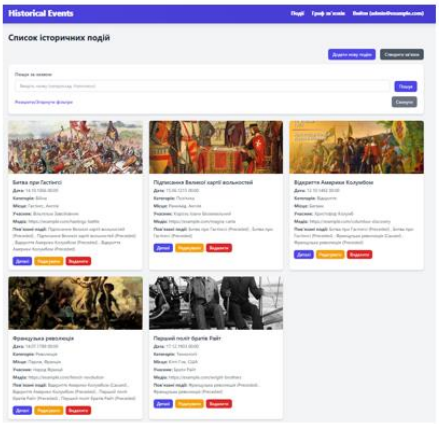


10

ЕКРАННІ ФОРМИ



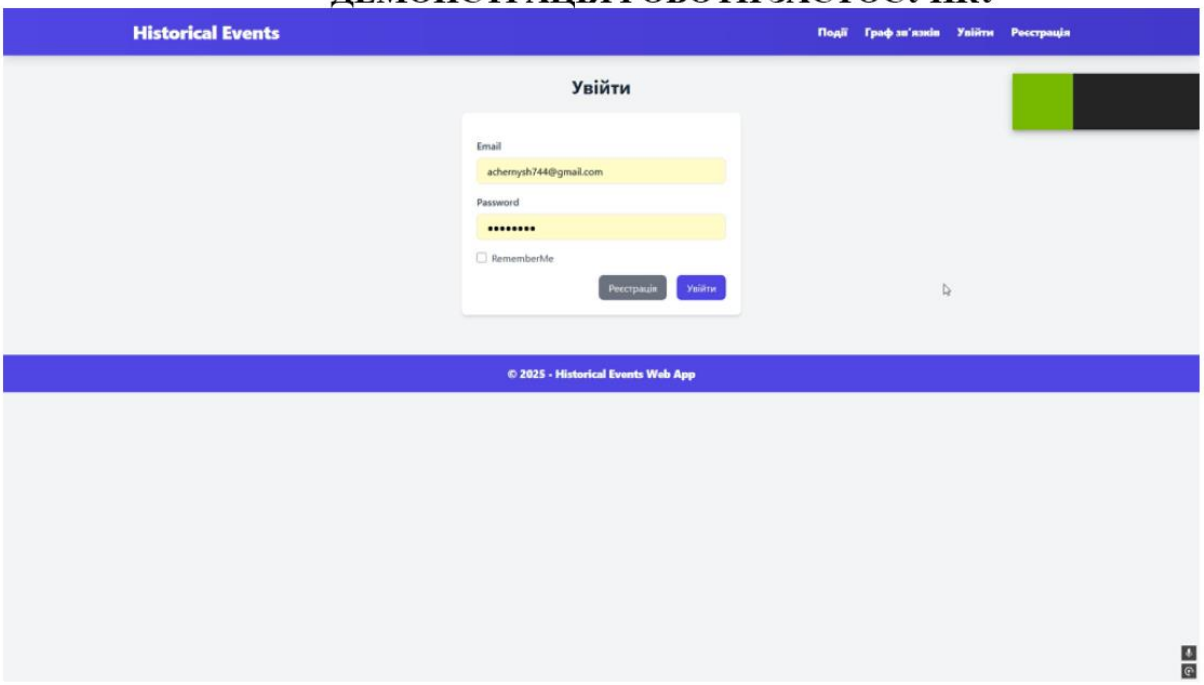
Головна сторінка



Сторінка історичних подій

Сторінка графу зв’язків між подіями

ДЕМОНСТРАЦІЯ РОБОТИ ЗАСТОСУНКУ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Черниш А.О., Щербина І.С., “Розробка web-застосунку для візуалізації історичних подій на основі онтологій”, VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.25, ДУІКТ, Київ, Україна, с. 199-202.

2. Черниш А.О., Щербина І.С., “Безпека даних у веб-застосунку для візуалізації історичних подій на основі онтологій: сучасні виклики та технологічні рішення” Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с. 338-339.

ВИСНОВКИ

1. Здійснено огляд сучасних технологій для побудови клієнт-серверних застосунків. Обрано стек технологій, який забезпечує ефективну роботу із семантичними даними та можливість реалізації інтерактивної візуалізації.
2. Проведено аналіз існуючих веб-ресурсів для візуалізації історичних подій на основі онтологій. Виявлено обмеження в їхній функціональності та відсутність інтерактивності, що обґрунтовує доцільність розробки нового рішення.
3. Сформовано вимоги до програмного забезпечення, що дозволило чітко окреслити функціональні можливості системи.
4. Спроєктовано архітектуру веб-застосунку з урахуванням принципів модульності, розширюваності та підтримки онтологічних моделей.
5. Реалізовано прототип клієнт-серверного застосунку, який дозволяє здійснювати візуалізацію історичних подій, фільтрацію даних та перегляд зв’язків між подіями.
6. Протестовано на відповідність сформульованим вимогам.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

DatabaseService.cs:

```
using Microsoft.EntityFrameworkCore;
using HistoricalEventsAppNew.Data;
using HistoricalEventsAppNew.Models;
using HistoricalEventsAppNew.ViewModels;
using System.IO;

namespace HistoricalEventsAppNew.Services
{
   public class DatabaseService
   {
      private readonly ApplicationDbContext _context;
      private readonly IWebHostEnvironment _environment;

      public DatabaseService(ApplicationDbContext context,
IWebHostEnvironment environment)
      {
         _context = context;
         _environment = environment;
      }

      public async Task<List<HistoricalEventRelation>>
GetEventRelationsAsync(int eventId)
      {
         return await _context.HistoricalEventRelations
            .Include(her => her.SourceEvent)
            .Include(her => her.TargetEvent)
            .Where(her => her.SourceEventId == eventId ||
her.TargetEventId == eventId)
            .ToListAsync();
      }

      public async Task<List<HistoricalEventViewModel>>
GetHistoricalEventsAsync()
      {
         var events = await _context.HistoricalEvents
            .Include(he => he.RelatedEvents)
            .ThenInclude(her => her.TargetEvent)
            .ToListAsync();

         var eventViewModels = events.Select(he => new
HistoricalEventViewModel
         {
            Id = he.Id,
            Title = he.Title,
            Date = he.Date,
            Category = he.Category,
            Location = he.Location,
            Person = he.Person,
            MediaResource = he.MediaResource,
            ImagePath = he.ImagePath,
            UserId = he.UserId,
            RelatedEventTitles =
_context.HistoricalEventRelations
               .Where(her => her.SourceEventId == he.Id ||
her.TargetEventId == he.Id)
               .Select(her => her.SourceEventId == he.Id ?
her.TargetEvent.Title : her.SourceEvent.Title)
               .Distinct()
               .ToList() ?? new List<string>()
         }).ToList();

         return eventViewModels;
      }

      public async Task<HistoricalEventViewModel>
GetHistoricalEventByIdAsync(int id)
      {
         var historicalEvent = await _context.HistoricalEvents
            .Include(he => he.RelatedEvents)
            .ThenInclude(her => her.TargetEvent)
            .FirstOrDefaultAsync(he => he.Id == id);

         if (historicalEvent == null)
         {
            return null;
         }

         var relatedEventTitles = await
_context.HistoricalEventRelations
            .Where(her => her.SourceEventId == id ||
her.TargetEventId == id)
            .Select(her => her.SourceEventId == id ?
her.TargetEvent.Title : her.SourceEvent.Title)
            .Distinct()
            .ToListAsync();

         return new HistoricalEventViewModel
         {
            Id = historicalEvent.Id,
            Title = historicalEvent.Title,
            Date = historicalEvent.Date,
            Category = historicalEvent.Category,
            Location = historicalEvent.Location,
            Person = historicalEvent.Person,
            MediaResource = historicalEvent.MediaResource,
            ImagePath = historicalEvent.ImagePath,
            UserId = historicalEvent.UserId,
            RelatedEventTitles = relatedEventTitles ?? new
List<string>()
         };
      }

      public async Task
CreateHistoricalEventAsync(HistoricalEvent historicalEvent)
      {
         _context.HistoricalEvents.Add(historicalEvent);
         await _context.SaveChangesAsync();
      }

      public async Task
UpdateHistoricalEventAsync(HistoricalEvent updatedEvent)
      {
         var originalEvent = await _context.HistoricalEvents
            .FirstOrDefaultAsync(he => he.Id ==
updatedEvent.Id);

         if (originalEvent == null)
         {
            throw new Exception($"HistoricalEvent with
Id={updatedEvent.Id} not found.");
         }

         originalEvent.Title = updatedEvent.Title;
         originalEvent.Date = updatedEvent.Date;
         originalEvent.Category = updatedEvent.Category;
         originalEvent.Location = updatedEvent.Location;
         originalEvent.Person = updatedEvent.Person;
```

```
        originalEvent.MediaResource =
updatedEvent.MediaResource;
        originalEvent.ImagePath = updatedEvent.ImagePath;
        originalEvent.UserId = updatedEvent.UserId;

        await _context.SaveChangesAsync();
    }

    public async Task DeleteHistoricalEventAsync(int id)
    {
        var historicalEvent = await
_context.HistoricalEvents.FindAsync(id);
        if (historicalEvent == null)
        {
            Console.WriteLine($"HistoricalEvent with Id={id}
not found. Nothing to delete.");
            return;
        }

        var relatedRelations = await
_context.HistoricalEventRelations
            .Where(her => her.SourceEventId == id ||
her.TargetEventId == id)
            .ToListAsync();

        if (relatedRelations.Any())
        {
            Console.WriteLine($"Found
{relatedRelations.Count} relations for HistoricalEvent with
Id={id}. Deleting relations...");
            foreach (var relation in relatedRelations)
            {
                Console.WriteLine($"Deleting relation:
Id={relation.Id}, SourceEventId={relation.SourceEventId},
TargetEventId={relation.TargetEventId}");
            }

_context.HistoricalEventRelations.RemoveRange(relatedRelati
ons);
            await _context.SaveChangesAsync();
            Console.WriteLine("All related relations deleted.");
        }

        if (!string.IsNullOrEmpty(historicalEvent.ImagePath))
        {
            var imagePath =
Path.Combine(_environment.WebRootPath,
historicalEvent.ImagePath.TrimStart('/'));
            if (System.IO.File.Exists(imagePath))
            {
                try
                {
                    System.IO.File.Delete(imagePath);
                    Console.WriteLine($"Deleted image file:
{imagePath}");
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error deleting image file
{imagePath}: {ex.Message}");
                }
            }
        }

        _context.HistoricalEvents.Remove(historicalEvent);
        await _context.SaveChangesAsync();
    }

    public async Task
CreateHistoricalEventRelationAsync(HistoricalEventRelation
relation)
    {
        var existingRelation = await
_context.HistoricalEventRelations
            .AnyAsync(r => (r.SourceEventId ==
relation.SourceEventId && r.TargetEventId ==
relation.TargetEventId) ||
                           (r.SourceEventId == relation.TargetEventId
&& r.TargetEventId == relation.SourceEventId));

        if (existingRelation)
        {
            Console.WriteLine($"Relation between
SourceEventId={relation.SourceEventId} and
TargetEventId={relation.TargetEventId} already exists (or vice
versa). Skipping creation.");
            return;
        }

        _context.HistoricalEventRelations.Add(relation);
        Console.WriteLine($"Added direct relation:
SourceEventId={relation.SourceEventId},
TargetEventId={relation.TargetEventId}");

        var reverseRelation = new HistoricalEventRelation
        {
            SourceEventId = relation.TargetEventId,
            TargetEventId = relation.SourceEventId
        };

_context.HistoricalEventRelations.Add(reverseRelation);
        Console.WriteLine($"Added reverse relation:
SourceEventId={reverseRelation.SourceEventId},
TargetEventId={reverseRelation.TargetEventId}");

        await _context.SaveChangesAsync();
    }

    public async Task<List<HistoricalEvent>>
GetAllHistoricalEventsAsync()
    {
        return await _context.HistoricalEvents.ToListAsync();
    }

    public async Task<List<string>>
GetRelatedEventTitlesAsync(int eventId)
    {
        var relatedEventTitles = await
_context.HistoricalEventRelations
            .Where(her => her.SourceEventId == eventId ||
her.TargetEventId == eventId)
            .Select(her => her.SourceEventId == eventId ?
her.TargetEvent.Title : her.SourceEvent.Title)
            .Distinct()
            .ToListAsync();

        return relatedEventTitles ?? new List<string>();
    }

    public async Task<List<HistoricalEventViewModel>>
GetFilteredHistoricalEventsAsync(
  string searchTitle, string filterTitle, DateTime? filterDate,
string filterCategory,
  string filterLocation, string filterPerson)
    {
        var query = _context.HistoricalEvents
            .Include(he => he.RelatedEvents)
            .ThenInclude(her => her.TargetEvent)
```

```
            .AsQueryable();

        if (!string.IsNullOrEmpty(searchTitle))
        {
            query = query.Where(he =>
he.Title.Contains(searchTitle));
        }

        if (!string.IsNullOrEmpty(filterTitle))
        {
            query = query.Where(he =>
he.Title.Contains(filterTitle));
        }

        if (filterDate.HasValue)
        {
            query = query.Where(he => he.Date.Date ==
filterDate.Value.Date);
        }

        if (!string.IsNullOrEmpty(filterCategory))
        {
            query = query.Where(he =>
he.Category.Contains(filterCategory));
        }

        if (!string.IsNullOrEmpty(filterLocation))
        {
            query = query.Where(he =>
he.Location.Contains(filterLocation));
        }

        if (!string.IsNullOrEmpty(filterPerson))
        {
            query = query.Where(he =>
he.Person.Contains(filterPerson));
        }

        var events = await query.ToListAsync();

        var eventViewModels = events.Select(he => new
HistoricalEventViewModel
        {
            Id = he.Id,
            Title = he.Title,
            Date = he.Date,
            Category = he.Category,
            Location = he.Location,
            Person = he.Person,
            MediaResource = he.MediaResource,
            ImagePath = he.ImagePath,
            UserId = he.UserId,
            RelatedEventTitles =
_context.HistoricalEventRelations
                .Where(her => her.SourceEventId == he.Id ||
her.TargetEventId == he.Id)
                .Select(her => her.SourceEventId == he.Id ?
her.TargetEvent.Title : her.SourceEvent.Title)
                .Distinct()
                .ToList() ?? new List<string>()
        }).ToList();

        return eventViewModels;
    }

    public async Task<List<string>> GetUniqueTitlesAsync()
    {
        return await _context.HistoricalEvents
            .Select(he => he.Title)
            .Distinct()
            .ToListAsync();
    }

    public async Task<List<string>>
GetUniqueCategoriesAsync()
    {
        return await _context.HistoricalEvents
            .Select(he => he.Category)
            .Distinct()
            .ToListAsync();
    }

    public async Task<List<string>>
GetUniquePersonsAsync()
    {
        return await _context.HistoricalEvents
            .Select(he => he.Person)
            .Distinct()
            .ToListAsync();
    }

    public async Task<List<string>>
GetUniqueLocationsAsync()
    {
        return await _context.HistoricalEvents
            .Select(he => he.Location)
            .Distinct()
            .ToListAsync();
    }

    public async Task<List<DateTime>>
GetUniqueDatesAsync()
    {
        return await _context.HistoricalEvents
            .Select(he => he.Date)
            .Distinct()
            .ToListAsync();
    }
  }
}
```

Program.cs

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using HistoricalEventsAppNew.Data;
using HistoricalEventsAppNew.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.Authorization;
using Microsoft.AspNetCore.Mvc.Razor;
using HistoricalEventsAppNew.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddLocalization(options =>
options.ResourcesPath = "Resources");

builder.Services.AddDbContext<ApplicationDbContext>(optio
ns =>

options.UseSqlServer(builder.Configuration.GetConnectionStri
ng("DefaultConnection")));

builder.Services.AddIdentity<IdentityUser,
IdentityRole>(options =>
{
  options.SignIn.RequireConfirmedAccount = false;
  options.Password.RequireDigit = false;
```

```
    options.Password.RequiredLength = 6;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
    options.Password.RequireLowercase = false;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();

builder.Services.AddScoped<RoleManager<IdentityRole>>();

builder.Services.AddScoped<DatabaseService>();
builder.Services.AddControllersWithViews(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
}).AddViewLocalization(LanguageViewLocationExpanderFor
mat.Suffix);
builder.Services.ConfigureApplicationCookie(options =>
{



 options.LoginPath = "/Account/Login";
    options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
    options.SlidingExpiration = true;
});

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context =
services.GetRequiredService<ApplicationDbContext>();
    var userManager =
services.GetRequiredService<UserManager<IdentityUser>>();
    var roleManager =
services.GetRequiredService<RoleManager<IdentityRole>>();

    // Застосовуємо міграції
    context.Database.Migrate();

    // Створюємо ролі
    if (!await roleManager.RoleExistsAsync("Admin"))
    {
        await roleManager.CreateAsync(new
IdentityRole("Admin"));
    }

    if (!await roleManager.RoleExistsAsync("User"))
    {
        await roleManager.CreateAsync(new
IdentityRole("User"));
    }

    // Створюємо адміністратора
    var adminEmail = "admin@example.com";
    var admin = await
userManager.FindByEmailAsync(adminEmail);
    if (admin == null)
    {
        admin = new IdentityUser { UserName = adminEmail,
Email = adminEmail };
        var result = await userManager.CreateAsync(admin,
"Admin123!");
        if (result.Succeeded)
        {
            await userManager.AddToRoleAsync(admin,
"Admin");
        }
    }

    // Створюємо користувача
    var userEmail = "user@example.com";
    var user = await
userManager.FindByEmailAsync(userEmail);
    if (user == null)
    {
        user = new IdentityUser { UserName = userEmail, Email
= userEmail };
        var result = await userManager.CreateAsync(user,
"User123!");
        if (result.Succeeded)
        {
            await userManager.AddToRoleAsync(user, "User");
        }
    }

    // Перевіряємо, чи є події в базі даних
    if (!context.HistoricalEvents.Any())
    {
        // Отримуємо Id адміністратора
        var adminId = admin.Id;

        // Додаємо початкові події (без явного вказування Id)
        var events = new List<HistoricalEvent>
        {
            new HistoricalEvent
            {
                Title = "Битва при Гастінгсі",
                Date = new DateTime(1066, 10, 14),
                Category = "Війна",
                Location = "Гастінгс, Англія",
                Person = "Вільгельм Завойовник",
                MediaResource = "https://example.com/hastings-
battle",
                ImagePath = "/uploads/hastings.jpg",
                UserId = adminId
            },
            new HistoricalEvent
            {
                Title = "Підписання Великої хартії вольностей",
                Date = new DateTime(1215, 6, 15),
                Category = "Політика",
                Location = "Раннімід, Англія",
                Person = "Король Іоанн Безземельний",
                MediaResource = "https://example.com/magna-
carta",
                ImagePath = "/uploads/magna-carta.jpg",
                UserId = adminId
            },
            new HistoricalEvent
            {
                Title = "Відкриття Америки Колумбом",
                Date = new DateTime(1492, 10, 12),
                Category = "Відкриття",
                Location = "Багами",
                Person = "Христофор Колумб",
                MediaResource = "https://example.com/columbus-
discovery",
                ImagePath = "/uploads/columbus.jpg",
                UserId = adminId
            },
            new HistoricalEvent
            {
                Title = "Французька революція",
                Date = new DateTime(1789, 7, 14),
```

```
            Category = "Революція",
            Location = "Париж, Франція",
            Person = "Народ Франції",
            MediaResource = "https://example.com/french-
revolution",
            ImagePath = "/uploads/french-revolution.jpg",
            UserId = adminId
        },
        new HistoricalEvent
        {
            Title = "Перший політ братів Райт",
            Date = new DateTime(1903, 12, 17),
            Category = "Технології",
            Location = "Кітті-Гок, США",
            Person = "Брати Райт",
            MediaResource = "https://example.com/wright-
brothers",
            ImagePath = "/uploads/wright-brothers.jpg",
            UserId = adminId
        }
    };

    // Додаємо події до бази даних
    context.HistoricalEvents.AddRange(events);
    await context.SaveChangesAsync();

    // Отримуємо згенеровані Id подій
    var hastings = await context.HistoricalEvents.FirstAsync(e
=> e.Title == "Битва при Гастінгсі");
    var magnaCarta = await
context.HistoricalEvents.FirstAsync(e => e.Title ==
"Підписання Великої хартії вольностей");
    var columbus = await
context.HistoricalEvents.FirstAsync(e => e.Title ==
"Відкриття Америки Колумбом");
    var frenchRevolution = await
context.HistoricalEvents.FirstAsync(e => e.Title ==
"Французька революція");
    var wrightBrothers = await
context.HistoricalEvents.FirstAsync(e => e.Title == "Перший
політ братів Райт");

    // Додаємо зв’язки між подіями (без явного вказування
Id)
    context.HistoricalEventRelations.AddRange(
        new HistoricalEventRelation
        {
            SourceEventId = hastings.Id,
            TargetEventId = magnaCarta.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = magnaCarta.Id,
            TargetEventId = hastings.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = hastings.Id,
            TargetEventId = columbus.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = columbus.Id,
            TargetEventId = hastings.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = columbus.Id,
            TargetEventId = frenchRevolution.Id,
            RelationType = RelationType.Caused
        },
        new HistoricalEventRelation
        {
            SourceEventId = frenchRevolution.Id,
            TargetEventId = columbus.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = frenchRevolution.Id,
            TargetEventId = wrightBrothers.Id,
            RelationType = RelationType.Preceded
        },
        new HistoricalEventRelation
        {
            SourceEventId = wrightBrothers.Id,
            TargetEventId = frenchRevolution.Id,
            RelationType = RelationType.Preceded
        }
    );

    // Зберігаємо зміни в базі даних
    await context.SaveChangesAsync();
  }
}

if (!app.Environment.IsDevelopment())
{
  app.UseExceptionHandler("/Home/Error");
  app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.Use(async (context, next) =>
{
  Console.WriteLine($"Request Path: {context.Request.Path},
IsAuthenticated: {context.User.Identity?.IsAuthenticated}");
  if (context.User.Identity?.IsAuthenticated ?? false)
  {
    if
(context.Request.Path.StartsWithSegments("/Account/Login")
||
context.Request.Path.StartsWithSegments("/Account/Register"
))
    {
      context.Response.Redirect("/Events");
    }
  }
  else if
(!context.Request.Path.StartsWithSegments("/Account/Login")
&&
!context.Request.Path.StartsWithSegments("/Account/Register
"))
  {
    context.Response.Redirect("/Account/Login");
  }
  await next();
});
```

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Layout.cshtml:

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>@ViewData["Title"] - Historical Events Web
App</title>
    <link
href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tail
wind.min.css" rel="stylesheet">
    <script
src="https://cdn.jsdelivr.net/npm/alpinejs@3.10.2/dist/cdn.min.
js" defer></script>
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <style>
        .fade-in {
            animation: fadeIn 0.5s ease-in-out;
        }

        @@keyframes fadeIn {
            0% {
                opacity: 0;
                transform: translateY(10px);
            }

            100% {
                opacity: 1;
                transform: translateY(0);
            }
        }

        .card-hover:hover {
            transform: scale(1.03);
            transition: transform 0.3s ease-in-out;
            box-shadow: 0 10px 15px rgba(0, 0, 0, 0.1);
        }
        /* Градієнт для хедера */
        .header-gradient {
            background: linear-gradient(90deg, #4f46e5 0%,
#4338ca 100%);
        }
        /* Стиль для x-cloak */
        [x-cloak] {
            display: none;
        }
    </style>
</head>
<body class="bg-gray-100">
    <header class="header-gradient text-white shadow-lg">
        <nav class="container mx-auto px-4 py-5 flex justify-
between items-center">
            <a class="text-3xl font-extrabold tracking-tight
hover:text-indigo-200 transition" href="/">Historical
Events</a>
            <div class="space-x-6">
                <a class="text-lg font-extrabold tracking-tight
hover:text-indigo-200 hover:underline transition"
href="/Events">Події</a>
                <a class="text-lg font-extrabold tracking-tight
hover:text-indigo-200 hover:underline transition"
href="/Events/Graph">Граф зв’язків</a>
                @if (User.Identity?.IsAuthenticated ?? false)
                {
                    <form asp-controller="Account" asp-
action="Logout" method="post" class="inline">
                        <button type="submit" class="text-lg font-
extrabold tracking-tight hover:text-indigo-200 hover:underline
transition">Вийти (@User.Identity.Name)</button>
                    </form>
                }
                else
                {
                    <a class="text-lg font-extrabold tracking-tight
hover:text-indigo-200 hover:underline transition" asp-
controller="Account" asp-action="Login">Увійти</a>
                    <a class="text-lg font-extrabold tracking-tight
hover:text-indigo-200 hover:underline transition" asp-
controller="Account" asp-action="Register">Реєстрація</a>
                }
            </div>
        </nav>
    </header>

    <div class="container mx-auto px-4 py-8">
        <main role="main">
            @RenderBody()
        </main>
    </div>

    <footer class="bg-indigo-600 text-white py-4 mt-8">
        <div class="container mx-auto px-4 text-center">
            <p class="text-lg font-extrabold tracking-tight


hover:text-indigo-200 transition">
                © 2025 - Historical Events Web App
            </p>
        </div>
    </footer>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

EventsController.cs:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using HistoricalEventsAppNew.Services;
using HistoricalEventsAppNew.Models;
using HistoricalEventsAppNew.ViewModels;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using HistoricalEventsAppNew.Data;

namespace HistoricalEventsAppNew.Controllers
{
    [Authorize]
    public class EventsController : Controller
    {
        private readonly DatabaseService _databaseService;
        private readonly IWebHostEnvironment _environment;
        private readonly UserManager<IdentityUser>
_userManager;
```

```
    private readonly ApplicationDbContext _context;

    public EventsController(DatabaseService databaseService,
IWebHostEnvironment environment,
UserManager<IdentityUser> userManager,
ApplicationDbContext context)
    {
        _databaseService = databaseService;
        _environment = environment;
        _userManager = userManager;
        _context = context;
    }

    public async Task<IActionResult> Index(string
searchTitle, string filterTitle, DateTime? filterDate, string
filterCategory, string filterLocation, string filterPerson)
    {
        Console.WriteLine("Loading Index action for Events.");

        ViewBag.SearchTitle = searchTitle;
        ViewBag.FilterTitle = filterTitle;
        ViewBag.FilterDate = filterDate;
        ViewBag.FilterCategory = filterCategory;
        ViewBag.FilterLocation = filterLocation;
        ViewBag.FilterPerson = filterPerson;

        ViewBag.UniqueTitles = await
_databaseService.GetUniqueTitlesAsync();
        ViewBag.UniqueDates = await
_databaseService.GetUniqueDatesAsync();
        ViewBag.UniqueCategories = await
_databaseService.GetUniqueCategoriesAsync();
        ViewBag.UniqueLocations = await
_databaseService.GetUniqueLocationsAsync();
        ViewBag.UniquePersons = await
_databaseService.GetUniquePersonsAsync();

        var events = await
_databaseService.GetFilteredHistoricalEventsAsync(
            searchTitle, filterTitle, filterDate, filterCategory,
filterLocation, filterPerson);

        var eventRelations = new Dictionary<int,
List<HistoricalEventRelation>>();
        foreach (var evt in events)
        {
            var relations = await
_databaseService.GetEventRelationsAsync(evt.Id);
            eventRelations[evt.Id] = relations;
        }
        ViewBag.EventRelations = eventRelations;

        Console.WriteLine($"Retrieved {events.Count}
historical events.");
        return View(events);
    }

    public IActionResult Create()
    {
        Console.WriteLine("Loading Create action for new
HistoricalEvent.");
        return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(HistoricalEvent
historicalEvent, IFormFile imageFile)
    {
        Console.WriteLine("Processing Create action for
HistoricalEvent.");
        Console.WriteLine($"Received model:
Id={historicalEvent.Id}, Title={historicalEvent.Title},
Date={historicalEvent.Date}, " +
            $"Category={historicalEvent.Category},
Location={historicalEvent.Location},
Person={historicalEvent.Person}, " +
            $"MediaResource={historicalEvent.MediaResource},
ImageFile={(imageFile != null ? imageFile.FileName :
"null")}");

        historicalEvent.Id = 0;

        var user = await _userManager.GetUserAsync(User);
        historicalEvent.UserId = user.Id;

        if (ModelState.ContainsKey("UserId"))
        {
            ModelState.Remove("UserId");
        }

        if (imageFile != null && imageFile.Length > 0)
        {
            var allowedExtensions = new[] { ".jpg", ".jpeg",
".png", ".gif" };
            var extension =
Path.GetExtension(imageFile.FileName).ToLower();
            if (!allowedExtensions.Contains(extension))
            {
                Console.WriteLine($"Invalid file extension:
{extension}. Allowed extensions: {string.Join(", ",
allowedExtensions)}");
                ModelState.AddModelError("imageFile",
"Дозволені формати зображень: JPG, JPEG, PNG, GIF.");
            }
            else
            {
                try
                {
                    var fileName = Guid.NewGuid().ToString() +
extension;
                    var filePath =
Path.Combine(_environment.WebRootPath, "uploads",
fileName);
                    using (var stream = new FileStream(filePath,
FileMode.Create))
                    {
                        await imageFile.CopyToAsync(stream);
                    }
                    historicalEvent.ImagePath =
$"/uploads/{fileName}";
                    Console.WriteLine($"Saved image to
{filePath}, ImagePath set to {historicalEvent.ImagePath}.");
                    if (ModelState.ContainsKey("ImagePath"))
                    {
                        ModelState.Remove("ImagePath");
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error saving image file:
{ex.Message}");
                    ModelState.AddModelError("imageFile",
"Помилка при збереженні зображення.");
                }
            }
        }
        else
        {
```

```
            Console.WriteLine("No image file provided.
Proceeding without image.");
            historicalEvent.ImagePath = null;
            if (ModelState.ContainsKey("ImagePath"))
            {
                ModelState.Remove("ImagePath");
            }
            if (ModelState.ContainsKey("imageFile"))
            {
                ModelState.Remove("imageFile");
            }
        }

        if (!ModelState.IsValid)
        {
            Console.WriteLine("ModelState is invalid.
Validation errors:");
            foreach (var error in
ModelState.Values.SelectMany(v => v.Errors))
            {
                Console.WriteLine($"Error:
{error.ErrorMessage}");
            }
            var viewModel = new HistoricalEventViewModel
            {
                Id = historicalEvent.Id,
                Title = historicalEvent.Title,
                Date = historicalEvent.Date,
                Category = historicalEvent.Category,
                Location = historicalEvent.Location,
                Person = historicalEvent.Person,
                MediaResource = historicalEvent.MediaResource,
                ImagePath = historicalEvent.ImagePath,
                UserId = historicalEvent.UserId,
                RelatedEventTitles = new List<string>()
            };
            return View(viewModel);
        }

        try
        {
            Console.WriteLine("Attempting to create
HistoricalEvent in database.");
            await
_databaseService.CreateHistoricalEventAsync(historicalEvent)
;
            Console.WriteLine($"Successfully created
HistoricalEvent with Id={historicalEvent.Id}.");
            return RedirectToAction(nameof(Index));
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error creating HistoricalEvent:
{ex.Message}");
            var viewModel = new HistoricalEventViewModel
            {
                Id = historicalEvent.Id,
                Title = historicalEvent.Title,
                Date = historicalEvent.Date,
                Category = historicalEvent.Category,
                Location = historicalEvent.Location,
                Person = historicalEvent.Person,
                MediaResource = historicalEvent.MediaResource,
                ImagePath = historicalEvent.ImagePath,
                UserId = historicalEvent.UserId,
                RelatedEventTitles = new List<string>()
            };
            return View(viewModel);
        }
    }

    public async Task<IActionResult> Details(int? id)
    {
        Console.WriteLine($"Loading Details action for
HistoricalEvent with Id={id}.");
        if (id == null)
        {
            Console.WriteLine("Id is null, returning
NotFound.");
            return NotFound();
        }
        var historicalEvent = await
_databaseService.GetHistoricalEventByIdAsync(id.Value);
        if (historicalEvent == null)
        {
            Console.WriteLine($"HistoricalEvent with Id={id}
not found, returning NotFound.");
            return NotFound();
        }

        var relations = await
_databaseService.GetEventRelationsAsync(id.Value);
        ViewBag.EventRelations = relations;

        Console.WriteLine($"Found HistoricalEvent:
Id={historicalEvent.Id}, Title={historicalEvent.Title}.");
        return View(historicalEvent);
    }

    public async Task<IActionResult> Edit(int? id)
    {
        Console.WriteLine($"Loading Edit action for
HistoricalEvent with Id={id}.");
        if (id == null)
        {
            Console.WriteLine("Id is null, returning
NotFound.");
            return NotFound();
        }

        var historicalEvent = await
_databaseService.GetHistoricalEventByIdAsync(id.Value);
        if (historicalEvent == null)
        {
            Console.WriteLine($"HistoricalEvent with Id={id}
not found, returning NotFound.");
            return NotFound();
        }

        var user = await _userManager.GetUserAsync(User);
        if (!User.IsInRole("Admin") && historicalEvent.UserId
!= user.Id)
        {
            Console.WriteLine($"User {user.Id} is not
authorized to edit event with Id={id}.");
            return Forbid();
        }

        Console.WriteLine($"Found HistoricalEvent for edit:
Id={historicalEvent.Id}, Title={historicalEvent.Title}.");
        return View(historicalEvent);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(int id,
HistoricalEvent historicalEvent, IFormFile imageFile)
    {
        Console.WriteLine($"Processing Edit action for
HistoricalEvent with Id={id}.");
```

```
        Console.WriteLine($"Received model:
Id={historicalEvent.Id}, Title={historicalEvent.Title},
Date={historicalEvent.Date}, " +
            $"Category={historicalEvent.Category},
Location={historicalEvent.Location},
Person={historicalEvent.Person}, " +
            $"MediaResource={historicalEvent.MediaResource},
ImageFile={(imageFile != null ? imageFile.FileName :
"null")}");

        if (id != historicalEvent.Id)
        {
            Console.WriteLine($"Id mismatch: Route Id={id}
does not match model Id={historicalEvent.Id}, returning
NotFound.");
            return NotFound();
        }

        var user = await _userManager.GetUserAsync(User);
        var existingEvent = await
_databaseService.GetHistoricalEventByIdAsync(id);
        if (!User.IsInRole("Admin") && existingEvent.UserId
!= user.Id)
        {
            Console.WriteLine($"User {user.Id} is not
authorized to edit event with Id={id}.");
            return Forbid();
        }

        historicalEvent.UserId = existingEvent.UserId;

        if (ModelState.ContainsKey("UserId"))
        {
            ModelState.Remove("UserId");
        }

        if (imageFile != null && imageFile.Length > 0)
        {
            var allowedExtensions = new[] { ".jpg", ".jpeg",
".png", ".gif" };
            var extension =
Path.GetExtension(imageFile.FileName).ToLower();
            if (!allowedExtensions.Contains(extension))
            {
                Console.WriteLine($"Invalid file extension:
{extension}. Allowed extensions: {string.Join(", ",
allowedExtensions)}");
                ModelState.AddModelError("imageFile",
"Дозволені формати зображень: JPG, JPEG, PNG, GIF.");
            }
            else
            {
                try
                {
                    var fileName = Guid.NewGuid().ToString() +
extension;
                    var filePath =
Path.Combine(_environment.WebRootPath, "uploads",
fileName);
                    using (var stream = new FileStream(filePath,
FileMode.Create))
                    {
                        await imageFile.CopyToAsync(stream);
                    }
                    historicalEvent.ImagePath =
$"/uploads/{fileName}";
                    Console.WriteLine($"Saved new image to
{filePath}, ImagePath set to {historicalEvent.ImagePath}.");
                    if (ModelState.ContainsKey("ImagePath"))
                    {
                        ModelState.Remove("ImagePath");
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error saving image file:
{ex.Message}");
                    ModelState.AddModelError("imageFile",
"Помилка при збереженні зображення.");
                }
            }
        }
        else
        {
            Console.WriteLine("No new image file provided.
Keeping existing ImagePath.");
            historicalEvent.ImagePath =
existingEvent.ImagePath;
            if (ModelState.ContainsKey("ImagePath"))
            {
                ModelState.Remove("ImagePath");
            }
            if (ModelState.ContainsKey("imageFile"))
            {
                ModelState.Remove("imageFile");
            }
        }

        if (!ModelState.IsValid)
        {
            Console.WriteLine("ModelState is invalid.
Validation errors:");
            foreach (var error in
ModelState.Values.SelectMany(v => v.Errors))
            {
                Console.WriteLine($"Error:
{error.ErrorMessage}");
            }
            var relatedEventTitles = await
_databaseService.GetRelatedEventTitlesAsync(id);
            var viewModel = new HistoricalEventViewModel
            {
                Id = historicalEvent.Id,
                Title = historicalEvent.Title,
                Date = historicalEvent.Date,
                Category = historicalEvent.Category,
                Location = historicalEvent.Location,
                Person = historicalEvent.Person,
                MediaResource = historicalEvent.MediaResource,
                ImagePath = historicalEvent.ImagePath,
                UserId = historicalEvent.UserId,
                RelatedEventTitles = relatedEventTitles
            };
            return View(viewModel);
        }

        try
        {
            Console.WriteLine($"Attempting to update
HistoricalEvent with Id={historicalEvent.Id}.");
            await
_databaseService.UpdateHistoricalEventAsync(historicalEvent
);
            Console.WriteLine($"Successfully updated
HistoricalEvent with Id={historicalEvent.Id}.");
            return RedirectToAction(nameof(Index));
        }
        catch (Exception ex)
        {
```

```
            Console.WriteLine($"Error updating HistoricalEvent:
{ex.Message}");
            var relatedEventTitles = await
_databaseService.GetRelatedEventTitlesAsync(id);
            var viewModel = new HistoricalEventViewModel
            {
                Id = historicalEvent.Id,
                Title = historicalEvent.Title,
                Date = historicalEvent.Date,
                Category = historicalEvent.Category,
                Location = historicalEvent.Location,
                Person = historicalEvent.Person,
                MediaResource = historicalEvent.MediaResource,
                ImagePath = historicalEvent.ImagePath,
                UserId = historicalEvent.UserId,
                RelatedEventTitles = relatedEventTitles
            };
            return View(viewModel);
        }
    }

    public async Task<IActionResult> Delete(int? id)
    {
        Console.WriteLine($"Loading Delete action for
HistoricalEvent with Id={id}.");
        if (id == null)
        {
            Console.WriteLine("Id is null, returning
NotFound.");
            return NotFound();
        }

        var historicalEvent = await
_databaseService.GetHistoricalEventByIdAsync(id.Value);
        if (historicalEvent == null)
        {
            Console.WriteLine($"HistoricalEvent with Id={id}
not found, returning NotFound.");
            return NotFound();
        }

        var user = await _userManager.GetUserAsync(User);
        if (!User.IsInRole("Admin") && historicalEvent.UserId
!= user.Id)
        {
            Console.WriteLine($"User {user.Id} is not
authorized to delete event with Id={id}.");
            return Forbid();
        }

        Console.WriteLine($"Found HistoricalEvent for delete:
Id={historicalEvent.Id}, Title={historicalEvent.Title}.");
        return View(historicalEvent);
    }

    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> DeleteConfirmed(int
id)
    {
        Console.WriteLine($"Processing Delete action for
HistoricalEvent with Id={id}.");

        var historicalEvent = await
_databaseService.GetHistoricalEventByIdAsync(id);
        if (historicalEvent == null)
        {
            Console.WriteLine($"HistoricalEvent with Id={id}
not found.");
            return NotFound();
        }

        var user = await _userManager.GetUserAsync(User);
        if (!User.IsInRole("Admin") && historicalEvent.UserId
!= user.Id)
        {
            Console.WriteLine($"User {user.Id} is not
authorized to delete event with Id={id}.");
            return Forbid();
        }

        try
        {
            Console.WriteLine($"Attempting to delete
HistoricalEvent with Id={id}.");
            await
_databaseService.DeleteHistoricalEventAsync(id);
            Console.WriteLine($"Successfully deleted
HistoricalEvent with Id={id}.");
            return RedirectToAction(nameof(Index));
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error deleting HistoricalEvent:
{ex.Message}");
            return RedirectToAction(nameof(Index));
        }
    }

    public async Task<IActionResult> CreateRelation()
    {
        Console.WriteLine("Loading CreateRelation action.");
        ViewBag.Events = await
_databaseService.GetAllHistoricalEventsAsync();
        Console.WriteLine($"Retrieved
{((List<HistoricalEvent>)ViewBag.Events)?.Count ?? 0}
events for relation creation.");
        return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult>
CreateRelation(HistoricalEventRelation relation)
    {
        Console.WriteLine("Processing CreateRelation
action.");
        Console.WriteLine($"Received relation model:
Id={relation.Id}, SourceEventId={relation.SourceEventId},
TargetEventId={relation.TargetEventId}");

        if (relation.SourceEventId == relation.TargetEventId)
        {
            Console.WriteLine("Validation error: SourceEventId
and TargetEventId cannot be the same.");
            ModelState.AddModelError("", "Вихідна подія і
пов’язана подія не можуть бути однаковими.");
            ViewBag.Events = await
_databaseService.GetAllHistoricalEventsAsync();
            Console.WriteLine($"Retrieved
{((List<HistoricalEvent>)ViewBag.Events)?.Count ?? 0}
events for relation creation after validation failure.");
            return View(relation);
        }

        var sourceEvent = await
_databaseService.GetHistoricalEventByIdAsync(relation.Sourc
eEventId);
```

```
        var targetEvent = await
_databaseService.GetHistoricalEventByIdAsync(relation.Targe
tEventId);

        if (sourceEvent == null)
        {
            Console.WriteLine($"Validation error: SourceEvent
with Id={relation.SourceEventId} does not exist.");
            ModelState.AddModelError("SourceEventId",
"Вибрана вихідна подія не існує.");
        }

        if (targetEvent == null)
        {
            Console.WriteLine($"Validation error: TargetEvent
with Id={relation.TargetEventId} does not exist.");
            ModelState.AddModelError("TargetEventId",
"Вибрана пов’язана подія не існує.");
        }

        if (ModelState.ContainsKey("SourceEvent"))
        {
            ModelState.Remove("SourceEvent");
        }
        if (ModelState.ContainsKey("TargetEvent"))
        {
            ModelState.Remove("TargetEvent");
        }

        if (!ModelState.IsValid)
        {
            Console.WriteLine("ModelState is invalid.
Validation errors:");
            foreach (var error in
ModelState.Values.SelectMany(v => v.Errors))
            {
                Console.WriteLine($"Error:
{error.ErrorMessage}");
            }
            ViewBag.Events = await
_databaseService.GetAllHistoricalEventsAsync();
            Console.WriteLine($"Retrieved
{((List<HistoricalEvent>)ViewBag.Events)?.Count ?? 0}
events for relation creation after validation failure.");
            return View(relation);
        }

        try
        {
            Console.WriteLine($"Attempting to create relation
between SourceEventId={relation.SourceEventId} and
TargetEventId={relation.TargetEventId}.");
            await
_databaseService.CreateHistoricalEventRelationAsync(relation
);
            Console.WriteLine($"Successfully created relation
(and reverse relation) between
SourceEventId={relation.SourceEventId} and
TargetEventId={relation.TargetEventId}.");
            return RedirectToAction(nameof(Index));
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error creating relation:
{ex.Message}");
            ViewBag.Events = await
_databaseService.GetAllHistoricalEventsAsync();
            Console.WriteLine($"Retrieved
{((List<HistoricalEvent>)ViewBag.Events)?.Count ?? 0}
events for relation creation after error.");
            return View(relation);
        }
    }

    public async Task<IActionResult> Graph()
    {
        var events = await
_databaseService.GetHistoricalEventsAsync();
        Console.WriteLine($"Retrieved {events.Count} events
for graph.");

        if (!events.Any())
        {
            Console.WriteLine("No events found for graph.");
        }

        var allRelations = new List<HistoricalEventRelation>();
        foreach (var evt in events)
        {
            var relations = await
_databaseService.GetEventRelationsAsync(evt.Id);
            allRelations.AddRange(relations);
        }

        var uniqueRelations = allRelations
            .GroupBy(r => new { Source =
Math.Min(r.SourceEventId, r.TargetEventId), Target =
Math.Max(r.SourceEventId, r.TargetEventId) })
            .Select(g => g.First())
            .ToList();

        var eventNodes = events.Select(e => new
EventNodeDTO
        {
            Id = e.Id,
            Title = e.Title,
            Category = e.Category
        }).ToList();

        var eventLinks = new List<EventLinkDTO>();
        foreach (var relation in uniqueRelations)
        {
            if (eventNodes.Any(n => n.Id ==
relation.SourceEventId) && eventNodes.Any(n => n.Id ==
relation.TargetEventId))
            {
                eventLinks.Add(new EventLinkDTO
                {
                    SourceEventId = relation.SourceEventId,
                    TargetEventId = relation.TargetEventId,
                    RelationType =
relation.RelationType.ToString()
                });
            }
            else
            {
                Console.WriteLine($"Skipping invalid link:
Source={relation.SourceEventId},
Target={relation.TargetEventId},
Type={relation.RelationType}");
            }
        }

        Console.WriteLine($"Prepared {eventNodes.Count}
nodes for graph:");
        foreach (var node in eventNodes)
        {
            Console.WriteLine($"Node: Id={node.Id},
Title={node.Title}, Category={node.Category}");
        }
```

```
            Console.WriteLine($"Prepared {eventLinks.Count}
links for graph:");
            foreach (var link in eventLinks)
            {
                Console.WriteLine($"Link:
Source={link.SourceEventId}, Target={link.TargetEventId},
Type={link.RelationType}");
            }

            ViewBag.EventNodes = eventNodes;
            ViewBag.EventLinks = eventLinks;

            return View();
        }
    }
}
```