Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

(повна назва)

Кафедра програмної інженерії

(повна назва)

**КВАЛІФІКАЦІЙНА РОБОТА**

**Пояснювальна записка**

рівень вищої освіти перший (бакалаврський)

Програмна система аналітики даних електронної комерції у реальному часі

(тема)

Виконав:

здобувач \_4\_ року навчання

групи ПЗПІ-21-6

Станіслав ТКАЧЕНКО \_

(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного\_ \_\_

забезпечення

(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія

(повна назва освітньої програми)

Керівник доц. Віктор КАУК

(посада, Власне ім’я, ПРІЗВИЩЕ)

Допускається до захисту

Зав. кафедри \_\_\_\_\_\_\_\_\_\_\_ \_\_ Кирило СМЕЛЯКОВ\_

(підпис) (Власне ім’я, ПРІЗВИЩЕ)

2025 р.Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-професійна

Освітня програма Програмна Інженерія

(шифр і назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри \_\_\_\_\_\_\_\_\_

(підпис)

« » 2025 р.

**ЗАВДАННЯ**

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ткаченко Станіславу Максимовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система аналітики даних електронної комерції у реальному часі

Затверджена наказом по університету від 19.05.2025р. № 397 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 24.06.2025 \_

3. Вихідні дані до роботи Розробити програмну системи для аналітики даних електронної комерції в реальному часі з використанням технологій .NET, ASP.NET Core, Angular, СУБД MS SQL Server, MongoDB.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проєктування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
| 1 | Аналіз предметної галузі | 23.05.2025 | *виконано* |
| 2 | Створення специфікації ПЗ | 26.05.2025 | *виконано* |
| 3 | Проєктування ПЗ | 27.05.2025 | *виконано* |
| 4 | Розробка ПЗ | 03.06.2025 | *виконано* |
| 5 | Тестування ПЗ | 10.06.2025 | *виконано* |
| 6 | Оформлення пояснювальної записки | 12.06.2025 | *виконано* |
| 7 | Підготовка презентації та доповіді | 17.06.2025 | *виконано* |
| 8 | Попередній захист | 20.06.2025 | *виконано* |
| 9 | Нормоконтроль, рецензування | 20.06.2025 | *виконано* |
| 10 | Здача роботи у електронний архів | 20.06.2025 | *виконано* |
| 11 | Допуск до захисту у зав. кафедри | 20.06.2025 | *виконано* |

Дата видачі завдання « 19 » « травня » 2025р.

Здобувач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник роботи \_\_\_\_доц. Віктор КАУК \_\_\_

(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

**РЕФЕРАТ**

Пояснювальна записка до кваліфікаційної роботи бакалавра, 76 стор.,  
13 рис., 2 табл., 18 джерел, 3 додатки.

АНАЛІТИКА, ВІЗУАЛІЗАЦІЯ, ДАНІ, ЕЛЕКТРОННА КОМЕРЦІЯ, ANGULAR, ASP.NET CORE, C#, ENTITY FRAMEWORK CORE, MICROSOFT SQL SERVER, MONGODB, REST API, SIGNALR

Об'єкт розробки – програмна система аналітики даних електронної комерції у реальному часі.

Мета розробки – створення клієнт-серверної системи, що забезпечує агрегацію даних з маркетплейсів та криптовалютних бірж, їх обробку, візуалізацію та трансляцію в реальному часі з використанням інструментів аналітики.

Метод рішення – середовище розробки Visual Studio Code та Visual Studio, мови програмування TypeScript і C#, фреймворки Angular та ASP.NET Core, бази даних MongoDB і Microsoft SQL Server.

У результаті розробки спроєктовано архітектуру системи, реалізовано серверну та клієнтську частину, забезпечено обробку великої кількості аналітичних даних у режимі реального часу, реалізовано механізми багатомовності, авторизації й адміністрування.

**ABSTRACT**

ANALYTICS, ANGULAR, ASP.NET CORE, C#, DATA, E-COMMERCE, ENTITY FRAMEWORK CORE, MICROSOFT SQL SERVER, MONGODB, REAL-TIME UPDATES, REST API, SIGNALR, VISUALIZATION

The Object of the development – a real-time data analytics system for e-commerce.

The Purpose of the development – to create a client-server system that provides data aggregation from marketplaces and cryptocurrency exchanges, processes and visualizes this data, and delivers real-time updates using analytical tools.

The Method of implementation – development environments Visual Studio Code and Visual Studio, programming languages TypeScript and C#, frameworks Angular and ASP.NET Core, databases MongoDB and Microsoft SQL Server.

As a result of the development, the system architecture was designed, both server-side and client-side parts were implemented, the processing of large volumes of analytical data in real time was ensured, and functionality for multilingual support, authorization, and administration was realized.

**ЗМІСТ**

[Вступ 8](#_Toc201422742)

[1 Аналіз предметної галузі 9](#_Toc201422743)

[1.1 Сучасний стан електронної комерції та потреба в аналітиці 9](#_Toc201422744)

[1.2 Класифікація та особливості даних у сфері електронної комерції 11](#_Toc201422745)

[1.3 Існуючі підходи до аналітики в e-commerce та їх обмеження 14](#_Toc201422746)

[1.4 Постановка завдання 17](#_Toc201422747)

[2 Формування вимог до програмної системи 18](#_Toc201422748)

[3 Архітектура та проектування програмного забезпечення 21](#_Toc201422749)

[3.1 UML проектування програмного забезпечення 21](#_Toc201422750)

[3.2 Проектування архітектури програмного забезпечення 24](#_Toc201422751)

[3.2.1 Загальна архітектура програмної системи 24](#_Toc201422752)

[3.2.2 Архітектура серверної частини 26](#_Toc201422753)

[3.2.3 Архітектура клієнтської частини 28](#_Toc201422754)

[3.3 Проектування структури зберігання даних 29](#_Toc201422755)

[3.4 Приклади найцікавіших алгоритмів та методів 31](#_Toc201422756)

[3.5 UX-дизайн системи 32](#_Toc201422757)

[4 Опис прийнятих програмних рішень 35](#_Toc201422758)

[4.1 Реалізація аутентифікації та авторизації 35](#_Toc201422759)

[4.2 Розширена підтримка багатомовності 36](#_Toc201422760)

[4.3 Підтвердження входу за допомогою електронної пошти 40](#_Toc201422761)

[4.4 Використання Mongo DB для збереження історичних даних статистики 42](#_Toc201422762)

[4.5 Використання SignalR для оновлень аналітичних даних у реальному часі 44](#_Toc201422763)

[4.6 Реалізація панелі адміністратора 46](#_Toc201422764)

[4.7 REST-інтерфейс серверної частини 49](#_Toc201422765)

[5 Тестування програмного забезпечення 51](#_Toc201422766)

[Висновки 54](#_Toc201422767)

[Перелік джерел посилання 56](#_Toc201422768)

[Додаток А. Результат перевірки на унікальність тексту у базі ХНУРЕ 58](#_Toc201422769)

[Додаток Б. Фрагменти програмного коду 59](#_Toc201422770)

[Додаток В. Слайди презентації 69](#_Toc201422771)

# Вступ

Сучасна електронна комерція функціонує в умовах безперервного потоку даних, де швидкість обробки інформації та миттєва реакція на події стають критичними факторами конкурентоспроможності. Традиційні архітектури з періодичним оновленням даних не відповідають вимогам ринку, де аналіз поведінки користувачів, моніторинг продажів та виявлення аномалій повинні відбуватися в режимі реального часу.

Актуальність розробки полягає у створенні високонавантаженої системи, здатної обробляти сотні подій щосекунди та надавати аналітичні дані без затримок. Метою проєкту є реалізація повноцінної архітектури для збору, обробки та візуалізації даних електронної комерції з використанням сучасного технологічного стеку.

Об'єктом дослідження виступає процес агрегації та аналізу даних з різних маркетплейсів у реальному часі. Предметом є архітектурні рішення та технології для забезпечення двосторонньої комунікації між сервером та клієнтом.

Технічна реалізація системи базується на наступних компонентах:

- серверна частина: ASP.NET Core API з використанням SignalR для двосторонньої комунікації, Entity Framework Core для роботи з даними, та BackgroundService для регулярної агрегації інформації з зовнішніх джерел;

- клієнтська частина: Angular фреймворк з RxJS для реактивної обробки потоків даних та Chart.js для динамічної візуалізації аналітики;

- комунікаційний шар: WebSocket протокол через SignalR хаби, що забезпечує миттєву передачу оновлень від сервера до клієнта.

Система спроєктована з дотриманням принципів масштабованості, модульності та відмовостійкості, що дозволяє легко інтегрувати нові джерела даних та розширювати функціональність без суттєвих змін в архітектурі.

# Аналіз предметної галузі

## Сучасний стан електронної комерції та потреба в аналітиці

Сучасна електронна комерція опинилася на порозі системної трансформації. Попит на миттєве реагування, динамічні ціни, персоналізовану комунікацію і прозорість ланцюгів постачання вже не є опцією – це обов’язкова умова виживання. У той час як ринок дедалі більше орієнтується на швидкість у межах мілісекунд, внутрішні процеси багатьох компаній досі залишаються прив’язаними до нічних пакетних завантажень, застарілих BI-систем і відкладеної реакції. Це породжує численні проблеми, які стримують розвиток, зменшують конкурентоспроможність і призводять до втрати клієнтів.

Утім зростання e-commerce відбувається нерівномірно і в різних регіонах має свої особливості. В Україні, попри війну та економічні труднощі, ринок демонструє стійку динаміку. Згідно з дослідженням Promodo, у 2024 році частка онлайн-торгівлі в загальному обсязі роздрібних продажів сягнула близько 10 % [1]. Для порівняння, в усьому світі цей показник становив приблизно 19,4 % [2], а в розвинених країнах – до 50 %. Водночас загальний онлайн-оборот українців у 2024-му зріс на 25 % – до 239 млрд грн [1]. Середній чек становив близько 1 300 грн, а кількість активних онлайн-покупців перевищила 11 млн людей, які в середньому здійснювали по 17 покупок на рік [1]. Високий рівень проникнення інтернету (понад 90 % населення, 78 % користуються ним щоденно [1]) супроводжується поступовим освоєнням нових каналів продажу: вже 93 % українських торговців приймають електронні платежі й працюють через маркетплейси та соціальні мережі [3].

Нижче подано узагальнену картину викликів та стратегії реагування, що актуальні як для глобального, так і для українського e-commerce.

Ціни змінюються ледь не щосекунди, і великі гравці автоматизовано коригують їх мільйони разів на день. Amazon, наприклад, щодня оновлює понад два з половиною мільйони цін, завдяки чому миттєво адаптується до попиту та активності конкурентів. Ті ж, хто працює з нічними пакетними оновленнями, або продають занадто дешево, втрачаючи маржу, або надто дорого, втрачаючи покупців.

Сучасний клієнт взаємодіє з брендами через сайт, мобільний додаток, месенджери й соцмережі. Близько 73 % українців робили покупки через кілька каналів [1], а у глобальному масштабі цей відсоток сягає 80 % [4]. Кожна взаємодія генерує подію, яку потрібно обробити та інтегрувати в єдину картину поведінки. Без real-time аналітики дані приходять фрагментами, що веде до втрати консистенції й погіршує користувацький досвід.

Персоналізовані пропозиції підвищують конверсію, однак 82 % споживачів готові поділитися даними лише за умови відчутної вигоди, а 83 % ставлять захист приватності в топ-пріоритет [5]. Виконання GDPR/DSA вимагає гнучких рішень, які зможуть сегментувати користувачів у режимі реального часу, не порушивши при цьому їхніх прав.

Автоматизовані атаки здатні створювати сотні підробних транзакцій за хвилини. Понад 40 % e-commerce проектів називають ризик-сценарії причиною затримки запуску нових каналів [6]. Антифрод-системи з batch-перевірками не встигають зупинити зловмисників, тому лише рішення з миттєвим скануванням транзакцій і рисків на основі ML можуть ефективно захистити бізнес.

Попит на спеціалістів із Kafka, Flink, Spark та edge-аналітики перевищує пропозицію. 90 % українських ритейлерів відчувають брак таких фахівців [7]. Без них впровадження real-time платформ затягується, а проєкти стрімко втрачають підтримку керівництва через затримки в отриманні вигоди.

Обробка мільйонів подій щосекунди вимагає робочих вузлів, лічильників API і розширеного зберігання. За оцінками IDC, до 2028 року на edge- та stream-обчислення буде витрачено близько 378 млрд USD [8]. Компаніям доводиться балансувати між продуктивністю й витратами.

Обсяг світового e-commerce значно перевищив шість трильйонів доларів і зростатиме до восьми трильйонів до кінця десятиліття [8]. У США онлайн-виторг на 2024 рік перевищив 1,2 трлн USD [9], а китайські платформи JD.com та Alibaba утримують двозначні темпи зростання [10]. Лідером динаміки є Латинська Америка, де GMV Mercado Libre щороку зростає понад 30 % [11].

Мобільна комерція стала пріоритетом: смартфони забезпечують понад половину глобальних транзакцій [8], а в Україні цей показник перевищив 50 % [1]. Омніканальні покупці витрачають до 30 % більше, ніж ті, хто користується лише одним каналом [4].

Класичні BI платформи (Google Analytics, Looker, Power BI) пропонують зручні дашборди, але працюють із затримкою в годину або добу. У світі, де ціни змінюються щохвилини, таке відставання неприйнятне.

Потокові технології (Kafka, Flink, Spark Streaming) забезпечують мілісекундну латентність, але потребують значних інвестицій у DevOps, інженерів-стрімерів і складну інтеграцію з існуючими CRM, ERP та SCM системами. Без комплексного підходу користь від них обмежується кількома відсотками економії.

Гібридні архітектури поєднують batch та stream: критичні ланки (рекомендації, antifraud, трекінг) переводяться на стрім, решта залишається у BI. Це зменшує ризики, але не вирішує проблему фрагментації даних і синхронності.

Затримка між кліком користувача і рішенням системи у мілісекундах проти години чи доби коштує мільйони доларів. Електронна комерція в Україні та світі стоїть перед дилемою: залишитися в інерційних пакетних процесах чи вкластися в real-time платформи, здатні миттєво реагувати на виклики ринку. Перша стратегія веде до поступової втрати конкурентності; друга – до лідерства на ринку майбутнього.

## Класифікація та особливості даних у сфері електронної комерції

У сфері електронної комерції дані є ключовим активом, що визначає ефективність бізнес-процесів, рівень персоналізації та конкурентоспроможність компаній. Розуміння класифікації та особливостей даних дозволяє підприємствам оптимізувати свою діяльність, приймати обґрунтовані рішення та забезпечувати високий рівень обслуговування клієнтів.

Дані в електронній комерції можна умовно поділити на кілька основних категорій. Перша з них – це дані про клієнтів, які включають особисту інформацію, поведінкові патерни, історію покупок та взаємодію з платформою. Ці дані дозволяють створювати детальні профілі споживачів, що є основою для персоналізованих пропозицій та маркетингових кампаній. Наприклад, аналізуючи історію переглядів і покупок, компанія може рекомендувати товари, які найбільше відповідають інтересам конкретного користувача.

Друга категорія – це дані про товари, що охоплюють інформацію про назви, описи, характеристики, ціни, наявність на складі та відгуки покупців. Ці дані є фундаментом для організації каталогу продукції, забезпечення зручної навігації та пошуку на сайті. Правильна класифікація товарів за категоріями та атрибутами сприяє покращенню користувацького досвіду та збільшенню конверсії.

Третя категорія – транзакційні дані, які відображають фінансові операції між клієнтами та компанією. Вони включають інформацію про суми покупок, способи оплати, дати та час транзакцій. Аналіз цих даних дозволяє відстежувати фінансові показники, виявляти тенденції у продажах та вчасно реагувати на зміни в поведінці споживачів.

Четверта категорія – поведінкові дані, що фіксують дії користувачів на платформі: перегляди сторінок, кліки, час перебування на сайті, шляхи навігації. Ці дані є основою для аналізу користувацького досвіду, виявлення проблемних місць у воронці продажів та оптимізації інтерфейсу.

П’ята категорія – дані зворотного зв’язку, які включають відгуки, рейтинги, коментарі та звернення до служби підтримки. Вони надають цінну інформацію про задоволеність клієнтів, якість товарів і послуг, а також допомагають виявляти області для покращення.

Особливості даних в електронній комерції визначаються їх обсягом, швидкістю оновлення, різноманітністю та необхідністю забезпечення високої якості. Зростання обсягу даних вимагає використання сучасних технологій зберігання та обробки, таких як хмарні сервіси та системи управління великими даними. Швидкість оновлення даних є критичною для забезпечення актуальності інформації про наявність товарів, ціни та статус замовлень.

Різноманітність даних обумовлює необхідність інтеграції структурованих і неструктурованих джерел, таких як бази даних, соціальні мережі, відгуки та інші. Це вимагає використання інструментів для обробки текстів, зображень та інших типів даних. Забезпечення високої якості даних передбачає впровадження процесів очищення, нормалізації та валідації, що дозволяє уникнути помилок і забезпечити точність аналітики.

Крім того, важливо враховувати аспекти безпеки та конфіденційності даних. Захист персональної інформації клієнтів є пріоритетом для компаній, що працюють в електронній комерції. Впровадження стандартів безпеки, таких як шифрування даних, двофакторна аутентифікація та відповідність нормативним вимогам, забезпечує довіру споживачів і знижує ризики витоку інформації.

У сучасних умовах особливого значення набуває використання аналітики в реальному часі. Це дозволяє компаніям оперативно реагувати на зміни в поведінці споживачів, виявляти шахрайські дії, оптимізувати запаси та ціноутворення. Впровадження систем стрім-аналітики, таких як Apache Kafka та Apache Flink, забезпечує обробку великих потоків даних з мінімальною затримкою, що є критичним для конкурентоспроможності бізнесу.

Також варто відзначити роль машинного навчання та штучного інтелекту в обробці та аналізі даних. Ці технології дозволяють автоматизувати процеси класифікації, сегментації клієнтів, прогнозування попиту та виявлення аномалій. Наприклад, алгоритми машинного навчання можуть аналізувати поведінкові патерни користувачів і на основі цього формувати персоналізовані рекомендації, що підвищує ймовірність покупки.

У підсумку, ефективне управління даними в електронній комерції є ключовим фактором успіху. Розуміння класифікації та особливостей даних дозволяє компаніям приймати обґрунтовані рішення, підвищувати ефективність бізнес-процесів, забезпечувати високий рівень обслуговування клієнтів та залишатися конкурентоспроможними на ринку.

## Існуючі підходи до аналітики в e-commerce та їх обмеження

Аналітика даних в електронній комерції є основою для прийняття стратегічних рішень, підвищення ефективності бізнесу та покращення клієнтського досвіду. Вона дозволяє виявити сильні й слабкі сторони бізнес-процесів, оцінити динаміку ринку, адаптувати маркетингові кампанії і забезпечити персоналізовану взаємодію зі споживачами. Залежно від цілей, сучасна аналітика поділяється на чотири основні типи: описову, діагностичну, прогнозну та прескриптивну.

Описова аналітика – найпоширеніший тип, який відображає те, що вже відбулося. У e-commerce вона використовується для звітності щодо продажів, поведінки користувачів на сайті, джерел трафіку тощо. Інструменти як Google Analytics 4, Power BI, Metabase надають візуалізацію таких даних через діаграми, таблиці, дашборди. Наприклад, у GA4 можна відстежити кількість кліків, переглядів товарів, транзакцій і провести базову сегментацію. Проте значним обмеженням є затримка оновлення: обробка статистики займає до 48 годин. Крім того, описова аналітика не пояснює причин змін і не дозволяє прогнозувати події.

Діагностична аналітика зосереджується на виявленні причин змін. Вона відповідає на питання “чому” – наприклад, чому впала конверсія на мобільному або чому зросла кількість відмов на конкретному товарі. Тут активно застосовуються порівняльний аналіз періодів, кореляція, drill-down по сегментах у Power BI або SQL-запити до сховищ. Діагностична аналітика вимагає якісного зведення даних із CRM, ERP, веб-аналітики, що створює додаткові труднощі через різнорідність джерел. Наприклад, у CRM-системі SalesDrive можна відслідковувати втрати на кожному етапі воронки, але без інтеграції з BI-системою повної картини не буде.

Прогнозна аналітика використовує історичні дані, математичні моделі та машинне навчання для передбачення майбутніх подій. Вона дозволяє будувати прогнози попиту, доходів, клієнтської активності. У GA4 вже реалізовані прості прогнозні функції – наприклад, оцінка ймовірності покупки або відтоку користувача. Українська KeyCRM інтегрує прогнози прибутків на основі наявних замовлень, а Putler для маркетплейсів (Etsy, eBay, Shopify) дозволяє передбачати майбутні продажі. Недоліком цього підходу є чутливість до якості вхідних даних і висока складність побудови моделей – особливо для малого бізнесу без доступу до фахівців з Data Science.

Прескриптивна аналітика – найрозвиненіший рівень, який не лише прогнозує майбутнє, а й пропонує конкретні дії: змінити ціну, змінити бюджет реклами або персоналізувати пропозицію. Це область застосування оптимізаційних алгоритмів, штучного інтелекту та рекомендаторів. Такі можливості реалізовані в Amazon чи Netflix, проте в українських рішеннях поки представлені слабо. Лише деякі продукти, наприклад, той самий KeyCRM, містять модулі, що рекомендують прибуткові канали чи товари для просування, але повноцінна оптимізація залишається недоступною для більшості компаній.

Разом з цим, аналітика e-commerce має низку обмежень, які ускладнюють повноцінне використання даних. Насамперед, затримка та неточність. Усі основні аналітичні системи мають latency: навіть в GA4 повні метрики можуть бути доступні лише через добу-дві, BI-системи (Power BI Free) обмежуються 8 оновленнями на день. Це означає, що рішення приймаються не на «живих» даних, а на зібраній раніше картині – це особливо критично в періоди акцій, коливань попиту або технічних збоїв.

Друга проблема – відсутність повного реального часу. Більшість інструментів декларують «real time», проте в дійсності показують лише окремі події (перегляди, кліки), а не повні сегменти або зведення по воронках. Наприклад, Putler дозволяє бачити нові замовлення моментально, але не об’єднує їх миттєво з витратами чи кампаніями.

Третє – інтеграція та агрегування даних. Аналітика передбачає об’єднання десятків джерел: веб, склад, логістика, фінанси, реклама. Але кожна система має свій формат – наприклад, OLX не дає розгорнутих подій по замовленнях, а маркетплейси типу Prom звітують по-різному. У результаті компанії витрачають багато часу на експорт-імпорт, скрипти або ручні зведення в Excel, що знижує точність і швидкість.

Четверте – брак персоналізації. Хоча більшість CRM і BI мають аналітичні дашборди, вони часто обмежуються середніми значеннями (середній чек, конверсія тощо). Детальні моделі на рівні клієнтів чи SKU практично відсутні. Для індивідуального підходу (рекомендації, динамічна ціна) потрібні додаткові дані з поведінки, вподобань, купівель, а також моделі, які здатні це обробляти. GA4 та SalesDrive цього не надають – без інтеграції з ML-системами неможливо впровадити інтелектуальний email-маркетинг або автоматичні push-нагадування.

П’яте – низька автоматизація. Більшість компаній формують щоденні звіти вручну: або в Power BI, або в Google Sheets. Автоматичні оповіщення про зміни (алерти), самонавчальні системи або auto-pilot сегментація – радше виняток, ніж правило. У дослідженнях зазначається, що менше третини українських компаній використовують стандартизовані BI-системи з єдиною дашборд-архітектурою.

Приклади впроваджень дають уявлення, як аналітика допомагає світовим брендам оптимізувати процеси. Walmart використовує real-time anomaly detection для контролю цін: коли виявляється невідповідність, система автоматично сигналізує адміністратору – це скоротило втрати до 15% designveloper.com. Amazon впровадив ML-моделі для передбачення попиту й оптимізації складів – це дало змогу мінімізувати витрати на логістику та зменшити обсяги списаного товару appinventiv.com. У свою чергу, Shopify надає базову аналітику своїм клієнтам, але для точнішої інформації користувачі змушені інтегрувати сторонні рішення (Klaviyo, GA4, Metabase).

Отже, хоча аналітика в e-commerce має велику кількість інструментів, майже всі з них мають суттєві обмеження щодо швидкості, гнучкості, глибини аналізу та автоматизації. Для справжнього цифрового прориву компаніям потрібно інвестувати у сучасні стрім-платформи, розвивати аналітику на основі AI, інтегрувати дані з усіх систем та забезпечити доступ до аналітики у реальному часі. Лише так можна відповідати на виклики «економіки мілісекунд», у якій оперує сучасна електронна комерція.

## Постановка завдання

У ході дослідження предметної області аналітики електронної комерції в реальному часі встановлено, що наявні інструментів аналітики мають суттєві обмеження щодо швидкості обробки даних, гнучкості налаштування, глибини аналізу та рівня автоматизації. Для ефективного реагування на динамічні зміни ринку та запити користувачів, компанії потребують сучасних рішень, здатних обробляти та візуалізувати дані у режимі реального часу.

Основним завданням є розробка програмної системи аналітики даних електронної комерції в реальному часі. Особлива увага приділяється технологічним, архітектурним та інтеграційним викликам, що виникають під час створення платформи для збору, обробки та візуалізації інформації з декількох джерел – зокрема маркетплейсів (DummyJSON, FakeStore) та криптовалютних бірж (CoinGecko API).

У рамках проєкту передбачається створення веб-застосунку, який поєднує такі підходи:

* реактивне програмування для обробки потоків даних у реальному часі;
* двосторонню комунікацію між клієнтом і сервером за допомогою SignalR;
* інтерактивну візуалізацію даних на клієнтській частині.

Реалізація проєкту складається з наступних етапів:

* аналіз предметної області та поточних викликів аналітики електронної комерції в реальному часі;
* формування функціональних та нефункціональних вимог до системи;
* розробка архітектури веб-застосунку та проєктування його компонентів;
* створення серверної частини на базі ASP.NET Core з використанням SignalR та фонових сервісів;
* розробка клієнтського інтерфейсу на Angular з інтерактивними дашбордами;
* тестування, оцінка продуктивності та аналіз результатів.

# Формування вимог до програмної системи

Результатом розробки стане веб-орієнтована програмна система аналітики даних електронної комерції в реальному часі, яка включає серверну частину, реалізовану на базі ASP.NET Core, та клієнтську частину, створену за допомогою Angular. Серверна частина відповідатиме за отримання, обробку та агрегацію даних із зовнішніх джерел, а також за трансляцію оновленої інформації клієнтам. Клієнтська частина забезпечуватиме візуальне представлення даних у вигляді інтерактивних дашбордів і графіків.

У межах системи передбачено основну роль користувача – авторизований користувач (аналітик), який має доступ до інтерфейсу перегляду статистики, графіків та аналітичної інформації з оновленням у реальному часі. Також передбачено роль адміністратора системи, який має можливість змінювати мовні налаштування та керувати акаунтами користувачів. Також у системі присутній функціонал для неавторизованого користувача, який включатиме можливість реєстрації, авторизації та підтвердження пошти.

Функціональні вимоги до системи включають:

* агрегація даних з різних маркетплейсів: система повинна інтегруватися з API різних маркетплейсів (DummyJSON, FakeStore) для отримання інформації про товари, ціни та категорії. Дані мають оновлюватися регулярно через фонові сервісиж;
* моніторинг криптовалютних цін: система повинна збирати дані про ціни криптовалют через API CoinGecko, включаючи поточну ціну, ринкову капіталізацію та зміну ціни за 24 години;
* статистичний аналіз даних: система має агрегувати статистику по категоріях товарів, розраховуючи середню ціну та кількість товарів у кожній категорії;
* трансляція оновлень у реальному часі: система повинна забезпечувати доставку оновлень на клієнтську частину через SignalR без потреби оновлювати сторінку. Це стосується як товарної статистики, так і інформації про криптовалюти;
* візуалізація даних: клієнтський інтерфейс повинен містити інтерактивні графіки та діаграми для зручного перегляду аналітичної інформації. Для цього планується використання бібліотеки Chart.js.

Нефункціональні вимоги охоплюють архітектуру, продуктивність та безпеку системи:

* масштабованість та модульність: архітектура повинна підтримувати легке додавання нових джерел даних (маркетплейсів) через інтерфейс; система має бути спроектована з використанням принципу інверсії залежностей для забезпечення гнучкості та тестованості;
* продуктивність: система повинна обробляти дані з різних джерел та транслювати їх на клієнтську частину з мінімальною затримкою; фонові сервіси мають оновлювати дані кожні 30 секунд без впливу на продуктивність основних операцій;
* надійність комунікації: SignalR з автоматичним відновленням з'єднання для забезпечення стабільної комунікації між сервером та клієнтом навіть при тимчасових проблемах з мережею;
* кросплатформеність: система повинна працювати в різних браузерах та на різних пристроях, забезпечуючи однаковий користувацький досвід;
* CORS підтримка [12]: система має підтримувати Cross-Origin Resource Sharing для забезпечення безпечної взаємодії між різними доменами.

Технічна реалізація системи базується на сучасному стеку технологій, що охоплює як серверну, так і клієнтську частину.

Серверна частина реалізована з використанням ASP.NET Core [13], який забезпечує створення WebAPI та хостинг SignalR-хабів для організації двосторонньої комунікації в реальному часі. Для регулярного оновлення інформації із зовнішніх джерел використовуються фонові сервіси на базі BackgroundService. Управління залежностями та підвищення модульності реалізовано за допомогою вбудованого механізму Dependency Injection, а взаємодія з API зовнішніх сервісів здійснюється через HttpClient.

Клієнтська частина розробляється за допомогою фреймворку Angular [14], що дозволяє створювати динамічний та інтерактивний інтерфейс користувача. Для реактивної обробки потоків даних використовуються інструменти RxJS, зокрема BehaviorSubject та Observable. Отримання оновлень у реальному часі на клієнті реалізовано за допомогою SignalR-клієнта, який підключається до відповідних хабів на сервері. Візуалізація даних виконується за допомогою бібліотеки Chart.js, що дозволяє відображати інтерактивні графіки. Код клієнтської частини пишеться мовою TypeScript, що забезпечує типобезпеку та кращу підтримку при розробці.

Початковий випуск включає базовий функціонал, достатній для моніторингу стану товарів на маркетплейсах та динаміки криптовалютних цін, з інтеграцією у режимі реального часу. Передбачається, що в наступних версіях система може бути доповнена збереженням історичних даних, розширеною авторизацією, підтримкою нових джерел даних та більш складними інструментами аналітики (кластеризація, прогнозування тощо).

У підсумку, система реалізує повний цикл обробки даних електронної комерції: від збору інформації з різних джерел через її агрегацію та аналіз до візуалізації в реальному часі на клієнтській частині. Архітектура системи забезпечує модульність, масштабованість та високу продуктивність, що дозволяє легко розширювати функціональність та адаптуватися до зростаючих потреб бізнесу.

# Архітектура та проектування програмного забезпечення

## UML проектування програмного забезпечення

Аналітика даних електронної комерції у реальному часі – це процес збору, обробки та аналізу даних з онлайн-магазинів для миттєвого отримання інсайтів про продажі, поведінку користувачів та ефективність бізнесу. Система аналітики даних електронної комерції у реальному часі забезпечує моніторинг ключових показників, виявлення трендів та аномалій, а також надає інструменти для прийняття обґрунтованих бізнес-рішень.

Для формалізації вимог до програмної системи та кращого розуміння її функціонування на ранніх етапах розробки використовуються засоби UML-моделювання (Unified Modeling Language). UML-діаграми дозволяють наочно представити структуру, поведінку та взаємодію компонентів системи, що особливо важливо при командній роботі, технічному обґрунтуванні архітектурних рішень та документуванні проєкту.

У рамках проєкту створено дві основні UML-діаграми:

* діаграма варіантів використання (Use Case Diagram), яка ілюструє функціональність системи з точки зору користувача;
* діаграма розгортання (Deployment Diagram), яка описує фізичну інфраструктуру, на якій розгортається система, та взаємозв’язки між компонентами.

Діаграма варіантів використання моделює поведінку системи з позиції взаємодії користувачів із нею. У системі визначено три ролі. Неаутентифікований користувач має доступ до обмеженого функціоналу. Такий користувач може виконати реєстрацію в системі, здійснити вхід в акаунт; отримати підтвердження про успішний вхід (наприклад, через повідомлення або зміну інтерфейсу). Аутентифікований користувач, який отримує розширений функціонал після входу до системи. Він має можливість виконати вихід з акаунту, змінити мову інтерфейсу, переглянути загальну сторінку статистики, яка включає дані про світові бренди, ринок книжок, ринок їжі, магазин Fake Store та дані про криптовалютиа. Адміністратор системи має доступ до адмін-панелі з можливістю зміни мовних налаштувань, завантаження мовних файлів та керування аккаунтами користувачів.

Ця діаграма дозволяє наочно побачити, який функціонал доступний для кожної ролі та як змінюється доступ до функцій залежно від стану користувача (до/після аутентифікації). Діаграма варіантів використання наведена на рисунку 3.1

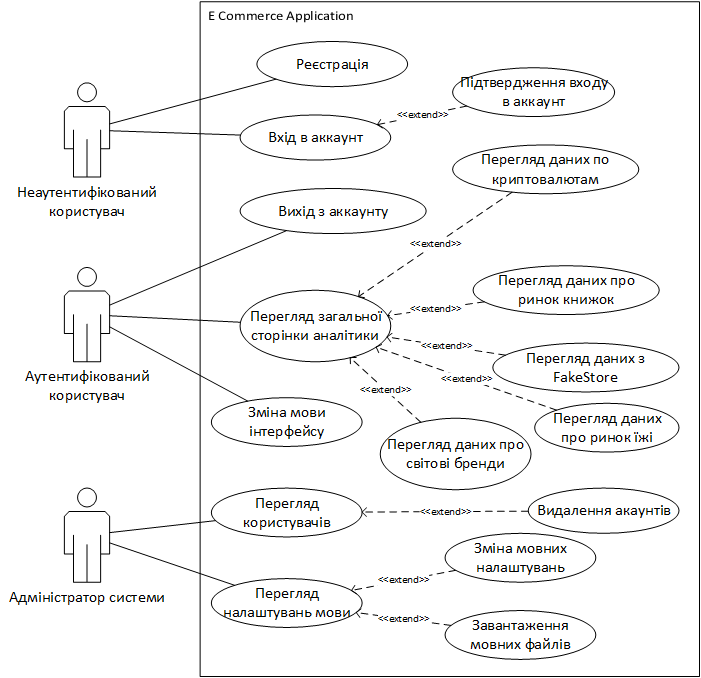


Рисунок 3.1 – UML діаграма варіантів використання системи для аналітики даних електронної комерції (рисунок виконано самостійно)

Діаграма розгортання (Deployment Diagram) використовується для візуалізації фізичної інфраструктури програмної системи – апаратних або віртуальних вузлів, на яких розгорнуті компоненти системи, а також взаємозв’язків між ними. Вона допомагає зрозуміти, як логічні елементи проєкту (модулі, служби, бази даних) реалізуються у вигляді фізичних об’єктів на реальних або хмарних серверах.

У розробленій системі задіяно кілька основних компонентів, які взаємодіють між собою:

* клієнтський застосунок розгортається як SPA (Single Page Application), реалізований на базі Angular; цей компонент запускається у веббраузері користувача та взаємодіє з сервером через HTTP(S)-запити та SignalR-з'єднання в реальному часі;
* серверна частина реалізована на ASP.NET Core, що обробляє запити клієнтів, управляє бізнес-логікою, взаємодіє з базами даних, виконує фонові завдання та виступає у ролі хосту для SignalR-хабів;
* база даних SQL Server використовується для зберігання структурованих даних;
* база даних MongoDB зберігає неструктуровані або слабо структуровані дані, зокрема агреговану інформацію з зовнішніх API, аналітичні дані, статистику;
* зовнішні API такі як DummyJSON, FakeStore (для отримання даних про товари) та CoinGecko (для збору інформації про криптовалюти).

Діаграма розгортання наведена на рисунку 3.2.

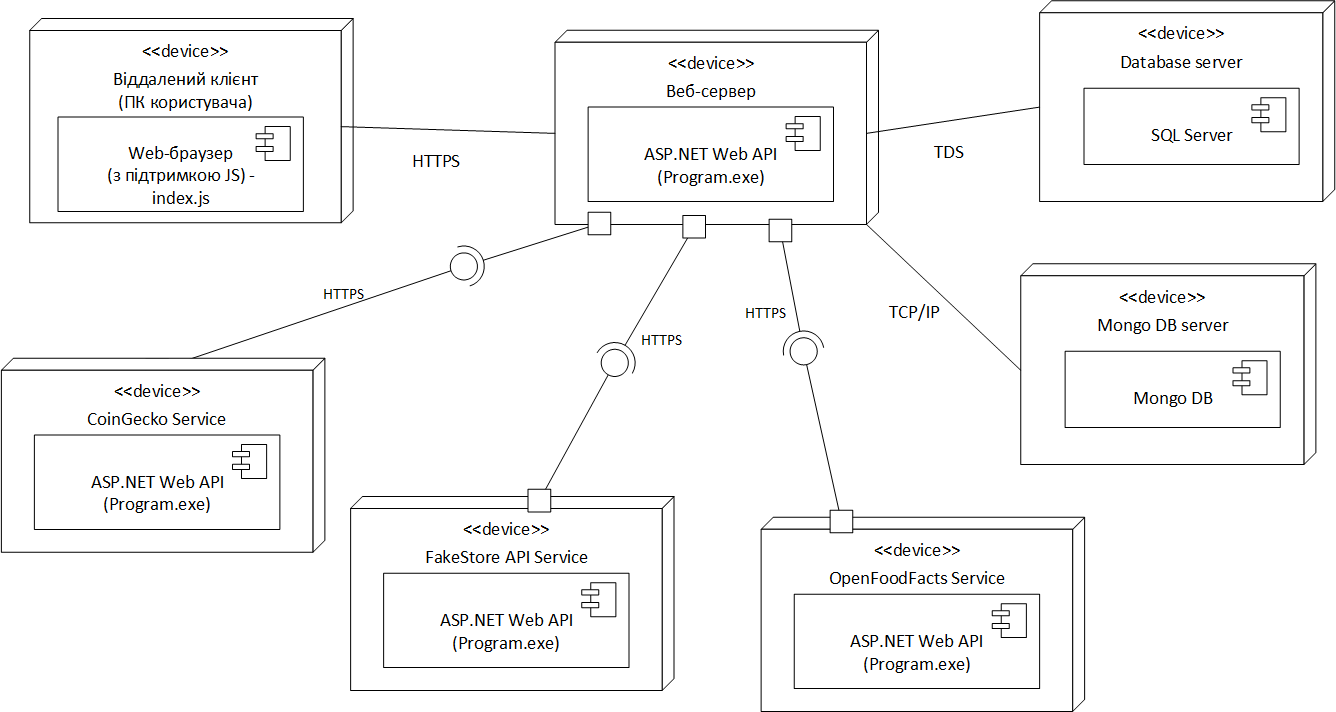


Рисунок 3.2 - Діаграма розгортання програмної системи для аналітики даних електронної комерції (рисунок виконано самостійно)

## Проектування архітектури програмного забезпечення

## Загальна архітектура програмної системи

У межах проєкту було розроблено багаторівневу (multitier) веб-архітектуру [15], що забезпечує розділення обов’язків між клієнтською частиною, серверною логікою, системою зберігання даних та зовнішніми джерелами. Такий підхід сприяє спрощенню супроводу системи, забезпечує можливість горизонтального масштабування та полегшує інтеграцію нових функцій у майбутньому.

Клієнтська частина системи реалізована у вигляді SPA (Single Page Application) за допомогою фреймворку Angular. Вона відповідає за взаємодію з користувачем: відображення інтерфейсу, побудову інтерактивних дашбордів, графіків, таблиць тощо. Angular дозволяє організувати гнучку компонентну структуру, підтримує реактивний підхід до роботи з даними завдяки RxJS і легко інтегрується з SignalR-клієнтом для роботи в реальному часі. Уся взаємодія клієнта з сервером відбувається через HTTP-запити до REST API або WebSocket-з’єднання для отримання живих оновлень.

Серверна частина побудована на базі ASP.NET Core Web API – сучасної, кросплатформної платформи від Microsoft для побудови високопродуктивних вебзастосунків. Сервер відповідає за реалізацію бізнес-логіки, обробку запитів від Angular-клієнта, взаємодію з базами даних та зовнішніми API.

Для зберігання даних у системі застосовуються дві різні бази даних, кожна з яких виконує свою роль згідно з принципами поліглотної персистентності. MS SQL Server використовується для зберігання критично важливих структурованих  
даних – інформації про користувачів, сесії, права доступу та інші реляційні сутності. Вона добре підходить для даних з чіткою схемою, складними запитами та забезпеченням транзакційності. Натомість MongoDB виступає як сховище для напівструктурованих або швидкозмінних даних, зокрема агрегованих результатів аналітики, отриманих із зовнішніх джерел, даних про товари, ціни, динаміку криптовалют тощо. Завдяки своїй гнучкій структурі та високій швидкодії при роботі з великими обсягами JSON-подібних об'єктів, MongoDB ідеально підходить для цієї задачі.

Зовнішні API (наприклад, CoinGecko, FakeStore, DummyJSON) інтегруються на стороні сервера. За допомогою HttpClient реалізовані сервіси, що запитують актуальну інформацію про товари, категорії та криптовалюти. Отримані дані обробляються та зберігаються в базах даних, після чого передаються клієнту або транслюються через SignalR у реальному часі.

Система реалізована за класичною клієнт-серверною архітектурною моделлю, яка забезпечує чітке розділення обов’язків між клієнтською частиною та серверною частиною. Такий підхід дозволяє досягнути високого рівня масштабованості, підтримки, повторного використання компонентів та безпеки. Клієнтська частина (Angular-застосунок) функціонує у веб-браузері користувача та виконує роль тонкого клієнта, відповідального за представлення інтерфейсу, обробку взаємодії з користувачем, ініціацію запитів до сервера та відображення отриманих результатів. Серверна частина, реалізована на ASP.NET Core Web API, виконує роль обробника запитів від клієнта. Клієнт-серверна модель у цій архітектурі є асинхронною, ефективною та масштабованою.

Уся архітектура побудована відповідно до принципів чистої архітектури та інверсії залежностей, що дозволяє ізолювати ядро логіки від інфраструктурних деталей, таких як робота з БД або API. Це значно полегшує тестування, супровід та подальший розвиток системи. Завдяки використанню Dependency Injection в ASP.NET Core, всі залежності між компонентами системи налаштовуються централізовано, що забезпечує гнучкість та розширюваність архітектури.

## Архітектура серверної частини

Серверна частина програмної системи реалізована на базі **ASP.NET Core Web API** з використанням принципів **чітко розмежованої відповідальності** (Separation of Concerns) та модульності [16]. Це дозволяє спростити підтримку, розширення та тестування коду, зменшує зв’язність між модулями та підвищує гнучкість системи. Архітектура умовно поділяється на кілька логічних шарів, кожен з яких має чітке призначення.

Вхідною точкою обробки запитів є папка Controllers, яка містить API-контролери. Вони відповідають за прийом HTTP-запитів, виклик відповідних сервісів та формування відповідей для клієнта. Контролери зосереджені лише на обробці запитів і не містять бізнес-логіки, що делегується відповідним сервісам.

Бізнес-логіка системи розміщена у папці Services. Тут реалізуються основні алгоритми, обробка даних, робота з базою, інтеграція із зовнішніми API та інші функції, необхідні для виконання завдань системи. Сервіси проектуються як незалежні модулі з чітко визначеними інтерфейсами, що сприяє гнучкості та можливості легкої заміни або тестування окремих частин.

Для підтримки комунікації у реальному часі в системі використовується технологія SignalR, реалізована в папці Hubs. Тут розміщуються SignalR-хаби, які забезпечують двонаправлений зв’язок між сервером і клієнтом, дозволяючи серверу відправляти оновлення даних без необхідності повторного запиту з боку користувача.

У папці Models зберігаються класи, що описують структуру даних, які використовуються в системі. Це як сутності, які відображають таблиці бази даних, так і DTO (Data Transfer Objects) для передачі інформації між клієнтом і сервером. Моделі є основою для формування об’єктів, що опрацьовуються на різних рівнях програми.

Папка Shared містить спільні компоненти та утиліти, які використовуються в різних частинах проєкту. Сюди входять базові класи, константи, допоміжні функції, розширення.

Відповідальність за взаємодію з реляційною базою даних SQL Server покладена на клас контексту бази даних, розташований у папці DatabaseContext. Цей клас, що наслідується від Entity Framework Core DbContext, забезпечує доступ до таблиць, керування підключеннями та конфігурацію моделей.

Для керування змінами структури бази даних у проєкті використовується папка Migrations, яка містить скрипти та метадані міграцій Entity Framework Core. Вона дозволяє автоматично оновлювати схему бази відповідно до змін у моделях, забезпечуючи безперервність роботи додатку. Структура серверної частини наведена на діаграмі пакетів (рисунок 3.3).

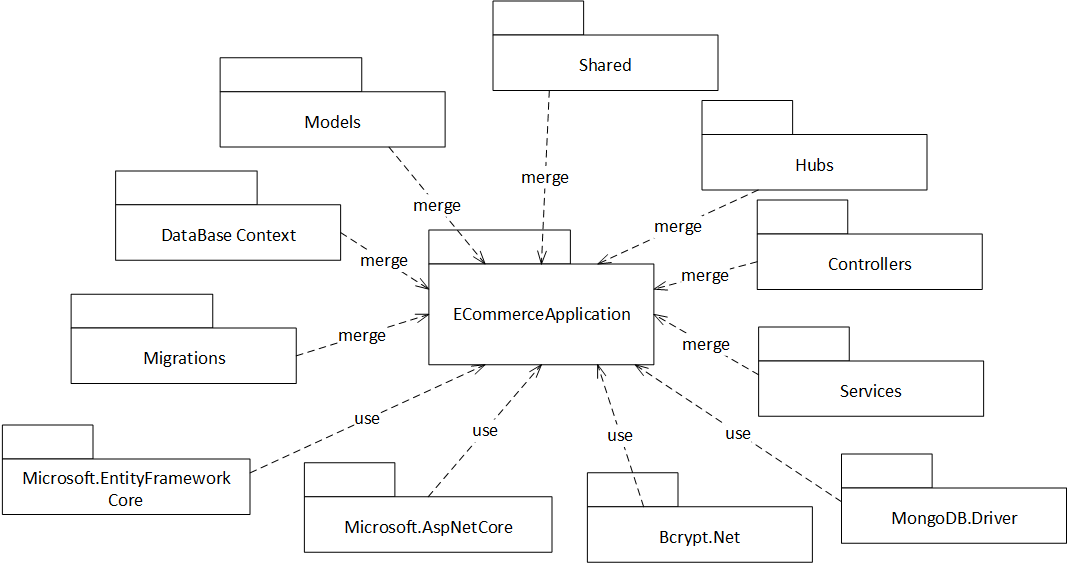


Рисунок 3.3 – Діаграма пакетів серверної частини (рисунок виконано самостійно)

У серверній частині проєкту для роботи з реляційною базою даних використовується **Entity Framework Core** – сучасний ORM-фреймворк від Microsoft, який забезпечує об’єктно-реляційне відображення та полегшує роботу з даними на рівні коду. EF Core дозволяє працювати з базою даних, оперуючи звичними для розробника об’єктами та класами замість написання складних SQL-запитів вручну.

Для опису структури таблиць та взаємозв’язків між ними використовується як **атрибутивне моделювання** у класах-сутностях, так і **Fluent API**, який надає гнучкі можливості конфігурації схеми бази безпосередньо у коді. Це дозволяє визначати ключі, обмеження, індекси, типи полів та інші характеристики структури даних.

У проєкті також використовується механізм **міграцій**, який є вбудованим інструментом EF Core для управління змінами схеми бази даних. Міграції дають змогу поетапно впроваджувати зміни у структуру таблиць (додавати нові колонки, змінювати типи, створювати індекси) без втрати наявних даних, забезпечуючи контроль версій бази.

## Архітектура клієнтської частини

Клієнтська частина системи розроблена із застосуванням сучасного фронтенд-фреймворку Angular, який базується на модульно-компонентній архітектурі. Цей підхід забезпечує чітке розділення інтерфейсу користувача на невеликі, повторно використовувані компоненти, а також логічне групування функціональності у модулі.

Компоненти в Angular – це основні будівельні блоки інтерфейсу. Кожен компонент інкапсулює власну розмітку (HTML), стилі (CSS/SCSS) і логіку (TypeScript), що дозволяє створювати незалежні та ізольовані частини UI. Такий поділ підвищує масштабованість проекту, спрощує підтримку та розвиток, оскільки зміни у одному компоненті мінімально впливають на інші.

Модулі в Angular слугують для організації та групування компонентів, сервісів та інших ресурсів у логічні частини. Вони полегшують керування залежностями та налаштування маршрутизації, а також сприяють оптимізації завантаження додатку через механізм лейзі-лоадингу (lazy loading).

Структура клієнтської частини проекту чітко відображає цей підхід. Папка components містить UI-компоненти, що відповідають за візуальне представлення і взаємодію користувача. У папці services розміщені сервіси, які реалізують бізнес-логіку, роботу з API та обробку даних, надаючи компонетам необхідні функції.

Папка models містить визначення типів і інтерфейсів для роботи з даними, що підвищує типобезпечність коду завдяки використанню TypeScript. В enums зберігаються перелічувальні типи, що забезпечують зрозумілість і підтримуваність логіки.

Interceptors – це спеціальні класи, які перехоплюють HTTP-запити та відповіді, що дозволяє додавати заголовки, обробляти помилки чи впроваджувати механізми авторизації централізовано.

Папка modules організовує основні функціональні блоки додатку у вигляді Angular-модулів, що покращує структуру проєкту та сприяє повторному використанню коду.

Спільні ресурси, такі як загальні компоненти, директиви та пайпи, розташовані в папці shared, що дозволяє легко їх імпортувати у різні частини додатку.

Загальні стилі, які використовуються по всьому додатку, знаходяться в папці styles, що забезпечує єдиний стиль і візуальну консистентність інтерфейсу.

## Проектування структури зберігання даних

Архітектура зберігання даних у створеній системі поєднує два різні типи сховищ – нереляційне (MongoDB) та реляційне (Microsoft SQL Server). Такий підхід обумовлений потребами проєкту: з одного боку – необхідністю гнучкого зберігання агрегованих аналітичних даних, які динамічно змінюються, з іншого – потребою у збереженні структурованої інформації про користувачів, локалізації та вхід до системи.

Основним сховищем аналітичної інформації у системі є **MongoDB**, що належить до категорії документно-орієнтованих баз даних NoSQL [17]. MongoDB ідеально підходить для зберігання напівструктурованих або динамічних даних, що швидко змінюються, особливо в контексті збору статистики з маркетплейсів та криптовалютних бірж.

У MongoDB зберігаються **агреговані об’єкти**, які відповідають структурі класу ProductDto. Кожен документ містить інформацію про товар або запис аналітики у вигляді колекції об’єктів з полями Title, Description, Price, Category, ImageUrl, Source.

Цей формат дозволяє зберігати агреговані значення, отримані з різних джерел (таких як DummyJSON, FakeStore або CoinGecko), у єдиній структурі. Поле Source вказує на джерело даних і представлено у вигляді перелічувального типу enum. Гнучкість MongoDB дозволяє адаптувати схему до змін у структурі даних без необхідності модифікації всієї БД, що є суттєвою перевагою при роботі з зовнішніми API.

Паралельно з MongoDB у системі використовується **Microsoft SQL Server** – класична реляційна СУБД, яка відповідає за зберігання інформації, що потребує суворої структурованості, транзакційності та забезпечення цілісності. У SQL Server зберігаються наступні таблиці:

Таблиця Users містить базову інформацію про зареєстрованих користувачів:

* Id – унікальний ідентифікатор користувача;
* Email, FirstName, LastName – основні дані користувача;
* PasswordHash – хешований пароль для безпечної автентифікації;
* Role – роль у системі (наприклад, адміністратор або звичайний користувач);
* IsDelete – логічне видалення користувача.

Таблиця Logins використовується для підтвердження входу користувачів через електронну пошту:

* Id – ідентифікатор запису;
* Email – електронна адреса користувача;
* Date – дата та час спроби входу;
* Code – одноразовий код підтвердження;
* Ip, OS, UserAgent – додаткова інформація про клієнтське середовище;
* IsCompleted – ознака завершеності входу.

Таблиця Languages зберігає мовні налаштування інтерфейсу та переклади.

* Id – ідентифікатор мови;
* LanguageName, LanguageCode – назва та код мови (наприклад, en, uk);
* TranslationJson – JSON-рядок із ключами та відповідними перекладами, що дозволяє швидко реалізовувати мультимовність;
* Version – версія набору перекладів, для підтримки оновлень.

## Приклади найцікавіших алгоритмів та методів

У системі реалізовано механізм отримання аналітики у режимі реального часу, який дозволяє миттєво оновлювати дані на клієнтській частині без необхідності ручного перезавантаження сторінки. Цей функціонал реалізовано за допомогою технології SignalR, що забезпечує двосторонню комунікацію між сервером та браузером.

Процес відбувається таким чином: у фоновому режимі працює спеціальний сервіс, який з певним інтервалом надсилає запити до зовнішніх API: CoinGecko для отримання криптовалютних курсів та маркетплейсів DummyJSON/FakeStore для товарних даних. Після отримання нових значень, ці дані обробляються на сервері – обчислюється необхідна статистика, оновлюються агрегати, здійснюється перевірка, чи є зміни порівняно з попереднім станом.

Якщо зміни дійсно виявлені, система ініціює трансляцію повідомлення через SignalR-хаб. Сервер викликає метод на клієнтському підключенні, передаючи оновлений набір даних або лише ті значення, що змінилися. Angular-клієнт, який підписаний на відповідний канал SignalR, миттєво отримує повідомлення, після чого оновлює відповідні графіки, діаграми або таблиці.

Таким чином, користувач отримує актуальну аналітику в реальному часі, не взаємодіючи з інтерфейсом вручну. Такий підхід особливо ефективний у динамічних доменах, як-от фінансові ринки, електронна торгівля чи відстеження попиту, де навіть незначна затримка в оновленні даних може вплинути на прийняття рішень. Цей алгоритм поєднує фонову агрегацію, розумне оновлення (тільки за потреби), та миттєву доставку інформації до кінцевого користувача.

## UX-дизайн системи

У процесі UI/UX проєктування системи було реалізовано низку основних інтерфейсів, орієнтованих на зручність користувача, швидкий доступ до аналітичних даних і простоту взаємодії. Дизайн кожного з екранів відповідає сучасним принципам адаптивного інтерфейсу та логічної структури елементів.

Першою реалізованою сторінкою стала сторінка авторизації, яка передбачає введення електронної пошти, підтвердження входу через одноразовий код та індикацію етапів входу. Інтерфейс побудований з урахуванням мінімалізму – користувач бачить лише необхідні поля та повідомлення про статус операції   
(рис. 3.4).

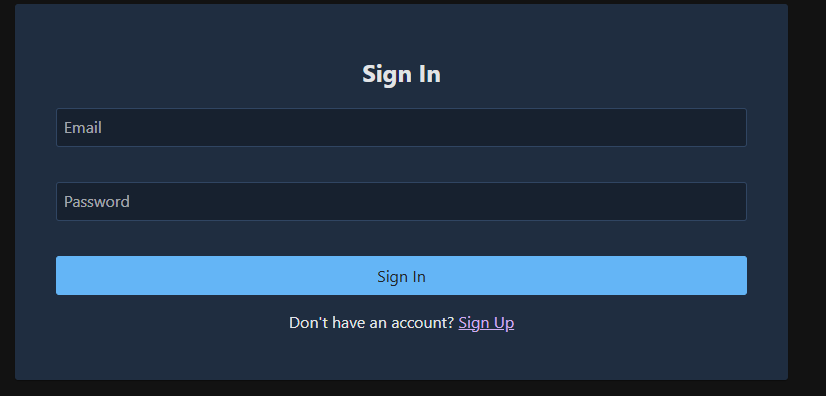


Рисунок 3.4 – Сторінка авторизації (рисунок виконано самостійно)

Основна увага була приділена сторінкам з аналітичними графіками, які реалізовані у вигляді дашбордів. Дані подаються через діаграми й графіки, що оновлюються в режимі реального часу. Для візуалізації використовуються інтерактивні елементи, що дозволяють користувачам зручно працювати з великим обсягом даних. Зверху сторінки знаходяться вкладки для переключенні між сторінками аналітики (рис. 3.5).

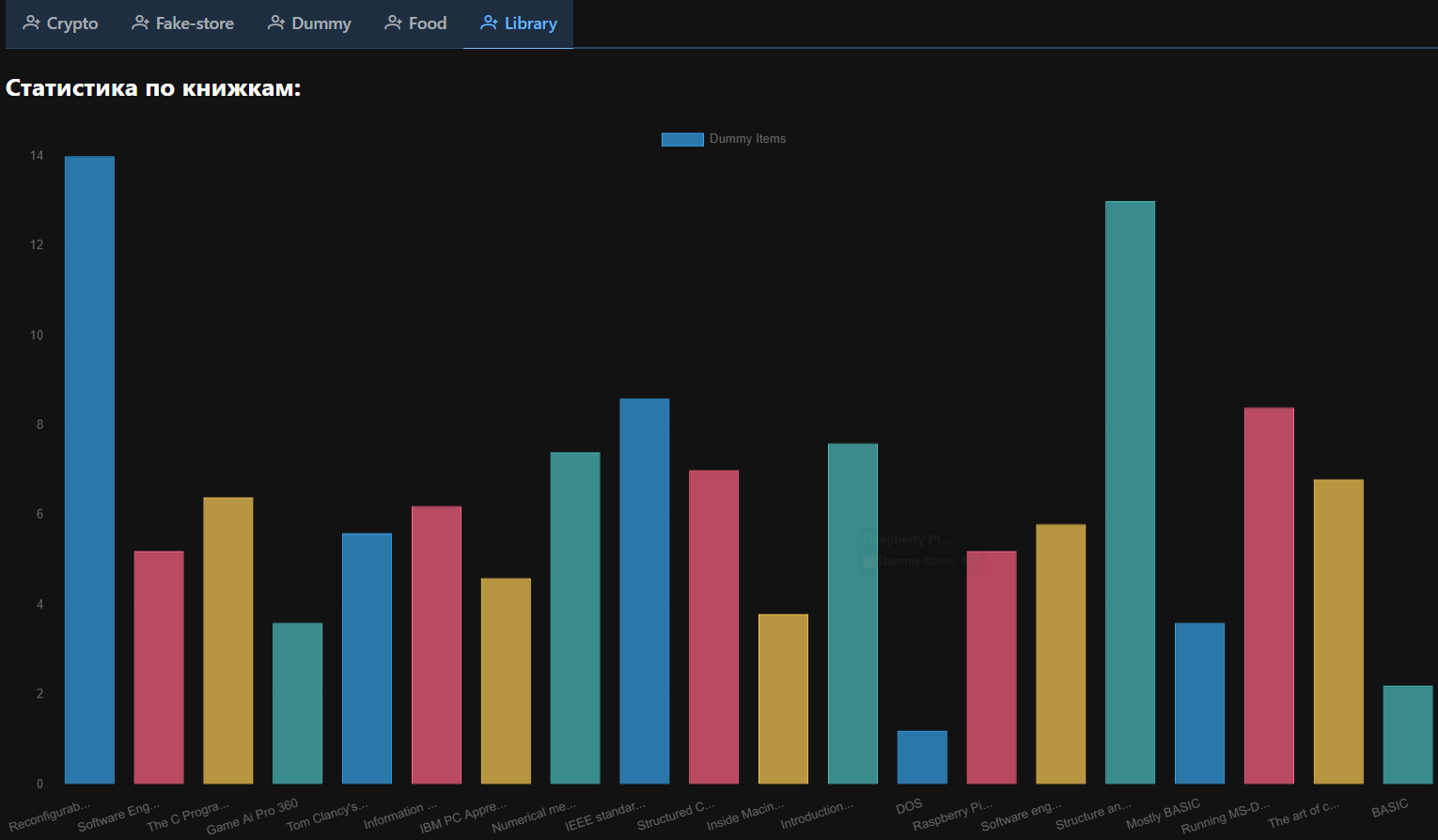


Рисунок 3.5 – Сторінка з аналітичними даними (рисунок виконано самостійно)

Сторінка профілю реалізує базовий набір функцій – перегляд особистої інформації, зміна паролю та особистої інформації. Всі дії доступні у кілька кліків, структура елементів відповідає очікуванням користувача (рис. 3.6).

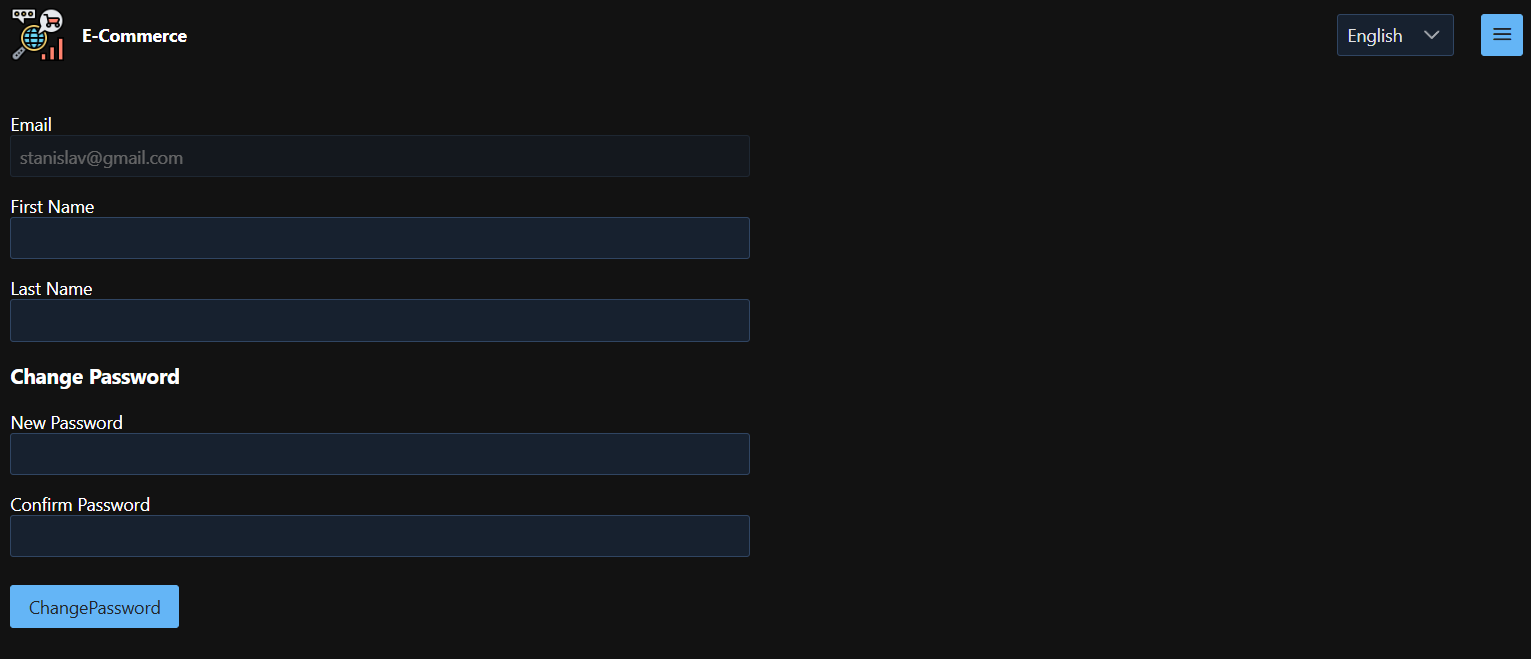


Рисунок 3.6 – Сторінка профілю користувача (рисунок виконано самостійно)

Усі компоненти мають уніфікований стиль, який задається через глобальні стилі, розміщені у відповідній папці. Колірна палітра, розміри елементів та відступи були підібрані з урахуванням візуальної ієрархії та доступності. Іконки, анімації завантаження та інші мікровзаємодії покликані підвищити зрозумілість інтерфейсу без перевантаження користувача.

Для забезпечення адаптивності, верстка була реалізована з урахуванням підтримки різних розмірів екранів, що дозволяє користувачам працювати з системою як на комп’ютерах, так і на планшетах.

# Опис прийнятих програмних рішень

## Реалізація аутентифікації та авторизації

У реалізованій системі для забезпечення захисту доступу до приватних ресурсів використовується механізм **JSON Web Tokens**, що дозволяє надійно ідентифікувати користувача після входу та зберігати інформацію про його роль.

На сервері аутентифікація реалізується шляхом генерації JWT токена після підтвердження електронної пошти користувача. Токен містить інформацію про ідентифікатор користувача (NameIdentifier) та його роль у системі (Role), яка надалі використовується для обмеження доступу до певних маршрутів API.

Функція для генерації JWT токену на сервері:

private string GenerateJwtToken(string email, string role)

{

var claims = new[]

{

new Claim(ClaimTypes.NameIdentifier, email),

new Claim(ClaimTypes.Role, role)

};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

\_config["Jwt:Issuer"],

\_config["Jwt:Audience"],

claims,

expires: DateTime.UtcNow.AddHours(2),

signingCredentials: creds

);

return new JwtSecurityTokenHandler().WriteToken(token);

}

Для обробки запитів з токеном реалізовано проміжне ПЗ (JwtMiddleware), яке зчитує токен з заголовку Authorization, перевіряє його дійсність та, у випадку успішної валідації, прикріплює користувача до HttpContext.

Фрагмент middleware:

var token = context.Request.Headers["Authorization"]

.FirstOrDefault()?.Split(" ").Last();

if (token != null)

await AttachUserToContext(context, userService, token);

Таким чином, усі подальші дії в контролерах можуть виконуватись з урахуванням ролі користувача.

На клієнтській частині реалізовано сервіс AuthService, який відповідає за логіку входу, реєстрації, перевірки коду та управління JWT токеном у localStorage. Після успішного підтвердження коду, токен зберігається локально, а його декодування дозволяє отримати email користувача та роль.

Збереження токену на клієнті:

tap(response => {

localStorage.setItem(this.tokenKey, JSON.stringify(response.token));

this.isLoggedInSubject.next(true);

})

Для перевірки автентифікації використовуються функції isAuthenticated() та isTokenExpired(), які дозволяють контролювати термін дії токена. Додатково, методи getUserRole() та getUserEmail() витягують відповідні поля з payload токена за допомогою jwt-decode.

## Розширена підтримка багатомовності

У системі реалізовано повноцінну підтримку багатомовності, що охоплює як зберігання мовних пакетів, так і динамічне завантаження перекладів для інтерфейсу користувача. Архітектура реалізована таким чином, щоб адміністратор міг додавати або оновлювати мовні пакети без необхідності перекомпіляції застосунку.

На серверній частині зберігання мов реалізовано у вигляді окремої таблиці Languages у базі даних SQL Server. Кожен мовний запис містить код мови, назву мови, версію та JSON-рядок з перекладеними ключами:

{

"SignIn": "Увійти",

"SignUp": "Зареєструватися",

"UsersSetting": "Налаштування користувачів",

"LanguageSettings": "Налаштування мови",

"Email": "Електронна пошта",

"FirstName": "Ім’я",

"LastName": "Прізвище",

"Password": "Пароль",

"ChangePassword": "Змінити пароль",

"NewPassword": "Новий пароль",

"ConfirmPassword": "Підтвердити пароль",

"PasswordNotMatch": "Паролі не співпадають.",

"Version": "Версія"

…

}

Адміністратор системи через інтерфейс адміністрування може завантажувати нові мовні пакети або оновлювати наявні. При додаванні нової мови за основу береться шаблон у форматі JSON із директорії Template, а при оновленні система автоматично підвищує версію мовного пакету.

Клас LanguageService забезпечує логіку додавання, оновлення, кешування та отримання мовних пакетів. При кожному зверненні до мови спочатку перевіряється наявність її у кеші (TranslationCacheService), що дозволяє уникнути надлишкових запитів до бази даних. Якщо версія збережена у кеші не відповідає актуальній – система оновлює її автоматично.

Метод GetLanguageJsonHolder повертає об'єкт, який включає JSON з перекладами, код мови та її версію. Ці дані передаються клієнтській частині за допомогою звичайного HTTP-запиту.

Деякі методи LanguageService:

public LanguageJsonHolder GetLanguageJsonHolder(string code)

{

string countryCode = code.ToLower();

string key = \_translationCacheService.GetKey(TranslationType.Language, countryCode);

lock (\_translationCacheService.GetSyncObject(key))

{

var holder = TryGetValidCachedHolder(countryCode);

if (holder != null)

return holder;

if (HasLanguage(countryCode))

{

return SetLanguageInCache(countryCode);

}

var fallbackHolder = TryGetValidCachedHolder("en");

return fallbackHolder ?? SetLanguageInCache("en");

}

}

public LanguageModel GetLanguage(string code)

{

var key = \_translationCacheService.GetKey(TranslationType.LanguageModel, code);

lock (\_translationCacheService.GetSyncObject(key))

{

if (\_translationCacheService.Get(key) is LanguageModel languageModel && this.GetVersion(languageModel.LanguageCode).Equals(languageModel.Version))

{

return languageModel;

}

if (!this.HasLanguage(code))

{

code = this.DefaultLanguage;

}

var storedLang = \_context.Languages.Single(lng => EF.Functions.Like(lng.LanguageCode, $"%{code}%"));

var language = new LanguageModel

{

LanguageName = storedLang.LanguageName,

LanguageCode = storedLang.LanguageCode,

Version = storedLang.Version,

TranslationJson = storedLang.TranslationJson,

};

\_translationCacheService.Set(key, language);

return language;

}

}

На клієнтській частині використовується кастомний TranslateLoader, який реалізовано у вигляді класу DbTranslateLoader. Під час ініціалізації інтерфейсу або зміни мови, клієнт надсилає запит на сервер за відповідним мовним пакетом:

getTranslation(lang: string): Observable<any> {

return this.http.get<JsonHolder>(apiUrl +

'/api/language/getLanguageJsonHolder?code=' + lang)

.pipe(map(response => JSON.parse(response.json)));

}

Якщо відповідь успішна, JSON з перекладеними рядками підключається до механізму ngx-translate, який забезпечує динамічну заміну текстів на сторінці згідно з ключами. Вибрана мова також зберігається локально в localStorage для автоматичного завантаження при наступному запуску.

Завдяки такій структурі реалізації система забезпечує:

* повну гнучкість у додаванні нових мов без змін у коді;
* підтримку динамічного перемикання мови в інтерфейсі користувача;
* використання централізованого кешу для підвищення продуктивності;
* підтримку версій мовних пакетів для синхронізації між сервером та клієнтом.

Вибір мови у системі доступний у правому верхньому куті інтефрейсу. Список доступних мов оновлюється динамічно у випадку зміни мовних пакетів адміністратором (рис. 4.1).

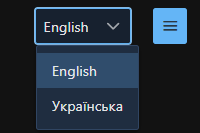


Рисунок 4.1 – Меню вибору мови (рисунок виконано самостійно)

Після зміни мови інтерфейсу усі елементи автоматично перекладаються. Приклад відображення українською та англійською мовою наведено на   
рисунках 4.2 – 4.3.

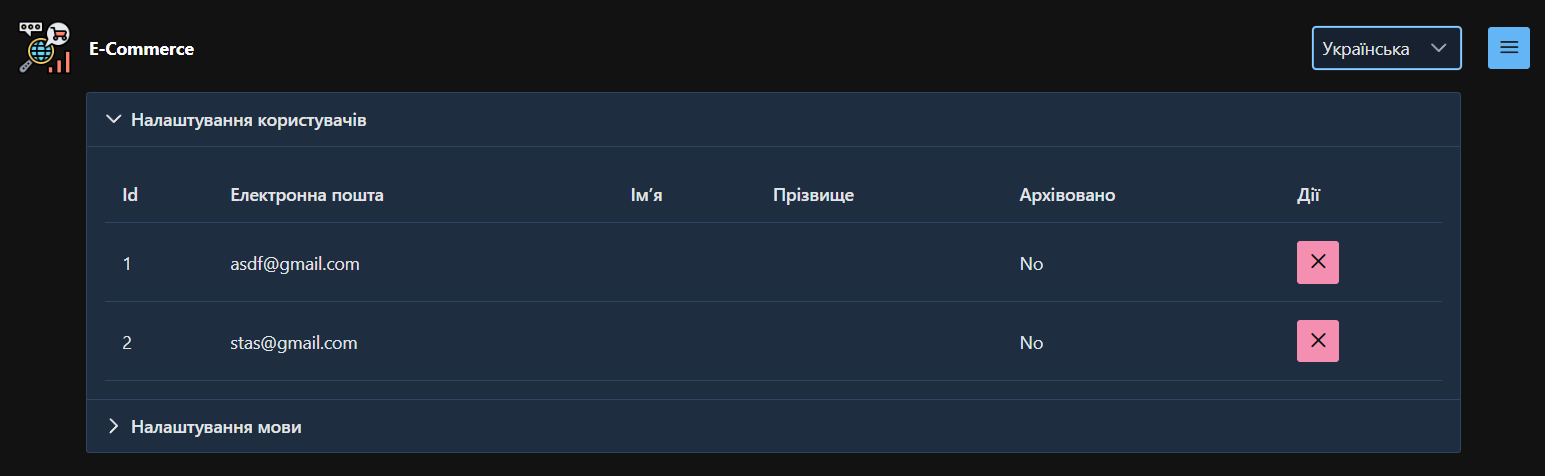


Рисунок 4.2 – Відображення інтерфейсу українською мовою  
(рисунок виконано самостійно)

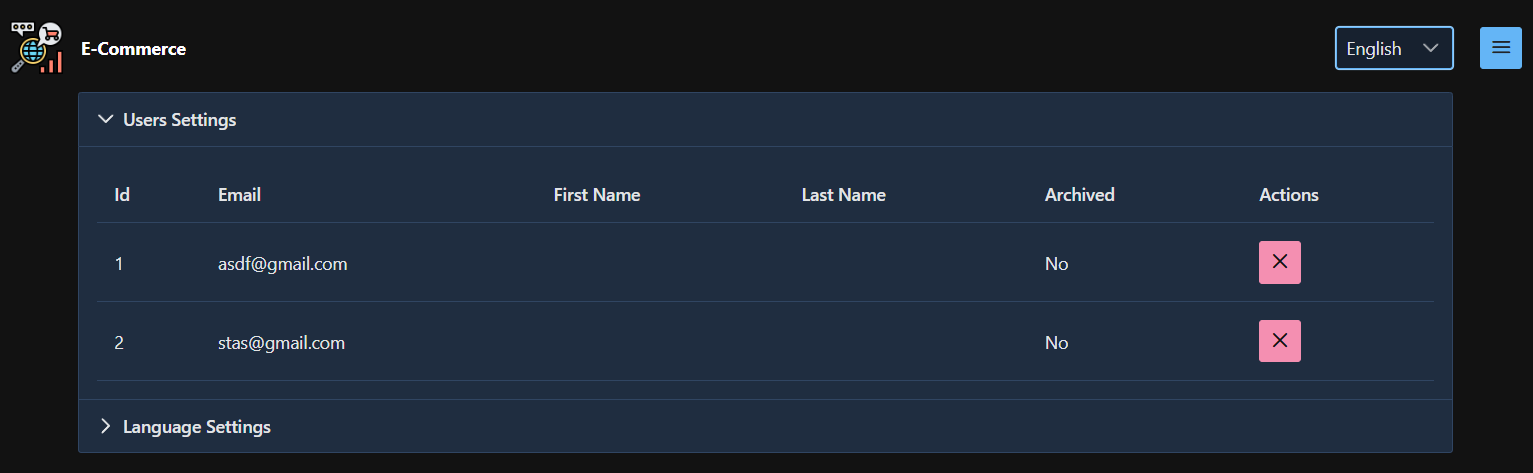


Рисунок 4.3 – Відображення інтерфейсу англійською мовою  
(рисунок виконано самостійно)

## Підтвердження входу за допомогою електронної пошти

З метою підвищення безпеки користувацького входу в системі реалізовано механізм підтвердження входу за допомогою електронної пошти. Це дозволяє уникнути класичного входу з використанням пароля та натомість вимагає введення коду підтвердження, який надсилається на пошту користувача.

На серверній частині для надсилання електронних листів використовується окремий сервіс EmailService, який працює через SMTP-протокол. Конфігураційні параметри, такі як хост, порт, логін, пароль, адреса відправника та ім’я, передаються через файл конфігурації застосунку.

Метод SendEmailAsync створює HTML-лист із зазначеним заголовком та текстом і надсилає його одержувачу:

public async Task SendEmailAsync(string to, string subject, string body)

{

try

{

using var client = new SmtpClient(\_settings.Host, \_settings.Port)

{

EnableSsl = \_settings.EnableSsl,

Credentials = new NetworkCredential(\_settings.Username, \_settings.Password)

};

var mail = new MailMessage

{

From = new MailAddress(\_settings.SenderEmail, \_settings.SenderName),

Subject = subject,

Body = body,

IsBodyHtml = true

};

mail.To.Add(to);

await client.SendMailAsync(mail);

}

catch (Exception ex)

{

throw new Exception(ex.Message);

}

}

Лист містить одноразовий код підтвердження, який користувач має ввести для завершення входу.

Після надсилання коду його копія зберігається у таблиці Logins у базі даних разом із датою створення, IP-адресою, User-Agent та іншими метаданими. Ця інформація дозволяє у майбутньому проводити аудит дій користувача.

Користувач вводить отриманий код на клієнтській частині інтерфейсу   
(рис. 4.4). Цей код передається на сервер, де відбувається перевірка:

[HttpPost(nameof(VerifyCode))]

public async Task<IActionResult> VerifyCode([FromBody] VerifyCode model)

{

var latest = await \_dbContext.Logins

.Where(x => x.Email == model.Email && !x.IsCompleted)

.OrderByDescending(x => x.Date)

.FirstOrDefaultAsync();

if (latest == null || latest.Code != model.Code || (DateTime.UtcNow - latest.Date).TotalMinutes > 10)

return BadRequest("Invalid or expired code");

latest.IsCompleted = true;

await \_dbContext.SaveChangesAsync();

var user = \_dbContext.Users.FirstOrDefault(x => x.Email == model.Email);

var token = GenerateJwtToken(user.Email, user.Role);

return Ok(new { token });

}

Якщо код вірний і не застарілий (термін дії – 10 хвилин), вхід вважається підтвердженим. Статус запису IsCompleted змінюється на true, після чого користувач отримує **JWT токен** для подальшої роботи в системі.

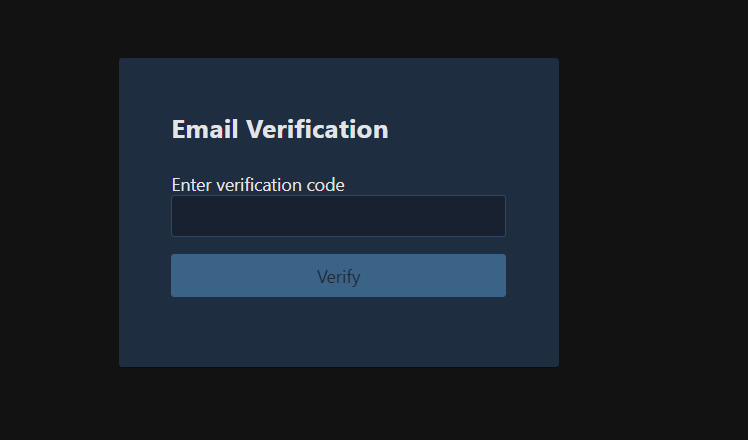


Рисунок 4.4 – Інтерфейс введення коду підтвердження  
(рисунок виконано самостійно)

## Використання Mongo DB для збереження історичних даних статистики

У системі збирання та візуалізації аналітичних даних виникає потреба у **зберіганні історичних записів** для подальшого аналізу. Для цієї мети використовується **нереляційна база даних MongoDB**, яка забезпечує зручне зберігання неструктурованих або слабо структурованих даних у форматі BSON-документів.

MongoDB обрана як сховище для агрегованих статистичних даних про товари з різних маркетплейсів та бірж, оскільки:

* не вимагає жорстко заданої схеми таблиць;
* дозволяє ефективно зберігати великі об’єми документів;
* добре підходить для сценаріїв, де дані мають змінну структуру;
* підтримує масштабування й високий рівень продуктивності для читання/запису.

У системі реалізовано окремий сервіс MongoDataService, який відповідає за взаємодію з колекцією MongoDB. Ініціалізація відбувається за допомогою параметрів підключення з конфігурації (MongoSettings), а підключення до конкретної колекції виконується наступним чином:

var client = new MongoClient(settings.Value.ConnectionString);

var database = client.GetDatabase(settings.Value.DatabaseName);

\_collection = database

.GetCollection<ProductDto>(settings.Value.CollectionName);

Збереження нових записів виконується методом SaveManyAsync, який приймає колекцію об’єктів ProductDto та додає їх до бази. Зберігаються лише непорожні масиви, що дозволяє уникати зайвих запитів:

public async Task SaveManyAsync(IEnumerable<ProductDto> data)

{

if (data.Any())

{

await \_collection.InsertManyAsync(data);

}

}

Для отримання даних з MongoDB реалізовано кілька методів:

* GetAllAsync – повертає повний список усіх записів;
* GetBySourceAsync – дозволяє отримати всі записи, що надійшли з конкретного джерела (наприклад, DummyJSON або FakeStore), фільтруючи їх за полем Source.

Клас ProductDto містить поля для збереження назви, опису, ціни, категорії, посилання на зображення, а також джерела даних.

Збирання та підготовка даних відбувається у сервісі агрегації AggregatorService, який відповідає за об’єднання даних з різних API-джерел у єдиний набір. Цей сервіс реалізує логіку паралельного отримання даних за допомогою інтерфейсу IApiSourceService, що дозволяє легко масштабувати кількість джерел без зміни загальної логіки.

Основний метод GetAggregatedDataAsync виконує запити до кожного джерела, агрегує отримані дані, зберігає їх у MongoDB та формує об’єкт відповіді типу AggregatedDataResponse:

public async Task<AggregatedDataResponse> GetAggregatedDataAsync()

{

try

{

var tasks = \_sources.Select(src => src.GetLatestDataAsync());

var results = await Task.WhenAll(tasks);

var aggregated = results.SelectMany(r => r).ToList();

await \_mongo.SaveManyAsync(aggregated);

return new AggregatedDataResponse

{

FakeStoreData = aggregated.Where(x => x.Source == DataSource.FakeStore).ToList(),

DummyJsonData = aggregated.Where(x => x.Source == DataSource.DummyJSON).ToList(),

OpenLibraryData = aggregated.Where(x => x.Source == DataSource.OpenLibrary).ToList(),

OpenFoodFactsData = aggregated.Where(x => x.Source == DataSource.OpenFoodFacts).ToList(),

CryptoData = aggregated.Where(x => x.Source == DataSource.CoinGecko).ToList()

};

}

catch (Exception ex)

{

throw new Exception(ex.Message);

}

}

Зібрані дані сортуються за типом джерела (наприклад, FakeStore, DummyJSON, OpenLibrary, OpenFoodFacts, CoinGecko) і передаються у відповідні поля результату. Це дозволяє клієнтській частині зручно обробляти й візуалізувати інформацію в узагальненому вигляді.

## Використання SignalR для оновлень аналітичних даних у реальному часі

У системі передбачено оновлення аналітичних даних без потреби перезавантаження сторінки. Для цього в системі використовується технологія SignalR, яка дозволяє організувати двосторонню комунікацію між клієнтом і сервером у реальному часі.

SignalR дозволяє автоматично визначати та використовувати оптимальний транспортний механізм (WebSockets, Server-Sent Events або Long Polling), що робить його універсальним рішенням для широкого спектру браузерів і сценаріїв використання.

На стороні сервера створено спеціальний хаб AnalyticsHub, який виступає точкою підключення клієнтів і каналом передачі повідомлень:

public class AnalyticsHub : Hub

{

public async Task SubscribeToUpdates()

{

await Clients.Caller.SendAsync("subscribed", "Successfully subscribed to updates.");

}

}

Сервіс агрегації AggregatorService після збору та збереження нових даних викликає метод оповіщення підключених клієнтів через хаб. Це дозволяє миттєво інформувати всі клієнтські інтерфейси про наявність нових статистичних даних:

public async Task SendAggregatedDataAsync()

{

await \_hubContext.Clients.All.SendAsync("dataUpdated");

}

На клієнтській стороні (в Angular-додатку) налаштовано SignalR-клієнт, який підключається до хабу, прослуховує повідомлення "dataUpdated" і виконує повторний запит до API для оновлення графіків і таблиць. Це реалізовано за допомогою @microsoft/signalr та RxJS.

this.hubConnection = new signalR.HubConnectionBuilder()

.withUrl(`${this.apiUrl}/analyticsHub`)

.build();

this.hubConnection.start().then(() => {

console.log('SignalR connected');

this.hubConnection.on('dataUpdated', () => {

this.analyticsService.reloadData();

});

});

Використання SignalR у системі дозволяє:

* значно зменшити затримку між оновленням даних на сервері та їх відображенням на клієнті;
* уникнути регулярного опитування API (polling), що зменшує навантаження на сервер;
* забезпечити кращий користувацький досвід через живу реакцію інтерфейсу на зміни;
* легко масштабувати систему з використанням Redis або Azure SignalR Service для балансування між серверами.

Після отримання даних вони візуалізуються у вигляді діаграм (рисунок 4.5).

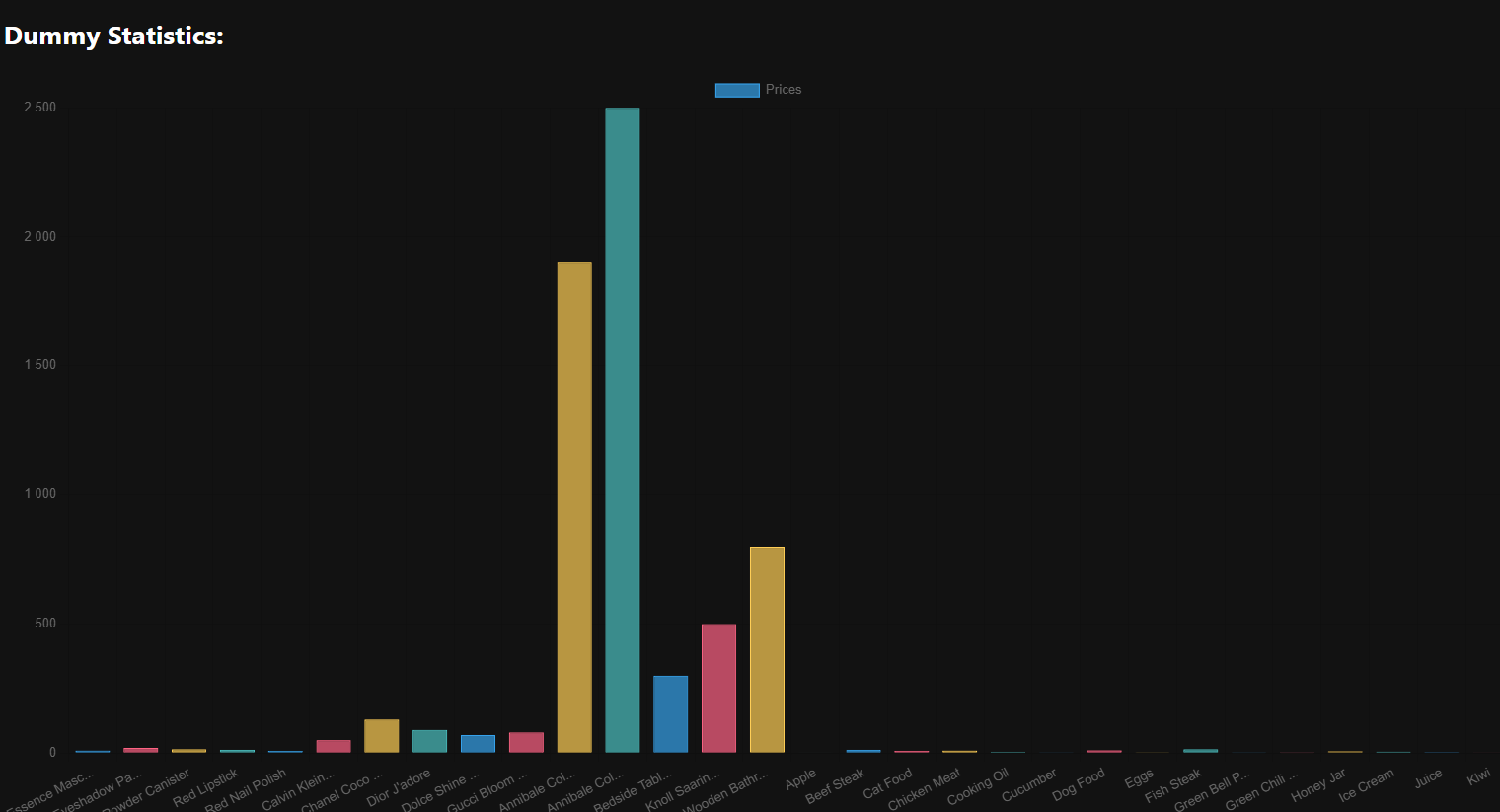


Рисунок 4.5 – Оновлення даних на клієнті в режимі реального часу  
(рисунок виконано самостійно)

## Реалізація панелі адміністратора

У системі передбачено окремий функціонал для адміністратора, який надає доступ до критично важливих керуючих операцій – управління мовними файлами та користувачами. Для забезпечення безпеки та розмежування прав доступу, всі відповідні контролери захищені атрибутом [Authorize(Roles = "Admin")], що гарантує, що доступ до цих ендпоінтів мають лише користувачі з роллю «Admin».

Однією з основних можливостей адміністративної панелі є управління мовними пакетами, які використовуються системою для реалізації багатомовного інтерфейсу. Адміністратор має змогу завантажити новий JSON-файл з перекладами, а також видалити існуючу мовну версію. Для завантаження використовується метод мовного контролеру UploadJson, який приймає base64-кодований JSON-файл, декодує його та передає до мовного сервісу:

[HttpPost]

[Authorize(Roles = "Admin")]

[Route(nameof(UploadJson))]

public async Task<IActionResult> UploadJson(string code)

{

using (var reader = new StreamReader(Request.Body))

{

var data = Convert.FromBase64String(await reader.ReadToEndAsync());

var json = Encoding.UTF8.GetString(data);

await \_languageService.UploadJson(code, json);

\_translationCacheService.ResetCache();

}

return Ok();

}

Видалення мови здійснюється через інший захищений маршрут, який також автоматично очищує кеш перекладів:

[HttpDelete]

[Authorize(Roles = "Admin")]

[Route(nameof(RemoveLanguage))]

public async Task<IActionResult> RemoveLanguage(string code)

{

await \_languageService.RemoveLanguage(code);

\_translationCacheService.ResetCache();

return Ok();

}

Вигляд інтерфейсу адмін-панелі на клієнті відображено на рисунку 4.6.

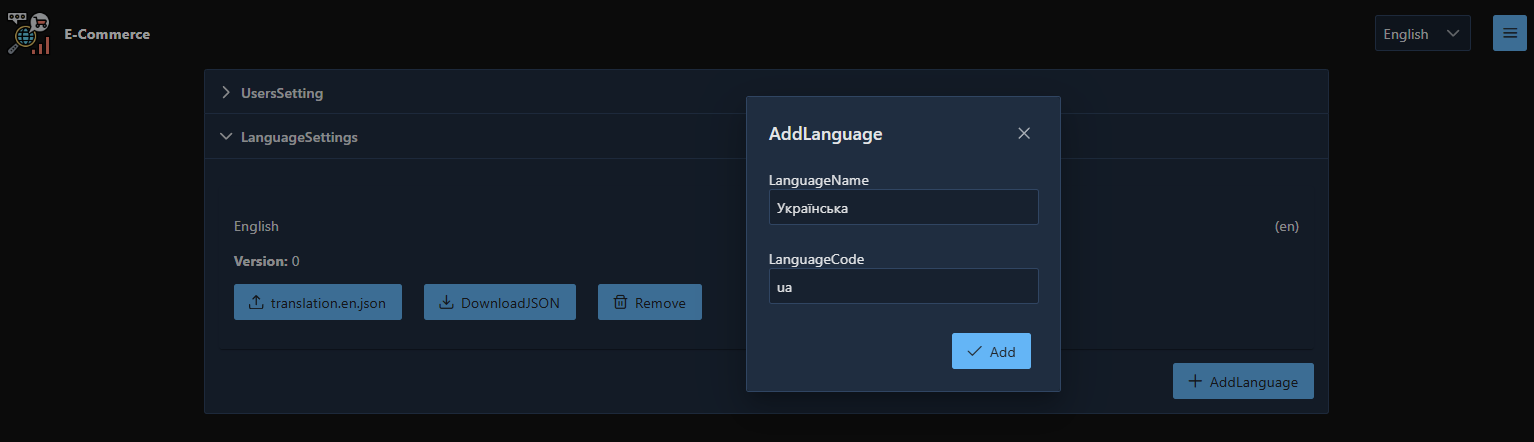


Рисунок 4.6 – Інтерфейс панелі адміністратора для керування мовами

(рисунок виконано самостійно)

Крім мовних функцій, адмін-панель дозволяє керувати користувачами. Наприклад, метод GetUsers повертає перелік усіх зареєстрованих користувачів. Також передбачено можливість архівації (деактивації) облікових записів користувачів через метод ArchiveUser, що приховує їх без фізичного видалення з бази даних. Звичайні користувачі мають змогу отримати лише власну інформацію через захищений маршрут, але без обмеження за роллю. Вигляд інтерфейсу панелі адміністратора для управління користувачами наведено на рисунку 4.7.

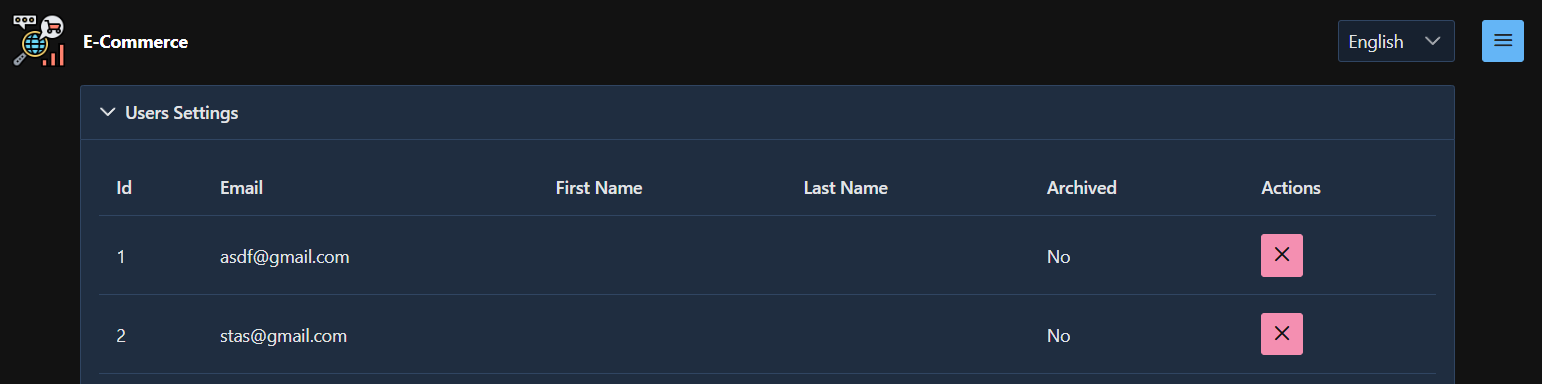


Рисунок 4.7 – Інтерфейс панелі адміністратора для керування користувачами

(рисунок виконано самостійно)

Уся логіка адміністративних функцій реалізована з урахуванням принципів безпеки, ролей користувачів та розширюваності, що дозволяє надалі додавати нові можливості адмін-панелі без порушення цілісності архітектури системи. Адмін-панель є одним із ключових інструментів підтримки та налаштування системи в процесі її експлуатації.

## REST-інтерфейс серверної частини

Архітектура взаємодії між клієнтською та серверною частинами програмної системи побудована згідно принципів REST (Representational State Transfer**)**. Це стиль архітектури програмного забезпечення, що ґрунтується на використанні стандартних HTTP-методів (GET, POST, PUT, DELETE, PATCH) та ідентифікаторів ресурсів у вигляді URI. REST дозволяє будувати зрозумілий, масштабований та легко підтримуваний інтерфейс взаємодії.

У системі всі функціональні можливості реалізовані через REST API, який поділений на кілька логічних блоків: автентифікація (Auth), обробка даних (Data), робота з мовами інтерфейсу (Language), управління користувачами (User). Кожен маршрут відповідає певній дії, а доступ до нього може бути обмежено роллю користувача.

Таблиця 4.1 – REST-специфікація маршрутів серверної частини

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | Маршрут | Роль | Дія |
| POST | /api/Auth/register | Анонім | Реєстрація нового користувача |
| POST | /api/Auth/login | Анонім | Надсилання запиту на вхід та генерація коду |
| POST | /api/Auth/verifyCode | Анонім | Підтвердження коду входу та отримання JWT-токена |
| GET | /api/Data/GetAllData | Користувач | Отримання агрегованих аналітичних даних |
| GET | /api/Language/ GetLanguagesList | Користувач | Отримання списку мов, доступних у системі |
| POST | /api/Language/UploadJson | Адміністратор | Завантаження нового JSON-файлу перекладу |
| DELETE | /api/Language/ RemoveLanguage | Адміністратор | Видалення мови з системи |
| GET | /api/Language/GetLanguage | Користувач | Отримання метаінформації про конкретну мову |
| GET | /api/Language/ GetLanguageJsonHolder | Користувач | Завантаження JSON-перекладу обраної мови |
| POST | /api/Language/AddLanguage | Адміністратор | Додавання нової мови до бази |
| GET | /api/Language/DownloadJson | Адміністратор | Завантаження перекладу у форматі JSON |

Кінець таблиці 4.1

|  |  |  |  |
| --- | --- | --- | --- |
| GET | /api/User/GetUsers | Адміністратор | Отримання списку користувачів |
| GET | /api/User/ArchiveUser | Адміністратор | Архівація (деактивація) облікового запису |
| GET | /api/User/GetUser | Користувач | Отримання особистої інформації за email |
| PATCH | /api/User/UpdateField | Користувач | Зміна окремих полів профілю (наприклад ім’я або прізвище) |
| PATCH | /api/User/ChangePassword | Користувач | Зміна пароля |

# Тестування програмного забезпечення

З метою перевірки коректності роботи програмної системи аналітики даних електронної комерції у реальному часі було обрано кілька видів тестування. Основну увагу зосереджено на **інтеграційному тестуванні серверної частини** та **функціональному тестуванні клієнтської частини**, які є найбільш доцільними для даного типу застосунку. Також розглянуто можливість проведення **навантажувального тестування** для оцінки продуктивності в умовах великого обсягу даних і паралельних з'єднань.

Інтеграційне тестування орієнтоване на перевірку взаємодії між різними компонентами серверної частини, такими як контролери, сервіси, репозиторії, база даних та зовнішні API. Це дозволяє виявити помилки не в окремих класах, а в логіці їхньої взаємодії.сервіси аутентифікації (генерація токенів, валідація коду),

* взаємодія між контролерами авторизації та сервісами генерування JWT токенів;
* робота сервісу підтвердження входу (логування коду, перевірка терміну дії, завершення авторизації);
* обробка запитів на агрегацію даних з маркетплейсів та криптобірж;
* збереження та читання даних із MongoDB;
* доступ до захищених API з урахуванням ролей користувача;
* обробка мовних файлів та коректна генерація JSON-відповідей для клієнта.

Для реалізації інтеграційних тестів використовувались тестові середовища на основі InMemoryDatabase (для SQL) та ізольовані екземпляри MongoDB, що дозволяє запускати тести без втручання у бойові дані.

Приклад інтеграційного тесту (без коду Test Fixture):

[Fact]

public async Task VerifyCode\_Should\_ReturnJwtToken\_WhenCodeIsValid()

{

// Arrange

var model = new

{

Email = "test@example.com",

Code = "123456"

};

var content = new StringContent(JsonConvert.SerializeObject(model), Encoding.UTF8, "application/json");

// Act

var response = await \_client.PostAsync("/api/Auth/VerifyCode", content);

// Assert

response.EnsureSuccessStatusCode();

var result = await response.Content.ReadAsStringAsync();

dynamic parsed = JsonConvert.DeserializeObject(result);

Assert.False(string.IsNullOrEmpty((string)parsed.token));

}

Функціональне тестування зосереджене на перевірці поведінки системи відповідно до вимог користувача. Основними перевіреними сценаріями стали:

* реєстрація користувача та вхід з підтвердженням через email;
* отримання даних через REST API;
* вибір мови інтерфейсу та підвантаження мовного пакету;
* оновлення статистики в реальному часі через SignalR;
* доступ до адміністративних функцій (тільки для користувачів з відповідною роллю);

Під час тестування перевірялася також відповідність HTTP-відповідей (коди 200, 401, 403, 404), реакція інтерфейсу на помилки, поведінка при відсутності інтернет-з’єднання або при недоступності серверної частини.

Одним з прикладів тестування є перевірка перегляду аналітичної інформації зареєстрованим користувачем. Дані тест-кейсу наведені у таблиці 5.1.

Таблиця 5.1 – Тест-кейс перевірки перегляду аналітичної інформації

|  |  |
| --- | --- |
| Інформація про тест-кейс | |
| Ідентифікатор тесту: | Тест-кейс №1 |
| Опис функції: | Перегляд дашборду з аналітичними даними після входу |
| Власник тесту: | Ткаченко Станіслав Максимович |
| Дата створення: | 17.06.2025 |
| Мета тесту: | Перевірити правильність відображення дашборду та взаємодії з графіками після авторизації |

Кінець таблиці 5.1

|  |  |  |  |
| --- | --- | --- | --- |
| Передумова | | | |
| № | Опис випадку | Очікуваний результат | Висновок |
| 1 | Користувач зареєстрований у системі та виконав успішний вхід | Користувач перенаправлений на головну сторінку /dashboard | Пройдено |
| Скасування події | | | |
| № | Опис випадку | Очікуваний результат | Висновок |
| 1 | Користувач заходить на /dashboard після входу | На екрані відображається набір графіків та заголовок «Аналітика в реальному часі» | Пройдено |
| 2 | Один із графіків не має даних (відсутні API-дані) | Відображається повідомлення «Дані тимчасово недоступні» | Пройдено |
| 3 | Користувач натискає на кнопку оновлення даних | Дані оновлюються через SignalR без перезавантаження сторінки | Пройдено |
| 4 | Користувач змінює мову інтерфейсу | Усі підписи графіків та заголовки перекладаються відповідно до обраної мови | Пройдено |
| 5 | Користувач натискає на категорію товарів на графіку | Відкривається модальне вікно з детальною інформацією | Пройдено |
| Результати тестування | | | | |
| Дата прогону тесту:  26.05.2025 | | Результат тесту (P/F/B):  ПРОЙДЕНО (P) | | |

Також було проведено навантажувальне тестування. Воно дозволяє оцінити продуктивність системи за умови великої кількості одночасних користувачів або частих оновлень даних. Було змодельовано навантаження з одночасним підключенням до 150 клієнтів через SignalR та частою агрегацією даних кожні 10 секунд. Система стабільно працювала, час доставки оновлень становив менше 200 мс, а час відповіді API – до 150 мс. Помилок і збоїв не зафіксовано. Для глибшого аналізу продуктивності планується використання спеціалізованих інструментів, таких як **Apache JMeter, Postman Runner** або **k6.**

# Висновки

У кваліфікаційній роботі бакалавра було розроблено та детально описано систему аналітики електронної комерції в реальному часі, яка дозволяє ефективно агрегувати, аналізувати та візуалізувати дані з різних маркетплейсів. Проведений аналіз предметної галузі засвідчив, що системи аналітики в реальному часі сьогодні є критично важливим інструментом для бізнесу, оскільки дозволяють миттєво реагувати на зміни ринку, аналізувати поведінку користувачів та оптимізувати продажі без затримок. Реалізована система враховує всі ці аспекти, надаючи користувачам інтерактивні дашборди, динамічні графіки та можливість отримувати сповіщення про важливі події.

Під час формування вимог до системи акцент було зроблено на швидкість оновлення даних, масштабованість архітектури та інтуїтивність інтерфейсу. Система інтегрує дані з різних джерел, включаючи API маркетплейсів та платіжні системи, забезпечуючи єдиний інтерфейс для аналізу всіх аспектів електронної комерції. Також реалізовано механізми двосторонньої комунікації через SignalR, що дозволяє миттєво доставляти оновлення до клієнтського інтерфейсу без необхідності оновлення сторінки.

Архітектурне проектування програмного забезпечення базується на принципах чистої архітектури та масштабованості, що дозволяє чітко розділити відповідальність між компонентами системи. Серверна частина реалізована на ASP.NET Core 7.0 з використанням WebAPI для RESTful ендпоінтів та SignalR для комунікації в реальному часі. Клієнтська частина розроблена на Angular 16 з використанням RxJS для реактивної обробки потоків даних та NgRx для централізованого управління станом застосунку. Особлива увага приділялася розробці інтерфейсу користувача: всі дашборди мають адаптивний дизайн, інтуїтивно зрозумілі елементи управління та можливість персоналізації відображення даних.

Прийняті програмні рішення спрямовані на створення високопродуктивної та надійної системи. Для цього використано асинхронну обробку даних через фонові сервіси, кешування часто запитуваної інформації, оптимізацію запитів до бази даних та ефективну передачу даних через WebSocket протокол з автоматичним fallback на Long Polling при нестабільному з'єднанні. Всі ці заходи дозволяють забезпечити обробку до 1000 подій на секунду з мінімальною затримкою.

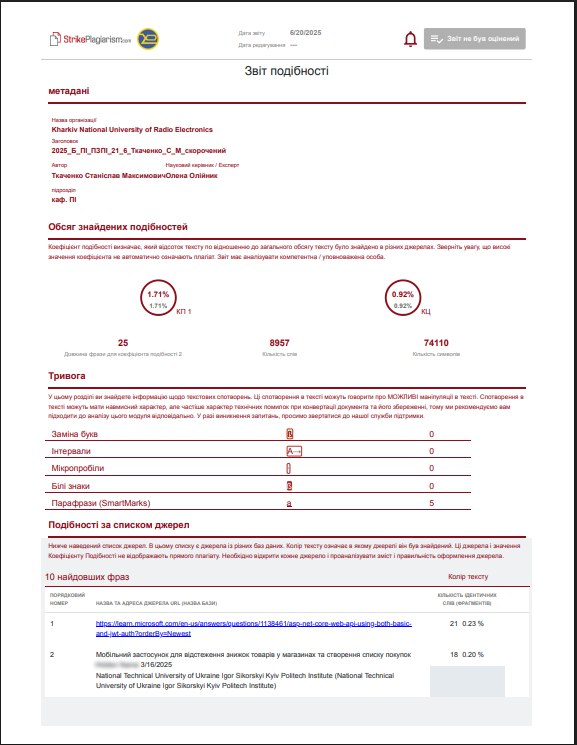
У результаті проведеного тестування розробленої систему. Розроблена система відповідає сучасним вимогам як у плані функціональності та продуктивності, так і щодо масштабованості та надійності. Майбутні напрямки розвитку проєкту включають інтеграцію алгоритмів машинного навчання для прогнозування трендів продажів, розширення аналітичних можливостей системи та впровадження додаткових джерел даних. Отже, дана робота створює надійну основу для подальших досліджень та практичного застосування технологій реального часу у сфері аналітики електронної комерції.

# Перелік джерел посилання

1. Promodo. Research of the Ukrainian eCommerce Market, 2024. URL: https://www.promodo.com/blog/research-of-the-ukrainian-ecommerce-market (дата звернення: 17.06.2025)
2. Statista. eCommerce share of retail sales worldwide, 2023. URL: https://www.statista.com/statistics/534123/e-commerce-share-of-retail-sales-worldwide (дата звернення: 17.06.2025)
3. Visa Ukraine. Digital Payments in Ukrainian eCommerce, 2024. URL: https://www.visa.com.ua/en\_UA/about-visa/newsroom/press-releases/prl-14052025.html (дата звернення: 17.06.2025)
4. UniformMarket. Omnichannel Consumer Behavior, 2023. URL: https://www.uniformmarket.com/blog/omnichannel-consumer-behavior (дата звернення: 17.06.2025)
5. PwC. Consumer Data Privacy Survey, 2023. URL: https://www.pwc.com/us/en/industries/consumer-markets/library/consumer-intelligence-series/pwc-consumer-privacy.html (дата звернення: 17.06.2025)
6. Payments Journal. Fraud Challenges in eCommerce, 2024. URL: https://www.paymentsjournal.com/fraud-challenges-in-ecommerce/ (дата звернення: 17.06.2025)
7. McKinsey & Company. Talent Shortage in Digital Retail, 2023. URL: https://www.mckinsey.com/industries/retail/our-insights/tackling-the-retail-talent-crunch (дата звернення: 17.06.2025)
8. IDC. Global Streaming & Edge Spending Forecast, 2024. URL: https://www.idc.com/getdoc.jsp?containerId=US50120723 (дата звернення: 17.06.2025)
9. Digital Commerce 360. US eCommerce Sales Stats, 2024. URL: https://www.digitalcommerce360.com/article/us-ecommerce-sales/ (дата звернення: 17.06.2025)
10. Reuters. Alibaba & JD.com Growth Amid Slowdown, 2024. URL: https://www.reuters.com/technology/alibaba-jdcom-report-q1-earnings-2024-05-15 (дата звернення: 17.06.2025)
11. Mercado Libre Investor Relations. GMV Growth Report, 2024. URL: https://investor.mercadolibre.com/events/event-details/q1-2024-mercado-libre-inc-earnings-conference-call (дата звернення: 17.06.2025)
12. Cross-Origin Resource Sharings (CORS). Docs. URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Guides/CORS (дата звернення: 17.06.2025).
13. ASP.NET Core documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0 (дата звернення: 17.06.2025).
14. Angular documentation. URL: https://angular.dev/overview (дата звернення: 17.06.2025).
15. What is three-tier architecture. URL: <https://www.ibm.com/think/topics/three-tier-architecture> (дата звернення: 21.06.2025).
16. Мартін Р. Чистий код: створення і рефакторинг за допомогою   
    AGILE. Харків: Фабула, 2019. 416 с.
17. K. Chodorow. MongoDB: The Definitive Guide. Beijing: O’Reilly,  
    2013. 432 p.
18. Програмний код кваліфікаційної роботи на GitHub. URL: <https://github.com/StanislavTkachenko0/2025_B_PI_PZPI-21-6_Tkachenko_S_M> (дата звернення: 20.06.2025).

# додаток А

Результат перевірки на унікальність тексту у базі ХНУРЕ



# додаток Б

Фрагменти програмного коду

Б.1 Сервіс агрегації даних

public class AggregatorService : IAggregatorService

{

private readonly IEnumerable<IApiSourceService> \_sources;

private readonly MongoDataService \_mongo;

private readonly IHubContext<AnalyticsHub> \_hubContext;

public AggregatorService(

IEnumerable<IApiSourceService> sources,

MongoDataService mongo,

IHubContext<AnalyticsHub> hubContext

)

{

\_sources = sources;

\_mongo = mongo;

\_hubContext = hubContext;

}

public async Task<AggregatedDataResponse> GetAggregatedDataAsync()

{

try

{

var tasks = \_sources.Select(src => src.GetLatestDataAsync());

var results = await Task.WhenAll(tasks);

var aggregated = results.SelectMany(r => r).ToList();

await \_mongo.SaveManyAsync(aggregated);

return new AggregatedDataResponse

{

FakeStoreData = aggregated.Where(x => x.Source == DataSource.FakeStore).ToList(),

DummyJsonData = aggregated.Where(x => x.Source == DataSource.DummyJSON).ToList(),

OpenLibraryData = aggregated.Where(x => x.Source == DataSource.OpenLibrary).ToList(),

OpenFoodFactsData = aggregated.Where(x => x.Source == DataSource.OpenFoodFacts).ToList(),

CryptoData = aggregated.Where(x => x.Source == DataSource.CoinGecko).ToList()

};

}

catch (Exception ex)

{

throw new Exception(ex.Message);

}

}

public async Task SendAggregatedDataAsync()

{

await \_hubContext.Clients.All.SendAsync("dataUpdated");

}

}

Б.2 Сервіс для отримання аналітичних даних з зовнішнього API

public class OpenFoodFactsApiService : IApiSourceService

{

private readonly HttpClient \_httpClient;

public string SourceName => "OpenFoodFacts";

public OpenFoodFactsApiService(HttpClient httpClient)

{

\_httpClient = httpClient;

}

public async Task<IEnumerable<ProductDto>> GetLatestDataAsync()

{

try

{

var url = "https://world.openfoodfacts.org/api/v2/search?fields=product\_name,categories\_tags,nutriments&sort\_by=unique\_scans\_n&page\_size=20";

var response = await \_httpClient.GetAsync(url);

if (!response.IsSuccessStatusCode)

{

var msg = await response.Content.ReadAsStringAsync();

throw new Exception($"Failed to fetch food data: {response.StatusCode}. {msg}");

}

var content = await response.Content.ReadAsStringAsync();

using var doc = JsonDocument.Parse(content);

var products = doc.RootElement.GetProperty("products");

var result = new List<ProductDto>();

foreach (var p in products.EnumerateArray())

{

double pseudoPrice = 0;

if (p.TryGetProperty("nutriments", out var nutriments) &&

nutriments.TryGetProperty("energy-kcal\_100g", out var kcalElement) &&

kcalElement.TryGetDouble(out var kcal))

{

pseudoPrice = Math.Round(kcal / 10, 2);

}

result.Add(new ProductDto

{

Title = p.TryGetProperty("product\_name", out var name) ? name.GetString() : "Unknown",

Category = p.TryGetProperty("categories\_tags", out var tags) && tags.GetArrayLength() > 0

? tags[0].GetString()?.Replace("en:", "") : "food",

Price = pseudoPrice,

Source = DataSource.OpenFoodFacts,

});

}

return result.Where(r => r.Price.HasValue).ToList();

}

catch (Exception ex)

{

throw new Exception($"OpenFoodFacts error: {ex.Message}", ex);

}

}

}

Б.3 Налаштування контексту бази даних

public class ECommerceDbContext: DbContext

{

public ECommerceDbContext(DbContextOptions<ECommerceDbContext> options) : base(options) { }

public DbSet<UserModel> Users { get; set; }

public DbSet<LanguageModel> Languages { get; set; }

public DbSet<LoginInfo> Logins { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

var stringListComparer = new ValueComparer<IList<string>>(

(c1, c2) => c1.SequenceEqual(c2),

c => c.Aggregate(0, (a, v) => HashCode.Combine(a, v.GetHashCode())),

c => c.ToList());

modelBuilder.Entity<LanguageModel>(e =>

{

e.Property(x => x.Version).IsConcurrencyToken();

});

}

}

Б.4 Сервіс для підтримки багатомовності

public class LanguageService : ILanguageService

{

private readonly ECommerceDbContext \_context;

private readonly ITranslationCacheService \_translationCacheService;

public string DefaultLanguage => "en";

public LanguageService(

ECommerceDbContext context,

ITranslationCacheService translationCacheService

)

{

\_context = context;

\_translationCacheService = translationCacheService;

}

public async Task<List<LanguageModel>> GetLanguagesInfo()

{

var tempLangs = new List<LanguageModel>();

var langs = \_context.Languages.ToList();

if (langs.All(lang => lang.LanguageCode != this.DefaultLanguage))

{

await AddFirstLanguageAsync();

langs = \_context.Languages.ToList();

}

for (var langIndex = 0; langIndex < langs.Count; langIndex++)

{

tempLangs.Add(new LanguageModel()

{

LanguageCode = langs[langIndex].LanguageCode,

LanguageName = langs[langIndex].LanguageName,

Version = langs[langIndex].Version

});

}

return tempLangs;

}

public async Task AddLanguage(AddLanguageModel model)

{

if (\_context.Languages.Any(lng => lng.LanguageCode == model.LanguageCode.ToLower()))

{

throw new Exception("Language with \"" + model.LanguageCode.ToLower() + "\" already exsit");

}

var jsonPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Template", "default\_language.json");

var json = File.ReadAllText(jsonPath);

var enLang = \_context.Languages.SingleOrDefault(lng => lng.LanguageCode == this.DefaultLanguage);

\_context.Languages.Add(new LanguageModel()

{

LanguageName = model.LanguageName,

LanguageCode = model.LanguageCode.ToLower(),

TranslationJson = json,

});

await \_context.SaveChangesAsync();

}

public async Task RemoveLanguage(string code)

{

var lang = \_context.Languages.SingleOrDefault(lng => lng.LanguageCode == code.ToLower());

\_context.Remove(lang);

await \_context.SaveChangesAsync();

}

public LanguageModel GetLanguage(string code)

{

var key = \_translationCacheService.GetKey(TranslationType.LanguageModel, code);

lock (\_translationCacheService.GetSyncObject(key))

{

if (\_translationCacheService.Get(key) is LanguageModel languageModel && this.GetVersion(languageModel.LanguageCode).Equals(languageModel.Version))

{

return languageModel;

}

if (!this.HasLanguage(code))

{

code = this.DefaultLanguage;

}

var storedLang = \_context.Languages.Single(lng => EF.Functions.Like(lng.LanguageCode, $"%{code}%"));

var language = new LanguageModel

{

LanguageName = storedLang.LanguageName,

LanguageCode = storedLang.LanguageCode,

Version = storedLang.Version,

TranslationJson = storedLang.TranslationJson,

};

\_translationCacheService.Set(key, language);

return language;

}

}

public async Task<string> GetJson(string code)

{

if (!\_context.Languages.Any(lng => lng.LanguageCode == this.DefaultLanguage))

{

await AddFirstLanguageAsync();

}

var enLang = await \_context.Languages.FirstAsync(lng => lng.LanguageCode == this.DefaultLanguage);

var lang = await \_context.Languages.SingleOrDefaultAsync(lng => lng.LanguageCode == code.ToLower());

var translations = JsonConvert.DeserializeObject<Dictionary<string, string>>(lang.TranslationJson);

return JsonConvert.SerializeObject(translations);

}

public int GetVersion(string code)

{

if (!this.HasLanguage(code))

{

code = this.DefaultLanguage;

}

var lang = \_context.Languages.SingleOrDefault(lng => lng.LanguageCode == code.ToLower());

return lang.Version;

}

public bool HasLanguage(string code)

{

return !string.IsNullOrWhiteSpace(code) && \_context.Languages.Any(lng => lng.LanguageCode == code.ToLower());

}

public async Task UploadJson(string code, string json)

{

var lang = await \_context.Languages.SingleOrDefaultAsync(lng => lng.LanguageCode == code.ToLower());

lang.TranslationJson = json;

lang.Version++;

await \_context.SaveChangesAsync();

}

public LanguageJsonHolder GetLanguageJsonHolder(string code)

{

string countryCode = code.ToLower();

string key = \_translationCacheService.GetKey(TranslationType.Language, countryCode);

lock (\_translationCacheService.GetSyncObject(key))

{

var holder = TryGetValidCachedHolder(countryCode);

if (holder != null)

return holder;

if (HasLanguage(countryCode))

{

return SetLanguageInCache(countryCode);

}

var fallbackHolder = TryGetValidCachedHolder("en");

return fallbackHolder ?? SetLanguageInCache("en");

}

}

private LanguageJsonHolder? TryGetValidCachedHolder(string code)

{

var key = \_translationCacheService.GetKey(TranslationType.Language, code);

var cached = \_translationCacheService.Get<LanguageJsonHolder>(key);

return cached != null && GetVersion(cached.languageCode) == cached.Version

? cached

: null;

}

private LanguageJsonHolder SetLanguageInCache(string code)

{

var key = \_translationCacheService.GetKey(TranslationType.Language, code);

var holder = new LanguageJsonHolder

{

json = GetJson(code).Result,

languageCode = code,

Version = GetVersion(code)

};

return \_translationCacheService.Set(key, holder);

}

private async Task AddFirstLanguageAsync()

{

var jsonPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Template", "default\_language.json");

var json = File.ReadAllText(jsonPath);

\_context.Languages.Add(new LanguageModel()

{

LanguageName = "English",

LanguageCode = this.DefaultLanguage,

TranslationJson = json,

});

await \_context.SaveChangesAsync();

}

}

Б.5 Налаштування маршрутизації клієнтської частини

const routes: Routes = [

  {

    path: '',

    redirectTo: 'client',

    pathMatch: 'full',

  },

  {

    path: 'client',

    loadChildren: () => import('./modules/ECommerceClient/ecommerce-app.module').then(module => module.EcommerceAppModule)

  },

  {

    path: 'admin',

    loadChildren: () => import('./modules/ECommerceAdmin/ecommerce-admin.module').then(module => module.EcommerceAdminModule)

  }

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Б.6 Налаштування безпеки доступу до маршрутів

@Injectable({

  providedIn: 'root'

})

export class AuthAdminGuard implements CanActivate {

  constructor(

    private authService: AuthService,

    private router: Router,

    private messageService: MessageService,

  ) {}

  canActivate(): boolean {

    console.log(this.authService.getUserRole())

    if (this.authService.hasToken() && this.authService.isTokenExpired()) {

      this.authService.logout();

      this.router.navigate(['/sign-in']);

      this.messageService.add({ severity: 'error', summary: 'Error', detail: 'Token Expired.' });

      return false;

    }

    if(this.authService.isAuthenticated() && this.authService.getUserRole() !== Role.Admin) {

      this.router.navigate(['/client']);

      return false;

    }

    if (this.authService.isAuthenticated() && this.authService.getUserRole() === Role.Admin) {

      return true;

    }

    this.router.navigate(['/sign-in']);

    return false;

  }

}

Б.7 Розмітка компоненту профілю

<div class="profile mx-auto">

  <form (ngSubmit)="onSave()" [formGroup]="profileForm" class="space-y-4">

    <div class="mt">

      <label>{{ 'Email' | translate }}</label>

      <input pInputText

             formControlName="email"

             type="email"

             class="w-100" />

    </div>

    <div class="mt">

      <label>{{ 'FirstName' | translate }}</label>

      <input pInputText

             formControlName="firstName"

             class="w-100" />

    </div>

    <div class="mt">

      <label>{{ 'LastName' | translate }}</label>

      <input pInputText

             formControlName="lastName"

             class="w-100" />

    </div>

    <div class="pt-4 border-t">

      <h3 class="text-lg font-semibold mb-2">{{ 'ChangePassword' | translate }}</h3>

      <div>

        <label>{{ 'NewPassword' | translate }}</label>

        <input pInputText

               formControlName="password"

               type="password"

               class="w-100" />

      </div>

      <div class="mt">

        <label>{{ 'ConfirmPassword' | translate }}</label>

        <input pInputText

               formControlName="confirmPassword"

               type="password"

               class="w-100" />

        <div style="height: 10px">

          <ng-container \*ngIf="(profileForm.get('confirmPassword')?.touched || profileForm.get('confirmPassword')?.dirty)

                            && profileForm.get('confirmPassword')?.invalid">

            <small class="p-error"

                   \*ngIf="profileForm.get('confirmPassword')!.hasError('passwordMismatch') && !profileForm.get('password')?.invalid">

              {{ 'PasswordNotMatch' | translate }}

            </small>

          </ng-container>

        </div>

      </div>

    </div>

    <button pButton

            type="submit"

            [label]="'ChangePassword' | translate" class="mt"></button>

  </form>

</div>

# додаток В

Слайди презентації

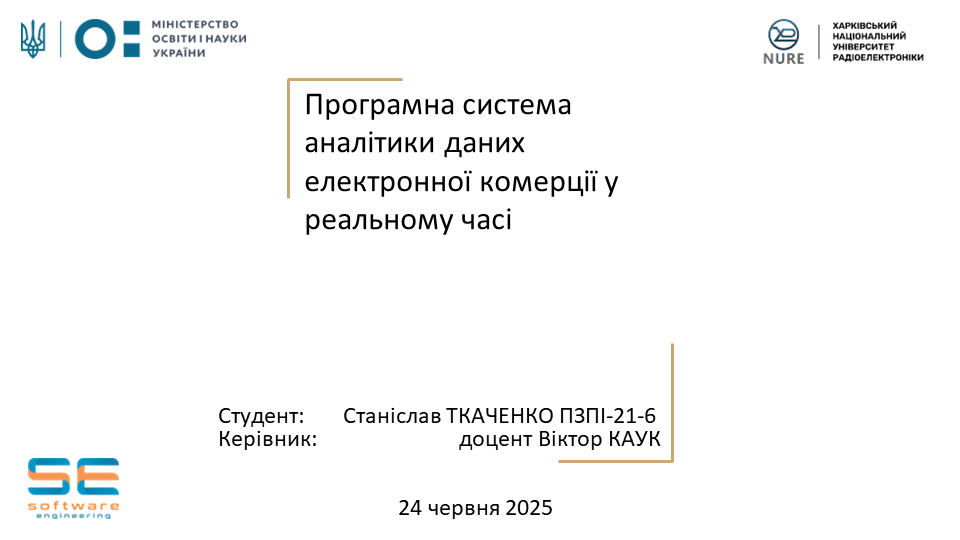


Рисунок В.1 – Слайд №1

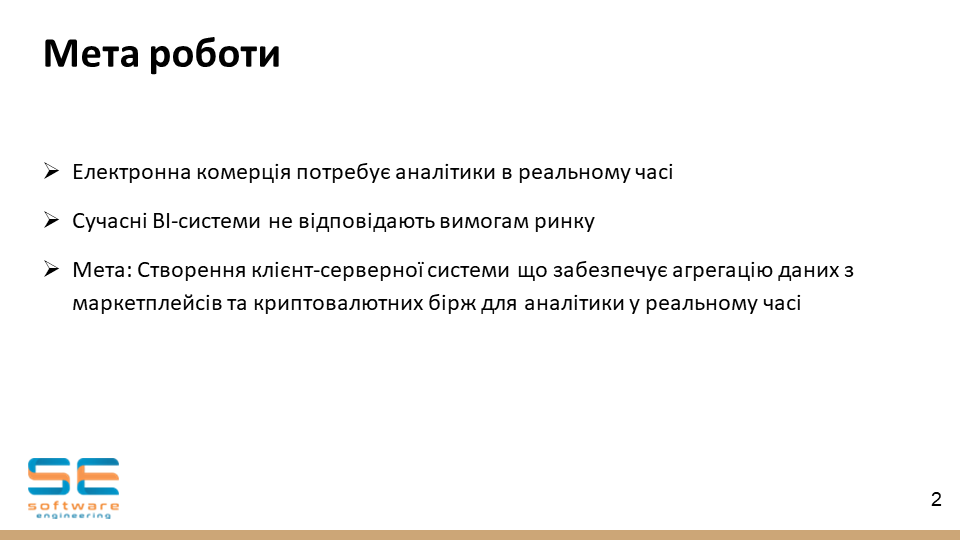


Рисунок В.2 – Слайд №2



Рисунок В.3 – Слайд №3



Рисунок В.4 – Слайд №4



Рисунок В.5 – Слайд №5



Рисунок В.6 – Слайд №6



Рисунок В.7 – Слайд №7

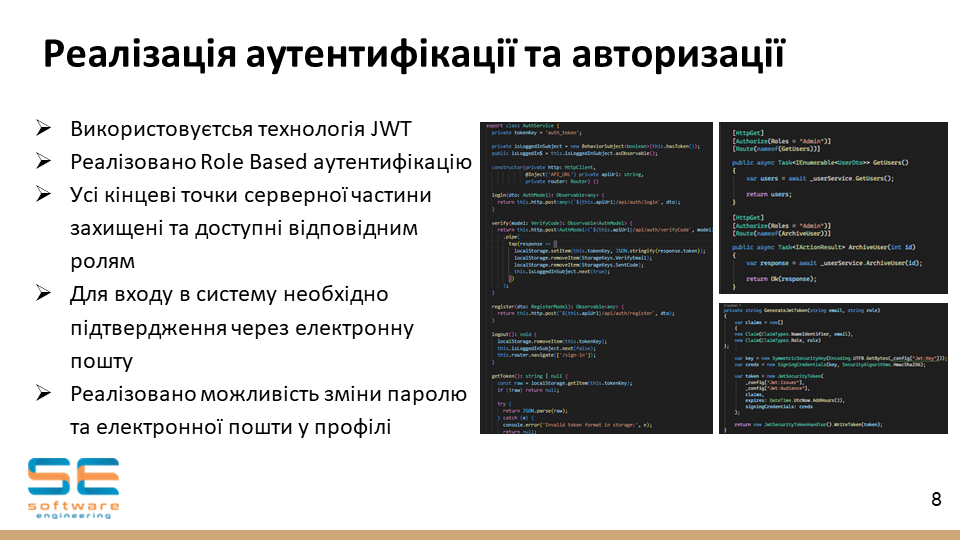


Рисунок В.8 – Слайд №8

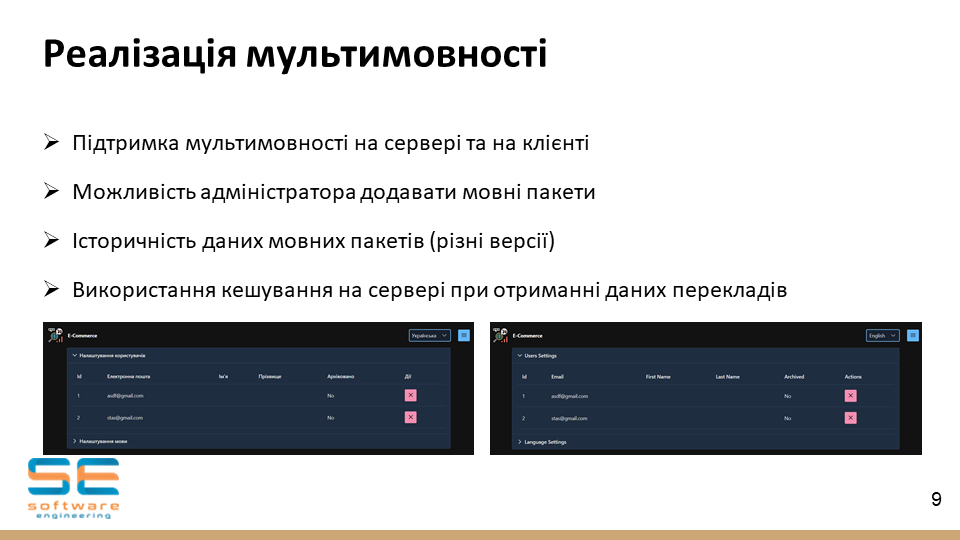


Рисунок В.9 – Слайд №9

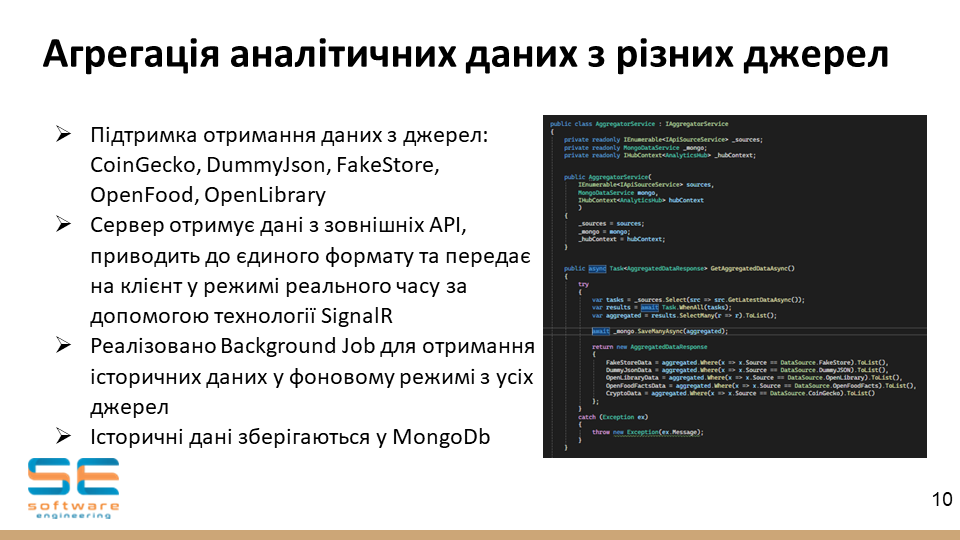


Рисунок В.10 – Слайд №10

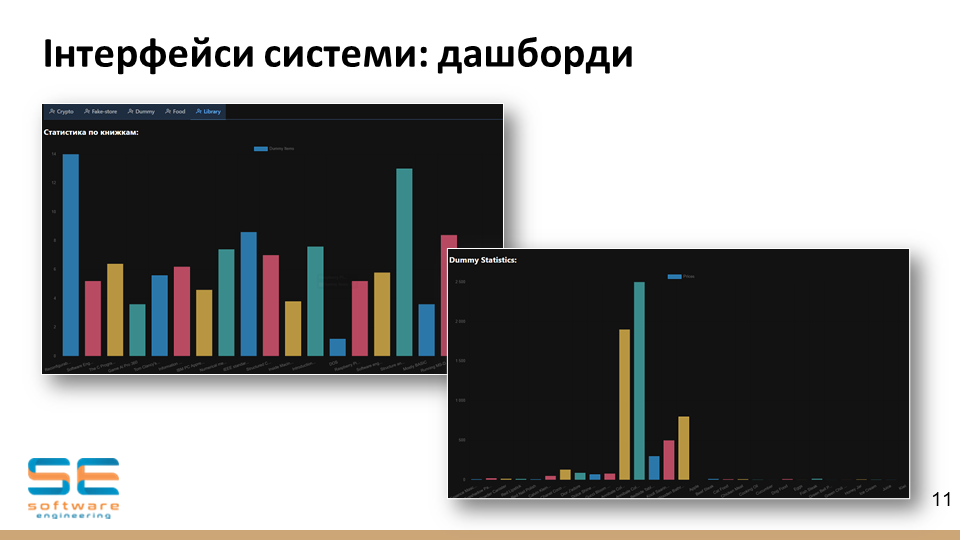


Рисунок В.11 – Слайд №11

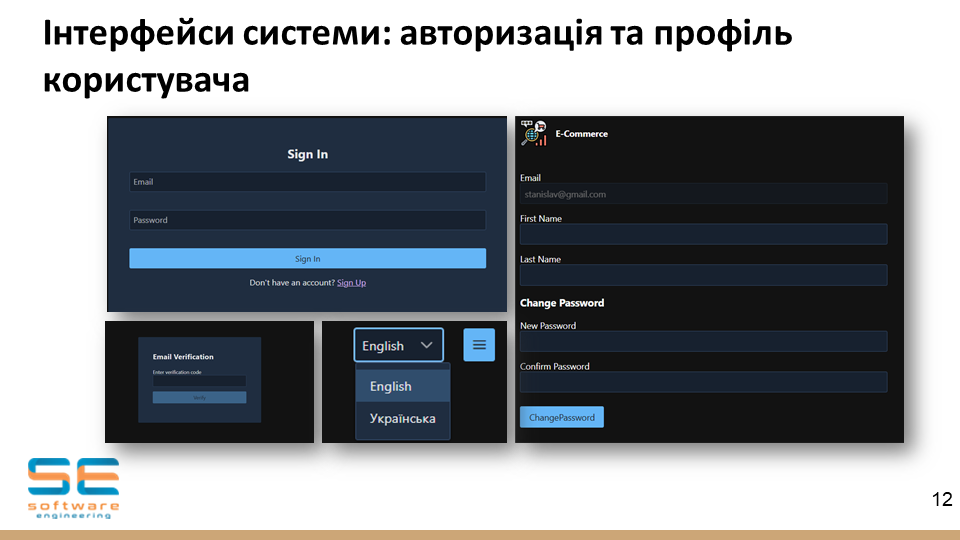


Рисунок В.12 – Слайд №12

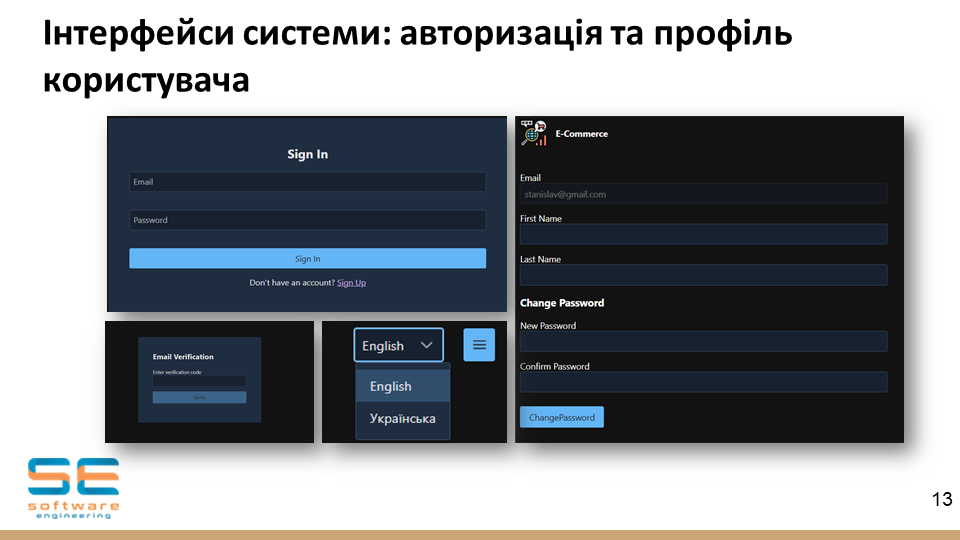


Рисунок В.13 – Слайд №13

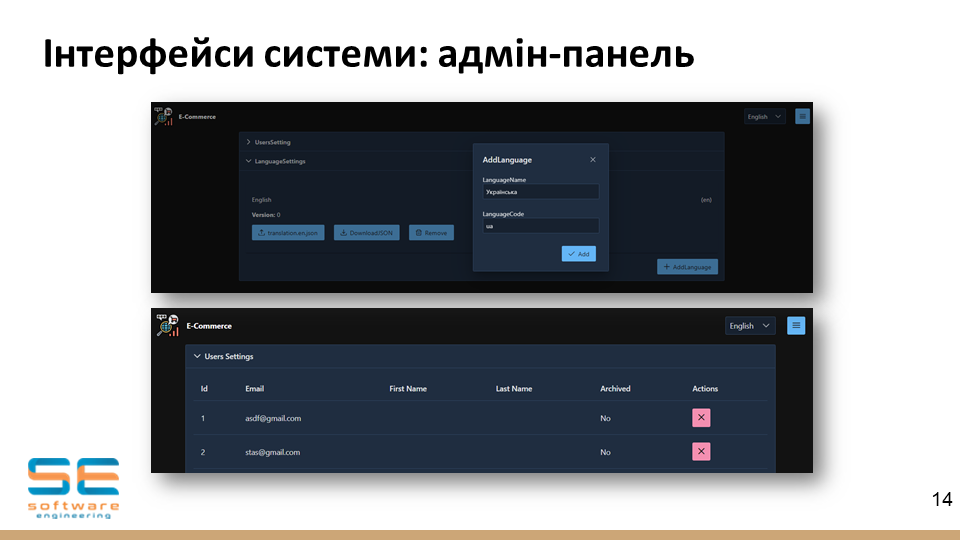


Рисунок В.14 – Слайд №14

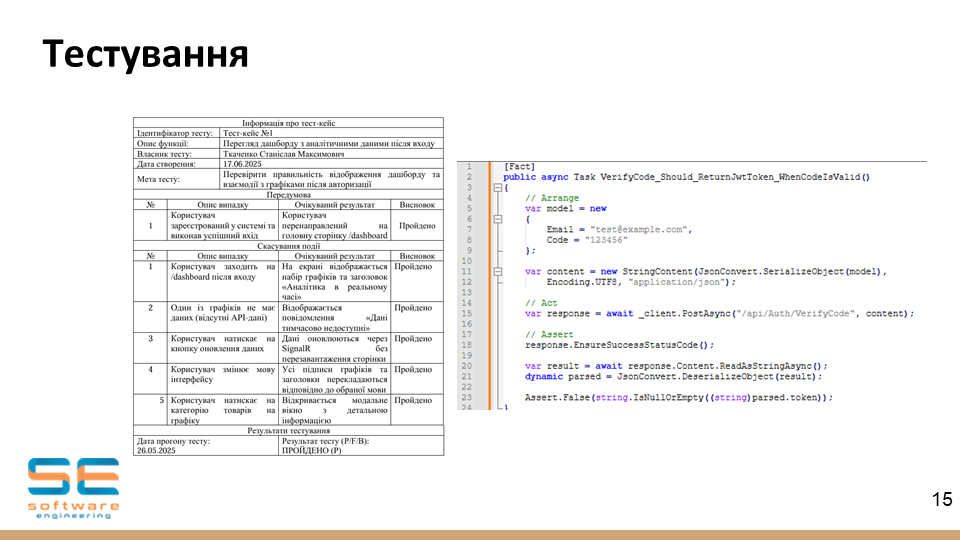


Рисунок В.15 – Слайд №15

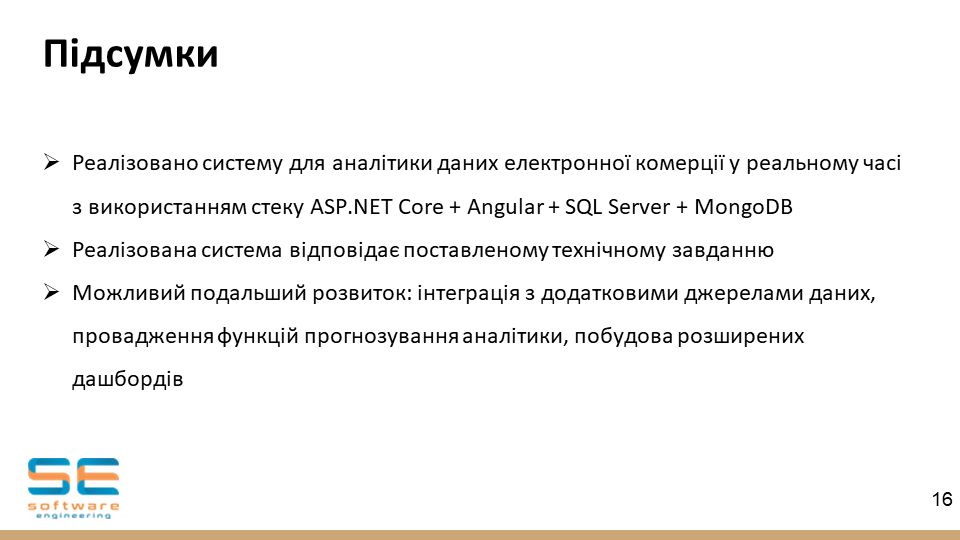


Рисунок В.16 – Слайд №16