МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

АЛЕКСЮК ВОЛОДИМИР ВІКТОРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2025 р.

ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА МЕТОДИ ОБРОБКИ ВЕЛИКИХ ДАНИХ У ВЕБЗАСТОСУНКАХ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (магістерська) робота

Науковий керівник:
Надія ПОТАПОВА,
доцент кафедри інформаційних технологій,
к. е. н., доцент

_____
(підпис)

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

АНОТАЦІЯ

Алексюк В.В. Інформаційні технології та методи обробки великих даних у вебзастосунках. Спеціальність 122 «Комп'ютерні науки». Освітня програма Комп'ютерна обробка даних (Data Science). Донецький національний університет імені Василя Стуса, Вінниця, 2025.

Магістерська робота присвячена дослідженню інформаційних технологій та методів обробки великих даних у вебзастосунку для отримання аналітики продажів у сфері електронної комерції. Використано сучасні підходи до проектування вебзастосунків, методи об'єктно-орієнтованого програмування та технології роботи з NoSQL-базами даних. Основними інструментами розробки стали Node.js, Express.js, MongoDB, Vue.js та Visual Studio Code.

Магістерська робота складається зі вступу, трьох розділів, висновків і додатків. У вступі обґрунтовується актуальність теми, визначається мета та завдання дослідження. У першому розділі досліджено теоретичні основи та проведено аналіз підходів до оброки великих даних. У другому розділі представлено архітектурні рішення в проектуванні та методи розробки вебзастосунку. Обґрунтовано вибір архітектурного підходу та інструментів реалізації, описано модель обробки та візуалізації даних, спроектовано структуру бази даних та визначено модель взаємодії компонентів системи. У третьому розділі викладено результати практичної реалізації вебзастосунку та експериментального дослідження його роботи з великими даними.

Ключові слова: інформаційні технології, великі дані, вебзастосунок, об'єктно-орієнтоване програмування, архітектура вебзастосунку, база даних.

100 с., 49 рис., 4 дод., 50 джерел.

ABSTRACT

Alexyuk V.V. Information technologies and methods of big data processing in web applications. Specialization 122 “Computer Science.” Educational program Computer Data Processing (Data Science). Vasyl’ Stus Donetsk National University, Vinnytsia, 2025.

The master's thesis is devoted to the study of information technologies and methods of processing big data in a web application for obtaining sales analytics in the field of e-commerce. It uses modern approaches to web application design, object-oriented programming methods, and technologies for working with NoSQL databases. The main development tools were Node.js, Express.js, MongoDB, Vue.js, and Visual Studio Code.

The master's thesis consists of an introduction, three chapters, conclusions, and appendices. The introduction justifies the relevance of the topic and defines the purpose and objectives of the study. The first chapter explores the theoretical foundations and analyzes approaches to big data processing. The second chapter presents architectural solutions in design and methods of web application development. The choice of architectural approach and implementation tools is justified, the data processing and visualization model is described, the database structure is designed, and the model of interaction between system components is defined. The third chapter presents the results of the practical implementation of the web application and experimental research of its work with big data.

Keywords: information technology, big data, web application, object-oriented programming, web application architecture, database.

100 p., 49 fig., bibliography: 50 items.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

APP – додаток (Application)

API – інтерфейс прикладного програмування (Application Programming Interface)

SQL – стандартна мова програмування, призначена для керування реляційними базами даних та взаємодії з ними (Structured Query Language)

NoSQL – широкий клас систем керування базами даних, які відрізняються від традиційних реляційних баз даних (Not Only SQL)

REST – архітектурний стиль для створення веб-сервісів (Representational State Transfer)

CSV – текстовий формат для збереження табличних даних (Comma-Separated Values)

SPA – веб-застосунок, який працює на одній HTML сторінці (Single Page Application)

URL – унікальна адреса ресурсу в інтернеті (Uniform Resource Locator)

JSON – текстовий формат обміну структурованими даними (JavaScript Object Notation)

BSON – розширений двійковий формат для зберігання та обміну даними (Binary JSON)

AWS - Amazon Web Services

GCP - Google Cloud Platform

ВСТУП

Актуальність теми дослідження. У сучасних умовах глобальної цифровізації та розвитку електронної комерції обсяги даних, які щодня генеруються підприємствами, значно зростають. Продажі товарів і послуг у мережі супроводжуються формуванням великих обсягів інформації про клієнтів, транзакції, маркетингові кампанії та поведінку користувачів. Аналіз цих даних дозволяє отримати цінні знання для підвищення ефективності бізнесу, прогнозування попиту та формування конкурентних стратегій.

Водночас обробка великих обсягів даних у вебзастосунках залишається складним завданням через необхідність забезпечення швидкодії, масштабованості та зручності представлення результатів кінцевому користувачу. Традиційні системи бізнес-аналітики часто є надто складними для впровадження в малих і середніх компаніях або не відповідають вимогам інтерактивності. Тому актуальною є розробка вебзастосунків, що інтегрують методи роботи з великими даними, використовуючи сучасні технології візуалізації та забезпечуючи зрозумілу та доступну форму подання результатів аналізу. Таким чином, дослідження інформаційних технологій та методів обробки великих даних у вебзастосунках є важливим як з теоретичної, так і з практичної точки зору. Це дозволяє продемонструвати можливості сучасних вебтехнологій на прикладі системи аналітики продажів у сфері електронної комерції.

Мета магістерської роботи полягає у дослідженні інформаційних технологій та методів обробки великих даних у вебзастосунках, а також особливостей їх використання на основі розробки прототипу інформаційної системи для аналітики продажів у сфері електронної комерції.

Для досягнення даної мети були поставлені **наступні завдання:**

1. Проаналізувати сучасні підходи та інструменти для обробки великих даних у вебзастосунках.

2. Дослідити методи збору, зберігання та представлення даних у

системах електронної комерції.

3. Обґрунтувати вибір архітектури та технологій для реалізації вебзастосунку.

4. Розробити вебзастосунок для аналітики продажів, що включає інтерактивну візуалізацію даних.

5. Провести тестування створеної системи та оцінити її потенціал у вирішенні поставлених завдань.

Об'єкт дослідження – інформаційні технології та процеси збору, аналізу та обробки великих даних у вебзастосунках.

Предмет дослідження – методи, моделі та архітектурні рішення, що використовуються для створення вебзастосунків з обробки великих даних, зокрема у сфері електронної комерції.

У процесі дослідження використовувалися такі **методи**:

- Аналіз літературних джерел та існуючих рішень у сфері обробки великих даних, вебаналітики та електронної комерції – для формування теоретичної бази та обґрунтування актуальності теми.
- Методи об'єктно-орієнтованого проектування та програмування – для розробки архітектури вебзастосунку, моделювання структури даних та реалізації логіки взаємодії між клієнтською і серверною частинами.
- Методи агрегації та трансформації даних на основі MongoDB Aggregation Pipeline – для побудови запитів, що забезпечують узагальнення, фільтрацію, сортування та форматування транзакційної інформації.
- Методи візуалізації даних – для створення інтерактивного інтерфейсу, що дозволяє користувачам аналізувати результати у вигляді графіків, діаграм та таблиць.
- Тестування – для перевірки працездатності системи, оцінки її ефективності, стабільності та відповідності функціональним вимогам.

Наукова новизна дослідження полягає в розробці прототипу вебзастосунку для аналітики онлайн-продажів з реалізованою серверною логікою, яка забезпечує агрегацію, фільтрацію, сортування та трансформацію

транзакційної інформації за допомогою MongoDB Aggregation Pipeline, що дозволяє формувати узагальнені аналітичні показники без додаткової обробки на клієнтській стороні.

Структуру роботи складають: вступ, три розділи, висновки, список використаних джерел та додатки. У першому розділі досліджено теоретичні основи та проведено аналіз підходів до оброки великих даних. Проведено огляд і порівняння існуючих програмних рішень, а також сформульовано вимоги до створюваного вебзастосунку. У другому розділі представлено проектування та методи розробки вебзастосунку. Обґрунтовано вибір архітектурного підходу та інструментів реалізації, описано модель обробки та візуалізації даних, спроектовано структуру бази даних та визначено модель взаємодії компонентів системи. У третьому розділі викладено результати практичної реалізації вебзастосунку та експериментального дослідження його роботи. Описано структуру та компоненти системи, реалізацію програмного продукту, проведено тестування та аналіз результатів, а також визначено практичне значення та перспективи використання розробленого рішення.

Практичне значення роботи полягає у використанні прототипу вебзастосунку для аналітики продажів у сфері електронної комерції при обробці великих даних з отриманням результатів інтерактивного аналізу транзакцій, формуванням узагальнених показників та їх візуалізацією. Прототип може бути використаний як практичний інструмент для компаній, а також як навчальний приклад чи основа для подальшого розширення функціоналу.

Апробація результатів дослідження. Результати даного дослідження було апробовано в доповіді «Інформаційні технології для підтримки прийняття рішень у сфері електронної комерції» на IV Міжнародній науково-практичній конференції «Прикладні аспекти сучасних міждисциплінарних досліджень» (м. Вінниця, 5 листопада 2025 року) та VI Всеукраїнській науково-практичній конференції «Комп’ютерні технології обробки даних» (м. Вінниця, 5 грудня 2025 року).

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ПІДХОДІВ ДО ОБРОБКИ ВЕЛИКИХ ДАНИХ У ВЕБЗАСТОСУНКАХ

1.1 Поняття та сутність вебзастосунків для обробки великих даних

Сучасний етап розвитку інформаційних технологій характеризується стрімким і безпрецедентним зростанням обсягів цифрової інформації, яка щоденно генерується у величезних масштабах різними суб'єктами – від приватних користувачів до великих корпорацій, державних установ, наукових центрів та соціальних платформ. Такий масив інформації, що має великий обсяг, високу швидкість надходження та різноманіття форматів, прийнято називати великими даними (Big Data) [1]. Вони включають текстові документи, мультимедійні файли, транзакційні записи, сенсорні потоки, лог-файли, геолокаційні дані тощо.

За даними дослідження міжнародної аналітичної компанії IDC, загальний обсяг даних, що створюються та копіюються у світі, зростає експоненційно, і, згідно з прогнозами, до кінця 2025 року він перевищить 175 зеттабайт [2]. Такий масштаб інформаційного потоку створює нові виклики для бізнесу, науки та технологій, зокрема – потребу в ефективних інструментах для збору, обробки, аналізу, зберігання та візуалізації даних у зручному та доступному форматі.

Основні характеристики великих даних традиційно описують через концепцію «трьох V»: Volume (обсяг), Velocity (швидкість), Variety (різноманітність). У сучасних дослідженнях додають також Veracity (достовірність) і Value (цінність). До складу великих даних входять не лише текстові записи чи транзакції, але й мультимедійні файли, сенсорні потоки, геолокаційна інформація, системні журнали та результати взаємодії користувачів у соціальних мережах. Їх масштаб і складність вимагають застосування спеціальних методів збору, обробки та аналізу, оскільки традиційні інструменти часто виявляються недостатніми.

Робота з великими даними є послідовним процесом, що включає кілька взаємопов’язаних стадій. Кожна з них відіграє важливу роль у формуванні кінцевого аналітичного результату:

- Аналіз джерела даних. На цьому етапі визначається, які веб-ресурси, бази даних, API або сенсорні пристрої будуть використовуватися. Важливо оцінити доступність, надійність і релевантність джерела, щоб уникнути надлишкового або неякісного матеріалу.
- Пошук і вилучення інформації. Дані вивантажуються в «сирому» вигляді – це можуть бути транзакційні записи, текстові документи, зображення або потоки подій. На даному етапі ключовим є коректне підключення до джерела і отримання максимально повного масиву інформації.
- Фільтрація та очищення. Сирі дані часто містять помилки, дублікати, пропуски або нерелевантні елементи. Очищення передбачає видалення зайвого, нормалізацію форматів (наприклад, дат і валют), а також приведення інформації до єдиного стандарту.
- Формування структурованого файлу. Після очищення дані перетворюються в зручні структури – таблиці, колекції або бази даних. Це забезпечує можливість подальшого аналізу і спрощує зберігання.
- Аналіз за заданими критеріями. На заключному етапі застосовуються методи статистики, агрегації та візуалізації. Дані групуються, сортуються, зіставляються за ключовими показниками, що дозволяє виявити закономірності, побудувати звіти і підтримати прийняття управлінських рішень.

Таким чином, обробка великих даних – це не одинична операція, а комплексний процес, який перетворює хаотичні масиви інформації в структуровані знання. Саме послідовне проходження всіх етапів забезпечує достовірність і цінність отриманих результатів.

Одним із найбільш перспективних і широко застосовуваних інструментів у цьому контексті є вебзастосунки, які поєднують у собі такі важливі характеристики, як доступність, гнучкість, масштабованість, а також

можливість інтеграції з різноманітними джерелами даних. Вебзастосунок – це програмне рішення, що функціонує на стороні сервера і забезпечує взаємодію з кінцевим користувачем через інтерфейс веб-браузера [3]. Такий підхід дозволяє уникнути необхідності встановлення програмного забезпечення на локальні пристрої, що, своєю чергою, спрощує доступ до функціоналу з будь-якої точки світу, де є підключення до мережі Інтернет. Це особливо актуально в умовах глобалізації, мобільності та віддаленої роботи, коли користувачі потребують швидкого і стабільного доступу до аналітичних інструментів незалежно від географічного розташування.

У сфері аналітики та обробки великих обсягів даних вебзастосунки виконують низку критично важливих функцій, які забезпечують повний цикл роботи з даними:

- Збір та агрегація інформації з різноманітних джерел, включаючи API, реляційні та нереляційні бази даних, хмарні сховища, локальні файлові системи, сенсорні пристрої тощо.
- Попередня обробка даних, яка включає етапи очищення, нормалізації, фільтрації, трансформації та структурування інформації для подальшого аналізу.
- Зберігання даних у відповідних форматах – як у традиційних реляційних базах (SQL), так і в гнучких нереляційних сховищах (NoSQL), що дозволяє адаптувати систему до різних типів навантаження та структури даних.
- Аналітика та візуалізація результатів, що реалізується через побудову інтерактивних таблиць, графіків, діаграм, карт та дашбордів, які дозволяють швидко оцінити стан процесів, виявити закономірності та приймати обґрунтовані рішення.
- Інтерактивна взаємодія з користувачем, яка передбачає можливість здійснення пошуку, сортування, фільтрації, порівняння та глибокого аналізу даних у реальному часі, що значно підвищує ефективність роботи з інформацією.

Однією з ключових переваг вебзастосунків у контексті Big Data є їхня здатність до масштабування. Це означає, що серверна інфраструктура може бути динамічно адаптована до зростаючих обсягів даних, збільшення кількості користувачів, розширення функціоналу та інтеграції з новими джерелами. Завдяки використанню хмарних технологій, контейнеризації, мікросервісної архітектури та розподілених обчислень, такі застосунки здатні забезпечити високу продуктивність, надійність, гнучкість та безперервний доступ до даних навіть у складних умовах навантаження.

Таким чином, вебзастосунки в умовах сучасної цифрової економіки відіграють роль не лише інструментів для представлення інформації, але й потужних платформ для інтеграції, обробки та аналітики великих обсягів даних. Вони дозволяють організаціям різного масштабу – від стартапів до транснаціональних корпорацій – приймати стратегічно важливі управлінські рішення, базуючись на глибокому аналізі різнорідних, багатовимірних та динамічних наборів даних.

1.2 Аналіз сучасних підходів та інструментів для аналітики даних у вебсередовищі

Обробка великих обсягів даних у вебсередовищі потребує використання надійних архітектурних рішень, здатних забезпечити масштабованість, відмовостійкість та високу продуктивність. Серед найбільш поширених підходів варто виділити монолітну, мікросервісну та серверлес-архітектуру, які застосовуються залежно від специфіки завдань та вимог до системи [4]. Особливу увагу в останні роки приділяють хмарним архітектурам, що дозволяють гнучко керувати обчислювальними ресурсами, швидко масштабувати інфраструктуру та забезпечувати доступ до даних з будь-якої точки світу [5].

У контексті обробки Big Data широко використовуються розподілені обчислювальні моделі, такі як MapReduce, Apache Spark та Apache Flink, які ефективно справляються з паралельною обробкою великих масивів інформації

[6]. Хмарні платформи – Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure – надають готові рішення для зберігання, обробки та візуалізації даних, включаючи компоненти Data Lakes, ETL-сервіси, серверлес-функції, контейнеризацію (Docker, Kubernetes) та інструменти потокової обробки [7]. Такі архітектурні підходи дозволяють створювати гнучкі та надійні аналітичні вебзастосунки, адаптовані до вимог сучасного бізнесу.

Обробка великих даних у сучасних вебзастосунках є складним багаторівневим процесом, що передбачає використання комплексних технологічних стеків, які забезпечують повний цикл роботи з інформацією – від її збору до візуального представлення результатів аналізу. У сучасних вебсистемах аналітика даних реалізується через три ключові компоненти, кожен з яких виконує важливу роль у загальній архітектурі аналітичного рішення.

1. Збір та інтеграція даних. На першому етапі здійснюється отримання даних з різноманітних джерел, що можуть включати внутрішні корпоративні системи, відкриті API, вебсайти, сенсорні пристрої, соціальні мережі, а також зовнішні бази даних. Для цього широко використовуються REST- та GraphQL API, які забезпечують стандартизований доступ до інформації, а також методи вебскрейпінгу, що дозволяють автоматично витягувати дані з вебсторінок. У випадках, коли необхідна обробка потокових даних у реальному часі, застосовуються спеціалізовані сервіси, такі як Apache Kafka та RabbitMQ [8], які дозволяють об’єднувати інформацію з різних джерел у єдину логічну структуру, забезпечуючи при цьому високу швидкість передачі та обробки даних [9].

2. Зберігання та обробка даних. Залежно від типу, обсягу та структури даних, а також вимог до продуктивності та доступності, використовуються різні системи управління базами даних. Для роботи зі строго структурованими наборами інформації застосовуються реляційні бази даних, такі як PostgreSQL та MySQL, які забезпечують високий рівень надійності та підтримку складних

SQL-запитів. У випадках, коли дані мають гнучку або неструктуровану природу, використовуються нереляційні (NoSQL) рішення – MongoDB, Cassandra, Redis – які дозволяють ефективно працювати з великими обсягами інформації, що швидко змінюється [10]. Окрему категорію становлять хмарні сховища, зокрема Amazon S3 та Google Cloud Storage, які забезпечують масштабованість, географічну доступність, резервне копіювання та високу надійність зберігання. У процесі обробки даних важливу роль відіграють механізми оптимізації запитів, індексації, кешування та паралельної обробки, що дозволяє досягти високої продуктивності системи та забезпечити швидкий доступ до необхідної інформації.

3. Візуалізація та аналіз даних. На завершальному етапі здійснюється представлення результатів аналітики у формі, зручній для сприйняття та інтерпретації. Для цього використовуються потужні інструменти бізнес-аналітики (BI), такі як Microsoft Power BI, Tableau, Google Data Studio [11]. Вони дозволяють створювати інтерактивні дашборди, графіки, діаграми, таблиці та інші візуальні компоненти, які допомагають користувачам швидко оцінити ситуацію, виявити тренди, закономірності та аномалії, а також приймати обґрунтовані управлінські рішення. У вебзастосунках візуалізація часто реалізується за допомогою JavaScript-бібліотек – D3.js, Chart.js, ECharts – які забезпечують гнучкість у відображенні різних типів даних, можливість інтерактивної взаємодії з графічними елементами, а також адаптацію до потреб конкретного інтерфейсу користувача [12].

Сучасні підходи до аналітики даних у вебсередовищі також включають застосування методів статистичного аналізу, математичного моделювання, агрегації показників, прогнозування, класифікації та кластеризації даних. Це дозволяє не лише описувати поточний стан системи, але й будувати прогностичні моделі, формувати рекомендації, виявляти приховані залежності та оптимізувати процеси в бізнесі, науці, медицині, освіті та інших сферах.

Таким чином, сучасні вебзастосунки для аналітики даних є багатокомпонентними системами, що поєднують різноманітні технології

збору, обробки, зберігання та візуалізації інформації. Їх ефективність залежить від правильного вибору інструментів, здатності до масштабування, рівня інтеграції між компонентами, а також від якості реалізації алгоритмів аналітики. У результаті такі системи забезпечують підтримку процесів прийняття рішень, оперативний доступ до аналітичних результатів та підвищення загальної ефективності управління даними в організаціях різного типу.

1.3 Огляд та порівняння існуючих програмних рішень

У сучасному цифровому середовищі вебзастосунки для аналітики даних відіграють ключову роль у забезпеченні ефективного управління інформацією. Вони представлені як комерційними продуктами, що пропонують розширений функціонал за ліцензійною моделлю, так і безкоштовними сервісами з відкритим доступом, які дозволяють інтегрувати дані, здійснювати їх візуалізацію та проводити аналітичні розрахунки у режимі реального часу. Такий широкий спектр рішень дає змогу організаціям різного масштабу обирати інструменти відповідно до своїх потреб, бюджету та технічних вимог.

До найпопулярніших програмних рішень у сфері веб-аналітики належать:

- Google Data Studio. Це безкоштовний хмарний сервіс, призначений для створення інтерактивних звітів і дашбордів, який активно використовується як у бізнесі, так і в освітніх та дослідницьких проєктах. Інструмент підтримує підключення до широкого спектра джерел даних, зокрема Google Analytics, BigQuery, Google Sheets, файлів CSV, а також сторонніх API. Його головними перевагами є простота інтеграції, інтуїтивно зрозумілий інтерфейс, можливість спільного доступу до звітів та оновлення даних у реальному часі. Водночас, обмеженням є недостатня гнучкість у реалізації складних аналітичних сценаріїв, обмежена підтримка великих обсягів даних та обмежені можливості кастомізації візуалізацій [13,14].

- Tableau. Один із найвідоміших комерційних BI-інструментів, який широко використовується у корпоративному середовищі для побудови глибокої аналітики та візуалізації даних. Tableau підтримує підключення до реляційних (SQL) та нереляційних (NoSQL) баз даних, а також до хмарних платформ, таких як Amazon Web Services, Google Cloud та Microsoft Azure. Його сильними сторонами є висока гнучкість у створенні візуалізацій, можливість роботи з великими наборами даних, інтерактивність та масштабованість. Проте, недоліком є висока вартість ліцензії, що може бути критичним фактором для малих підприємств або освітніх установ [15].

- Microsoft Power BI. Цей інструмент для бізнес-аналітики тісно інтегрований з екосистемою Microsoft, зокрема з Office 365, Azure та Teams. Power BI дозволяє створювати інтерактивні дашборди, автоматизувати звіти, здійснювати глибоку аналітику та обробку великих обсягів даних. Його перевагами є зручність для користувачів Microsoft-продуктів, підтримка хмарних сервісів, можливість розгортання у локальному середовищі та доступність мобільних додатків. Водночас, обмеженням є залежність від Microsoft-інфраструктури та менша гнучкість у налаштуванні візуалізацій порівняно з Tableau [16].

- Metabase. Це open-source рішення для аналітики даних, яке дозволяє швидко створювати звіти та дашборди без необхідності програмування. Metabase підтримує підключення до різних баз даних (PostgreSQL, MySQL, MongoDB тощо) та має простий інтерфейс для формування запитів. Його перевагами є безкоштовна ліцензія, можливість локального або хмарного розгортання, а також доступність для невеликих команд або стартапів. Недоліком є обмежена функціональність для складної обробки даних, обмежена підтримка масштабування та менш розвинуті можливості візуалізації порівняно з комерційними продуктами [17].

- JavaScript-бібліотеки для візуалізації. Для інтеграції аналітичних компонентів безпосередньо у вебінтерфейс застосовуються спеціалізовані JavaScript-бібліотеки, серед яких найбільш популярними є D3.js, Chart.js та

ECharts. Вони дозволяють створювати висококастомізовані графіки, діаграми, гістограми, теплові карти та інші типи візуалізацій, що адаптуються до потреб конкретного вебзастосунку. Перевагою є повна гнучкість у дизайні, можливість інтеграції у SPA (Single Page Application), а також підтримка інтерактивності. Недоліком є необхідність знання програмування, складність реалізації складних аналітичних функцій та обмежена підтримка роботи з великими обсягами даних без додаткових серверних рішень [18].

Проаналізувавши сучасні інструменти для аналітики даних у вебсередовищі, можна дійти висновку, що кожне з рішень має свої сильні та слабкі сторони, які слід враховувати при виборі платформи для конкретного проєкту. Комерційні продукти пропонують розширений функціонал, але потребують фінансових витрат, тоді як open-source та JavaScript-бібліотеки забезпечують гнучкість і доступність, але вимагають технічної експертизи. У багатьох випадках готові рішення не дозволяють повністю контролювати робочі процеси, інтегрувати специфічні алгоритми обробки даних або адаптувати інтерфейс до унікальних потреб користувача.

Саме тому розробка власного вебзастосунку для демонстрації аналітики даних є доцільним підходом, що дозволяє реалізувати індивідуальні сценарії роботи з інформацією, забезпечити гнучку інтеграцію необхідних функцій, оптимізувати продуктивність та створити унікальний інтерфейс, адаптований до конкретних завдань і вимог користувачів.

1.4 Формулювання завдань та вимог до вебзастосунку

На основі проведеного аналізу предметної області, а також вивчення сучасних підходів до аналітики даних у вебсередовищі, сформульовано ключові завдання та вимоги до створюваного вебзастосунку, який має забезпечити ефективну взаємодію користувачів з аналітичною інформацією у сфері електронної комерції.

Головна мета розробки вебзастосунку полягає у створенні зручного, функціонального та інтуїтивно зрозумілого інструменту, що дозволить

користувачам оперативно отримувати доступ до аналітичних даних, здійснювати їх перегляд, порівняння та аналіз. Це, у свою чергу, сприятиме прийняттю обґрунтованих управлінських рішень, підвищенню ефективності бізнес-процесів, оптимізації маркетингових стратегій та покращенню взаємодії з клієнтами.

Аналітика даних у сфері електронної комерції охоплює широкий спектр показників – від динаміки продажів і популярності окремих товарів до поведінкових характеристик клієнтів та ефективності рекламних кампаній. Основні завдання, які необхідно реалізувати в межах створюваного вебзастосунку:

- Розробити вебінтерфейс, що дозволяє відображати та аналізувати дані, які зберігаються у базі даних, з урахуванням їх структури та типу. Інтерфейс має бути простим, зрозумілим і зручним для користувачів.
- Реалізувати базові механізми інтерактивної візуалізації, які дадуть змогу переглядати ключові показники продажів у вигляді графіків, діаграм та таблиць.
- Надати можливість перегляду даних за різними категоріями, такими як товари, клієнти або типи замовлень, без складної фільтрації чи сегментації.
- Забезпечити функцію пошуку та сортування, щоб користувачі могли швидко знаходити потрібну інформацію за ключовими параметрами.

Створюваний вебзастосунок повинен задовольняти потреби кінцевих користувачів у швидкому, надійному та зручному доступі до аналітичної інформації, а також забезпечувати можливість масштабування, інтеграції з іншими системами та подальшого розвитку функціоналу. Важливо, щоб архітектура застосунку була гнучкою, модульною та відповідала сучасним вимогам до продуктивності, безпеки та зручності використання.

Таким чином, розробка вебзастосунку є не лише технічним завданням, але й стратегічним кроком у напрямку цифровізації бізнес-процесів, що дозволяє компаніям ефективно працювати з даними, підвищувати конкурентоспроможність та адаптуватися до змін ринку.

ВИСНОВКИ З РОЗДІЛУ 1

У першому розділі визначено сутність та характеристики великих даних, окреслено масштаби їх зростання та виклики для бізнесу й науки. Розглянуто основні етапи їх обробки, що забезпечують перетворення «сирої» інформації у структуровані знання. Показано роль вебзастосунків як універсальних інструментів збору, зберігання, аналізу та візуалізації даних, а також їхню здатність до масштабування й інтеграції з різними джерелами.

Окремо підкреслено, що ефективність роботи з Big Data залежить не лише від технологій, а й від організації процесів, які забезпечують достовірність та актуальність результатів. Вебзастосунки у цьому контексті виступають ключовою платформою, що поєднує методи обробки даних із зручними засобами взаємодії з користувачем. Таким чином, вони стають важливим інструментом цифрової трансформації, дозволяючи приймати обґрунтовані управлінські рішення на основі комплексного аналізу інформації.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА МЕТОДИ РОЗРОБКИ ВЕБЗАСТОСУНКУ ЕЛЕКТРОННИХ ПРОДАЖІВ

2.1 Вибір архітектурного підходу та обґрунтування інструментів розробки

При проектуванні вебзастосунку, призначеного для аналізу великих обсягів даних у сфері електронної комерції, особливо важливо правильно вибрати архітектурний підхід і відповідні інструменти розробки. Архітектура системи визначає, яким чином взаємодіють її компоненти, наскільки масштабованою і зручною в супроводі буде система, як вона поведеться при високих навантаженнях, а також наскільки легко її можна буде розширювати або модифікувати в майбутньому. Вибір інструментів розробки, в свою чергу, впливає на швидкість реалізації проекту, стійкість системи до збоїв, а також на компроміси, які доводиться приймати в процесі розробки.

У даному розділі проводиться огляд архітектурних стилів, аналізуються їх сильні і слабкі сторони, а також обґрунтовується вибір можливих інструментів для реалізації серверної частини, клієнтського інтерфейсу, зберігання даних і візуалізації аналітичної інформації.

Архітектурні стилі: Монолітна, Мікросервісна та Модульна (Гібридна) архітектура.

Архітектурні стилі:

- Монолітна архітектура являє собою єдиний програмний блок, до якого входять всі основні компоненти: користувальницький інтерфейс, серверна логіка, доступ до даних і бізнес-логіка[19]. Всі ці елементи упаковуються і розгортаються як єдине ціле. Такий підхід вважається найбільш простим на початкових етапах розробки, особливо коли проект має обмежений масштаб, а команда розробників невелика. Монолітна архітектура дозволяє швидко приступити до реалізації, спростити тестування і прискорити процес розгортання. Однак у міру зростання проекту монолітна структура

починає демонструвати свої обмеження[20]. Будь-яка зміна в одному компоненті вимагає повторної збірки і розгортання всього додатка. Жорстка зв’язність між модулями може призвести до крихкості системи, а проблеми з продуктивністю в одній ділянці можуть негативно позначитися на всій системі. Крім того, масштабування моноліту ускладнене, особливо якщо необхідно обслуговувати велику кількість користувачів або обробляти великі обсяги даних.

- Мікросервісна архітектура передбачає розбиття додатка на незалежні сервіси, кожен з яких відповідає за певний функціональний контекст – наприклад, управління користувачами, аналітика продажів, генерація звітів тощо. Перевагами такого підходу є можливість масштабування окремих компонентів (наприклад, тільки тих, які відчувають високе навантаження), гнучкість у виборі технологій (різні сервіси можуть використовувати різні мови програмування та фреймворки), а також стійкість до збоїв (відмова одного сервісу не призводить до падіння всієї системи)[21]. Проте, мікросервісна архітектура значно ускладнює розробку і супровід. Необхідно реалізувати механізми міжсервісної взаємодії, організувати DevOps-процеси, забезпечити виявлення сервісів, а також більш складне тестування і налагодження. Такий підхід вимагає високої кваліфікації команди і додаткових ресурсів на інфраструктуру.

- Модульна (гібридна) архітектура є компромісом між монолітною і мікросервісною. Додаток будується у вигляді окремих модулів або шарів, кожен з яких має чітко визначені межі та інтерфейси, але при цьому розгортається як єдиний додаток (або як обмежена кількість компонентів). Такий стиль архітектури дозволяє зберегти керовану складність на початковому етапі, а в майбутньому – за необхідності – перейти до мікросервісної моделі. З огляду на те, що даний проект носить демонстраційний характер і являє собою прототип можливостей, а не повнофункціональний комерційний продукт, розрахований на тисячі одночасних користувачів, найбільш доцільним є вибір модульної монолітної

архітектури. Такий підхід забезпечує:

- швидшу початкову розробку;
- спрощене розгортання та супровід;
- зручність підтримки при обмеженому обсязі функціоналу;
- логічний поділ компонентів, що полегшує подальше розширення.

При цьому архітектурні рішення повинні прийматися таким чином, щоб у майбутньому була можлива міграція до мікросервісної моделі або поділ модулів. Це передбачає дотримання принципів слабкої зв’язності, чітко визначених інтерфейсів і поділу відповідальності між компонентами.

Ключові вимоги, що впливають на архітектуру. При виборі архітектурного підходу, не менш важливо, врахувати низку функціональних та нефункціональних вимог, які безпосередньо впливають на структуру системи, її поведінку, продуктивність та можливості масштабування. Ці вимоги формують основу для прийняття технічних рішень і визначають, наскільки ефективною, стабільною та гнучкою буде розроблена система. До цих умов можна віднести:

- Масштабованість. Система повинна бути здатною обробляти зростаючі обсяги даних, які можуть надходити з різних джерел, а також витримувати потенційне збільшення кількості користувачів. Це означає, що архітектура має передбачати можливість горизонтального або вертикального масштабування, використання кешування, балансування навантаження та оптимізацію запитів до бази даних.

- Відгук та інтерактивність. Інтерфейс користувача має забезпечувати швидке реагування на дії – фільтрацію, сортування, оновлення даних, побудову графіків тощо. Затримки повинні бути мінімальними, щоб користувач міг оперативно отримувати аналітичну інформацію без очікування. Це особливо важливо для дашбордів, які оновлюються в реальному або близькому до реального часу.

- Розширюваність. Архітектура має передбачати можливість додавання нових модулів – наприклад, нових типів аналітики, джерел даних, форматів

візуалізації або інтеграцій з іншими сервісами. Це забезпечується завдяки чітко визначеним інтерфейсам та слабкій зв’язності між компонентами.

- Гнучкість у роботі з різними форматами даних. Оскільки аналітика може охоплювати різні типи інформації – текстові поля, числові показники, часові ряди, категорії – архітектура повинна підтримувати роботу з напівструктурованими даними, а також забезпечувати можливість фільтрації, агрегації та трансформації даних у зручному форматі.

Усі перелічені вимоги мають бути враховані на етапі проєктування архітектури, оскільки вони визначають технічні обмеження, можливості розвитку системи та якість користувацького досвіду. Вибір архітектурного стилю повинен забезпечити баланс між простотою реалізації, продуктивністю, гнучкістю та можливістю масштабування в майбутньому.

Порівняння технологій серверної частини. Вибір технології для реалізації серверної частини вебзастосунку – це стратегічне рішення, яке впливає на продуктивність, масштабованість, зручність супроводу та швидкість розробки. У контексті аналітики даних особливо важливо враховувати, наскільки ефективно платформа справляється з обробкою запитів, взаємодією з базами даних та підтримкою API.

Розглянемо три підходи до побудови backend-частини:

- Node.js (JavaScript / TypeScript) – це платформа, заснована на JavaScript, яка ідеально підходить для обробки великої кількості одночасних запитів, особливо якщо вони пов’язані з операціями введення-виведення. Серед його переваг:

➢ висока швидкість розробки завдяки єдиній мові для фронтенду і бекенду[22];

➢ багата екосистема модулів і бібліотек;

➢ відмінна сумісність з візуалізаційними бібліотеками на JavaScript;

➢ активна спільнота і безліч готових рішень.

Однак Node.js має і обмеження:

➢ не є оптимальним для ресурсоємних обчислень, таких як складна

аналітика або машинне навчання;

- однопотокова модель вимагає особливої уваги до блокуючих операцій;
- необхідно ретельно керувати пам’яттю та обробляти помилки, щоб уникнути збоїв.

• Python (FastAPI, Django, Flask та інші) – це мова, яка широко використовується в науковому середовищі та аналітиці даних. Вона пропонує потужні інструменти для статистичного аналізу, обробки масивів даних і побудови моделей[23]. Її переваги:

- висока читабельність і простота коду;
- зріла екосистема для аналітики та машинного навчання;
- безліч бібліотек для роботи з даними (NumPy, Pandas, Scikit-learn);
- зручність інтеграції з візуалізацією та зовнішніми сервісами.

Недоліки Python:

- менш ефективний при високій конкуренції запитів;
- вимагає додаткових зусиль для реалізації асинхронної логіки;
- може вимагати більше ресурсів при масштабуванні.

• Гібридний підхід (Node.js + Python). В рамках гібридного підходу можна використовувати Python для виконання важких аналітичних завдань, а Node.js – для реалізації API і взаємодії з клієнтом. Переваги такого підходу:

- використання сильних сторін обох технологій;
- висока чуйність інтерфейсу;
- можливість масштабування аналітичних процесів окремо від логіки користувача.

Серед складнощів:

- необхідність підтримки декількох мов і технологій;
- ускладнення інфраструктури та налагодження;
- потреба в чіткій організації взаємодії між компонентами.

З огляду на демонстраційний характер проекту та його фокус на візуалізації та базовій аналітиці, найбільш раціональним вибором є

використання Node.js або TypeScript.

Особливості вибору інструментів для клієнтської частини та візуалізації даних. Клієнтська частина вебзастосунку відіграє ключову роль у забезпеченні зручності взаємодії користувача з системою, особливо коли мова йде про аналітику даних. Саме через інтерфейс користувач отримує доступ до візуалізованої інформації, фільтрує дані, відстежує динаміку показників і приймає рішення на основі представлених графіків, таблиць і діаграм. Тому вибір інструментів для фронтенду і візуалізації повинен бути ретельно продуманий з урахуванням вимог до інтерактивності, продуктивності, адаптивності та масштабованості.

При розробці інтерфейсу необхідно враховувати наступні аспекти:

- Підтримка динамічного оновлення даних – користувач повинен бачити актуальну інформацію без необхідності перезавантаження сторінки.
- Можливість фільтрації, сортування і деталізації – інтерфейс повинен дозволяти гнучко керувати відображуваними даними.
- Адаптивність – коректне відображення інтерфейсу на різних пристроях (ПК, планшет, смартфон).
- Розділення логіки – візуальна частина повинна бути відокремлена від логіки отримання та обробки даних, що спрощує супровід і тестування.

Візуалізація – це центральний елемент аналітичного вебзастосунку. Вона дозволяє перетворити числові та категоріальні дані в наочні графічні форми, полегшуючи сприйняття інформації та виявлення закономірностей. Існує безліч JavaScript-бібліотек, призначених для візуалізації, кожна з яких має свої особливості:

- Chart.js – одна з найпростіших і інтуїтивно зрозумілих бібліотек. Підходить для побудови стандартних графіків: лінійних, стовпчастих, кругових[24,25]. Відрізняється легкістю інтеграції та мінімальними вимогами до налаштування. Ідеально підходить для невеликих проектів і базової візуалізації.
- D3.js – потужна і гнучка бібліотека, що надає повний контроль над

візуальними елементами. Дозволяє створювати складні інтерактивні графіки, анімації, діаграми з високим ступенем кастомізації. Однак вимагає більш глибоких знань JavaScript і часу на реалізацію.

- ECharts – бібліотека, розроблена Apache, орієнтована на створення інтерактивних і масштабованих візуалізацій. Підтримує безліч типів графіків, включаючи теплові карти, діаграми зв'язків, часові ряди. Має вбудовані засоби взаємодії з користувачем і добре підходить для аналітичних панелей.
- Plotly.js – бібліотека з підтримкою 3D-графіків, статистичних діаграм і складних візуальних компонентів. Часто використовується в наукових і інженерних додатках, де потрібна висока точність і деталізація.
- Apache Superset – хоча це не бібліотека, а повноцінний BI-інструмент, він заслуговує на згадку. Superset дозволяє створювати інтерактивні дашборди, підключатися до різних джерел даних і керувати візуалізацією через веб-інтерфейс[26,27]. Може бути корисний при масштабуванні проекту.

Вибір фреймворка для фронтенду – не менш важливий етап. Для реалізації клієнтської частини вебзастосунку рекомендується використовувати компонентні фреймворки, такі як React, Vue.js або Svelte [28]. Вони забезпечують:

- зручну структуру коду;
- повторне використання компонентів;
- ефективну роботу зі станом додатку;
- легку інтеграцію з візуалізаційними бібліотеками.

Вибір конкретного фреймворка залежить від уподобань команди, вимог до продуктивності та складності інтерфейсу. React, наприклад, має широку підтримку і безліч готових рішень, Vue.js – легший та інтуїтивніший, а Svelte – відрізняється високою швидкістю та компактністю[29].

З всього вище сказаного, можна зробити висновок, що клієнтська частина вебзастосунку повинна бути побудована з урахуванням зручності користувача, гнучкості візуалізації та можливості масштабування. Використання сучасних JavaScript-фреймворків і візуалізаційних бібліотек

дозволяє створити потужний аналітичний інтерфейс, здатний ефективно представляти дані, забезпечувати інтерактивність і адаптуватися до різних сценаріїв використання. При цьому важливо дотримуватися балансу між функціональністю і складністю реалізації, особливо на етапі прототипування.

Обґрунтування вибору інструментів розробки. На підставі аналізу архітектурних підходів, вимог до системи та специфіки проекту, можна сформулювати попереднє обґрунтування вибору інструментів, які будуть використовуватися при розробці вебзастосунку для аналітики даних у сфері електронної комерції. Оскільки проект має демонстраційний характер і орієнтований на візуалізацію та базову обробку інформації, пріоритет надається простоті реалізації, гнучкості компонентів і можливості подальшого розширення функціоналу.

Обрана архітектура – модульна монолітна – забезпечує оптимальний баланс між простотою і можливістю масштабування. Вона дозволяє швидко приступити до розробки, не вимагає складної інфраструктури, але при цьому зберігає логічний поділ компонентів, що важливо для супроводу і майбутньої еволюції системи. Такий підхід особливо ефективний для невеликих команд і проектів з обмеженим обсягом функціоналу, де важлива швидкість реалізації і стабільність.

Для реалізації серверної логіки пропонується використовувати Node.js з можливим застосуванням TypeScript. Це рішення обумовлено наступними факторами:

- висока продуктивність при обробці великої кількості одночасних запитів;
- відмінна сумісність з клієнтською частиною, написаною на JavaScript;
- наявність безлічі готових бібліотек і модулів для роботи з API, базами даних, обробкою запитів;
- простота розгортання і супроводу.

Якщо в процесі розробки виникне необхідність у виконанні більш складних аналітичних операцій – наприклад, агрегації великих масивів даних

або статистичних розрахунків – можна додатково використовувати Python з його потужною екосистемою для аналізу даних. Такий гібридний підхід дозволить використовувати сильні сторони обох технологій без зайвого ускладнення архітектури.

Для реалізації користувацького інтерфейсу рекомендується використовувати сучасний компонентний фреймворк, такий як React або Vue.js. Ці інструменти забезпечують:

- модульність і повторне використання компонентів;
- ефективну роботу зі станом додатка;
- зручну інтеграцію з бібліотеками візуалізації;
- адаптивність інтерфейсу під різні пристрої.

Вибір між React і Vue залежить від уподобань команди та вимог до інтерфейсу. React має ширшу екосистему і підтримку, тоді як Vue простіший в освоєнні і легший в реалізації невеликих проектів. Оскільки наш проект не є великим і над його розробкою працює одна людина, як на мене, кращим варіантом буде використовувати Vue.

База даних. З урахуванням можливої роботи з напівструктурованими даними (наприклад, інформація про товари, клієнтів, замовлення), доцільно використовувати NoSQL-базу даних, таку як MongoDB. Вона забезпечує:

- гнучкість у структурі зберігання даних;
- можливість масштабування;
- зручність фільтрації та агрегації інформації.

При необхідності можна комбінувати NoSQL з реляційною базою (наприклад, PostgreSQL), якщо буде потрібна сувора структура для окремих модулів.

Для відображення аналітичних даних пропонується використовувати Chart.js – просту і легку бібліотеку, яка підходить для базових графіків і діаграм. У разі необхідності більш складної візуалізації можна підключити D3.js або ECharts, які забезпечують високий рівень кастомізації та інтерактивності.

Отже, в результаті маємо такий стек технологій – Node.js + Vue + MongoDB + Chart.js, який забезпечує:

- швидку розробку та простоту супроводу;
- гнучкість у візуалізації та обробці даних;
- можливість масштабування та розширення функціоналу;
- хорошу сумісність між компонентами.

Такий підхід дозволяє створити стабільний, зручний і функціональний вебзастосунок, який буде ефективно виконувати завдання аналітики даних і демонструвати можливості візуального представлення інформації в сфері електронної комерції.

2.2 Модель обробки та візуалізації даних

Модель обробки та візуалізації даних є фундаментальним компонентом вебзастосунку для аналітики продажів, що розробляється. Її основна мета – забезпечити ефективне отримання, трансформацію та графічне представлення великих обсягів даних. Ця модель визначає, яким чином необроблені дані, зібрані з різних джерел, обробляються, агрегуються[30] і представляються в інтуїтивній та інтерактивній формі для підтримки аналітичного прийняття рішень.

Концепція обробки даних у веб-аналітиці. У контексті веб-орієнтованих аналітичних систем обробка даних означає перетворення необроблених і часто неструктурованих даних у осмислену і структуровану форму, придатну для візуалізації. Розроблювана система працює переважно з транзакційними даними, включаючи записи про продажі, категорії товарів, ціни, сегменти клієнтів, маркетингові витрати і тимчасові показники. Застосунок слідує моделі клієнт-серверної взаємодії, при якій серверна частина або зовнішні API надають структуровані або не структуровані набори даних, а клієнтська частина відповідає за відображення їх аналітичних результатів.

Модель даних забезпечує ефективне управління великими та динамічними наборами даних на стороні клієнта. Отримання даних

здійснюється асинхронно через REST API-запити, а потім дані зберігаються в системі управління станом додатка[31]. Такий підхід дозволяє оновлювати візуальні елементи в реальному часі без перезавантаження сторінки, покращуючи продуктивність і користувацький досвід.

Методи попередньої обробки та трансформації. Перед візуалізацією дані проходять кілька етапів попередньої обробки, спрямованих на підвищення узгодженості, точності та читабельності. Сирі набори даних можуть містити неповні або дублюючі записи, некоректні типи даних або відсутні значення. Тому система включає механізми перевірки та нормалізації даних на стороні сервера. Попередня обробка даних включає:

- фільтрацію та перевірку – видалення порожніх або некоректних записів;
- перетворення типів – забезпечення коректного форматування числових і часових полів;
- агрегацію – групування транзакцій за часовими періодами;
- обчислення показників – розрахунок метрик, таких як загальний дохід, середня знижка або товари, що найбільше продаються.

Ці операції реалізуються з використанням функцій на TypeScript, що забезпечує типову безпеку і запобігає помилкам виконання при трансформації. В результаті дані стають чистими, узгодженими і готовими до візуалізації.

Модель візуалізації та використовувані інструменти. Модель візуалізації визначає, як оброблені дані перетворюються в графічні форми, що дозволяють користувачам швидко інтерпретувати складні закономірності та тренди. Візуалізація – це ключовий етап аналітики, оскільки вона перетворює числову інформацію в інтуїтивно зрозумілі графіки та діаграми.

У даному вебзастосунку компоненти візуалізації реалізовані з використанням сучасних бібліотек, таких як Chart.js, які забезпечують високий рівень налаштування та інтерактивності. Дашборд включає кілька основних типів візуалізації:

- лінійні графіки для відображення динаміки щомісячних продажів і

зростання виручки;

- стовпчасті діаграми для порівняння обсягів продажів за категоріями товарів або сегментами клієнтів;
- кругові діаграми для представлення розподілу найбільш успішних продуктів або регіонів;
- таблиці даних, що узагальнюють детальну інформацію на рівні транзакцій.

Архітектурна інтеграція моделі даних. Загальний робочий процес моделі обробки та візуалізації даних можна описати як послідовність взаємопов'язаних етапів:

- отримання даних – отримання наборів даних із серверної частини або зовнішніх API з використанням Axios [32];
- попередня обробка та трансформація – очищення та структурування даних для аналітичного використання;
- управління станом – зберігання оброблених даних у централізованому сховищі для реактивного доступу компонентами;
- візуалізація – відображення оброблених даних у графічній формі за допомогою повторно використовуваних компонентів Vue.

Така архітектура забезпечує модульність, супровідність і масштабованість. Кожен шар моделі виконує окрему функцію, що дозволяє незалежно розробляти, тестувати і розширювати компоненти.

2.3 Проектування структури бази даних

Структура бази даних є фундаментальною основою розробленого вебзастосунку для аналітики продажів, оскільки саме вона визначає, яким чином зберігається, організовується та пов'язується інформація про клієнтів, товари та транзакції. Від якості проектування бази даних залежить ефективність обробки запитів, швидкість виконання аналітичних операцій, гнучкість моделі при зміні вимог, а також масштабованість всієї системи.

З огляду на те, що проект працює з напівструктурованими даними і

орієнтований на можливість масштабування, в якості системи управління базами даних була обрана MongoDB[33]. Це сучасна документно-орієнтована NoSQL-база, яка дозволяє гнучко зберігати дані у форматі JSON-подібних документів і спрощує їх обробку при виконанні аналітичних обчислень і візуалізації.

Переваги документної моделі MongoDB. На відміну від традиційних реляційних баз даних, які використовують жорсткі схеми з заздалегідь визначеними зв'язками між таблицями, MongoDB зберігає інформацію у вигляді документів всередині колекцій [34]. Кожен документ є самостійною одиницею даних, що містить поля і значення, які можуть динамічно змінюватися[35]. Така модель підтримує еволюцію схеми, що особливо важливо в аналітичних додатках, де структура даних може змінюватися з часом – наприклад, при додаванні нових джерел інформації або метрик. MongoDB дозволяє:

- зберігати дані з довільною структурою;
- легко додавати нові поля без необхідності перестворення схеми;
- використовувати вкладені об'єкти і масиви;
- виконувати складні агрегації і фільтрації за допомогою вбудованого фреймворка.

Основні колекції в системі. В рамках розробленого додатка визначено три ключові колекції, кожна з яких відповідає за певний аспект даних:

Колекція клієнтів (Customers Collection). Містить загальну інформацію про клієнтів. Кожен документ включає унікальний ідентифікатор клієнта і країну проживання (див. рис. 2.1). Ці дані використовуються для сегментації продажів за географічними ознаками і підтримки регіональної аналітики[36].

```
const customerSchema = new Schema<ICustomer>({
  customerId: { type: String, required: true, unique: true },
  country: { type: String, required: true },
});
```

Рисунок 2.1 - Схема моделі клієнта

Колекція товарів (Products Collection). Зберігає відомості про кожен товар, що реалізується в системі. Документ включає код товару (stockCode), опис і ціну за одиницю (див. рис. 2.2). Ця колекція служить довідковим джерелом для зв’язування інформації про товар з конкретними транзакціями.

```
const productSchema = new Schema<IProduct>({
  stockCode: { type: String, required: true, unique: true },
  description: { type: String, required: true },
  unitPrice: { type: Number, required: true },
});
```

Рисунок 2.2 - Схема моделі товару

Колекція продажів (Sales Collection). Є центральним набором даних, що містить інформацію про всі транзакції. Кожен документ включає номер рахунку (invoiceNo), посилання на клієнта і товар (customerId і stockCode), а також такі параметри, як кількість, дата і ціна (див. рис. 2.3). Ці поля дозволяють проводити детальний аналіз динаміки продажів, поведінки клієнтів і ефективності товарів.

```
const saleSchema = new Schema<ISale>({
  invoiceNo: { type: String, required: true },
  customerId: { type: String, required: true },
  stockCode: { type: String, required: true },
  quantity: { type: Number, required: true },
  invoiceDate: { type: Date, required: true },
  unitPrice: { type: Number, required: true },
});
```

Рисунок 1.3 – Схема моделі продажу

Зв’язки між колекціями та денормалізація. Зв’язки між колекціями реалізуються за допомогою референсних полів (наприклад, customerId і stockCode), які служать логічними зв’язками, аналогічними зовнішнім ключам у реляційних базах. Такий підхід забезпечує гнучкість моделювання даних і дозволяє уникнути жорсткої пов’язаності, характерної для традиційних систем.

У випадках, коли потрібно прискорити операції читання, може застосовуватися денормалізація – тобто часткове вбудовування інформації про клієнта або товар безпосередньо в документ транзакції. Це дозволяє скоротити кількість запитів і підвищити продуктивність при виконанні аналітичних операцій.

Переваги обраної структури. Проектна структура бази даних має ряд ключових переваг:

- Висока масштабованість – MongoDB підтримує горизонтальне масштабування (sharding), що дозволяє обробляти великі обсяги транзакційних даних без втрати продуктивності.
- Гнучкість та еволюція схеми – колекції легко розширюються новими полями у міру зростання аналітичних вимог, без необхідності перестворення структури.
- Зручність інтеграції – формат даних BSON (розширений JSON) спрощує взаємодію між серверною частиною (Node.js) і клієнтською (Vue.js), забезпечуючи швидку передачу та обробку даних.

Ефективна аналітична обробка – структура підтримує агрегаційні запити, що дозволяють розраховувати ключові показники: загальний обсяг продажів за товарами, виручку по країнах, середню вартість транзакції тощо.

З впевненістю можна сказати, що розроблена структура бази даних забезпечує ефективну організацію, зберігання та вилучення аналітичної інформації, залишаючись при цьому гнучкою та адаптованою до майбутніх розширень системи. Надалі вона може бути доповнена новими сутностями – такими як постачальники, маркетингові кампанії, категорії транзакцій – без необхідності кардинальної переробки архітектури.

2.4 Модель взаємодії компонентів системи

Ефективна робота вебзастосунку для аналітики даних неможлива без чітко спроектованої моделі взаємодії між його основними компонентами. Така модель визначає, як клієнтська частина, серверна логіка і база даних

координують свої дії для забезпечення цілісності, продуктивності та масштабованості системи. У цьому розділі розглядається концептуальна схема взаємодії компонентів, описуються потоки даних, типи запитів і принципи синхронізації між шарами додатка.

Загальна архітектурна структура. Вебзастосунок, що розробляється, побудований за принципом трирівневої архітектури, що включає:

- Клієнтську частину (Frontend) – реалізовану з використанням компонентного JavaScript-фреймворку, а саме Vue.js, що відповідає за відображення інтерфейсу, візуалізацію аналітичних даних і взаємодію з користувачем.
- Серверну частину (Backend) – засновану на платформі Node.js, що забезпечує обробку запитів, виконання бізнес-логіки, агрегацію даних і взаємодію зі сховищем.
- Базу даних (Database Layer) – представлену документно-орієнтованою системою MongoDB, призначеною для зберігання інформації про клієнтів, товари, транзакції та інші сутності.

Кожен з цих компонентів виконує строго визначену роль, а їх взаємодія організована через стандартизовані інтерфейси та протоколи.

Потоки даних і логіка взаємодії. Взаємодія між компонентами здійснюється за такою логікою:

- Користувач ініціює дію через інтерфейс (наприклад, вибирає категорію товару).
- Клієнтська частина формує запит і відправляє його на сервер з використанням HTTP-протоколу.
- Серверна частина приймає запит, виконує необхідні операції: фільтрацію, агрегацію, розрахунок метрик.
- Для отримання даних сервер звертається до бази MongoDB, використовуючи драйвери і запити до колекцій.
- Після обробки даних сервер формує відповідь у форматі JSON і повертає її клієнту.

- Клієнтська частина отримує дані, зберігає їх у локальному сховищі стану і передає у візуалізаційні компоненти.
- Інтерфейс оновлюється без перезавантаження сторінки, забезпечуючи інтерактивність і чуйність.

Такий підхід дозволяє реалізувати реактивну модель, при якій будь-які зміни даних автоматично відображаються в інтерфейсі.

Типи взаємодій. У системі передбачені такі типи взаємодій:

- Запити на читання – отримання агрегованих даних для візуалізації (наприклад, обсяг продажів за регіонами).
- Запити на фільтрацію – динамічна зміна параметрів відображення (наприклад, вибір періоду або категорії).
- Запити на сортування – динамічна зміна параметрів сортування (наприклад, вибір поля для сортування).

UML-діаграма взаємодії компонентів. Для наочного представлення архітектурної моделі можна використовувати UML-діаграму компонентів, яка демонструє зв’язки між основними частинами системи (див. рис. 2.4).

Рисунок 2.4 – UML-діаграма компонентів

Принципи проектування взаємодії. При проектуванні моделі взаємодії компонентів були враховані наступні принципи:

- Модульність – кожен компонент реалізований як незалежний блок, що спрощує супровід і тестування.
- Слабка пов’язаність – взаємодія здійснюється через API, що дозволяє замінювати або масштабувати компоненти без впливу на інші частини

системи.

- Уніфікація форматів – всі дані передаються у форматі JSON, що забезпечує сумісність між шарами.
- Реактивність – зміни даних автоматично відображаються в інтерфейсі завдяки використанню сховища стану.

Отже, модель взаємодії компонентів системи забезпечує узгоджену роботу всіх шарів вебзастосунку, дозволяючи ефективно обробляти запити користувачів, керувати даними та візуалізувати аналітичну інформацію. Чіткий розподіл відповідальності, стандартизація форматів і реактивна архітектура роблять систему гнучкою, масштабованою та зручною для подальшого розвитку.

ВИСНОВКИ З РОЗДІЛУ 2

У другому розділі було розроблено архітектуру вебзастосунку для аналітики продажів у сфері електронної комерції. Обґрунтовано вибір модульної монолітної архітектури, яка забезпечує оптимальний баланс між простотою реалізації та можливістю масштабування системи. Визначено ключові компоненти серверної частини, що відповідають за обробку великих даних та формування агрегованих показників, а також клієнтської частини, яка реалізує інтерфейс для взаємодії користувача з аналітичними інструментами. Окремо підкреслено доцільність використання сучасного технологічного стеку (Node.js, Express.js, MongoDB, Mongoose, Vue.js, Chart.js), що забезпечує гнучкість, продуктивність та зручність розробки. Таким чином, у другому розділі було закладено фундаментальні технічні рішення, які визначають ефективність і надійність створеного вебзастосунку.

РОЗДІЛ 3
РОЗРОБКА ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ФУНКЦІОНУВАННЯ ВЕБЗАСТОСУНКУ

3.1 Структура та компоненти розробленої системи

Розроблена система являє собою демонстраційний вебзастосунок, призначений для візуалізації та аналізу великих обсягів даних, характерних для сфери електронної комерції. Основний акцент зроблений на серверній обробці інформації: агрегації, фільтрації, розрахунку метрик і наданні користувачеві готових аналітичних результатів. Користувач не взаємодіє з вихідними даними безпосередньо – він отримує доступ до заздалегідь завантаженої інформації, що зберігається в базі даних, і може аналізувати її за допомогою зручного інтерфейсу.

Архітектура застосунку побудована за принципом клієнт-серверної моделі. Нижче представлена схема, що ілюструє ключові компоненти системи та їх взаємодію (див. рис. 3.1).

Рисунок 2.1 – Схема ключових компонентів системи

На стороні клієнта використовується JavaScript-фреймворк Vue, який відповідає за відображення аналітичних даних і взаємодію з користувачем. Інтерфейс включає дашборд, таблиці, графіки та фільтри, що дозволяють зручно аналізувати ключові метрики.

Серверна частина додатка реалізована на платформі Node.js з використанням Express.js. Вона приймає HTTP-запити від клієнта, виконує бізнес-логіку, звертається до бази даних і повертає агреговані результати. Також обробляє запити, пов’язані з отриманням, фільтрацією та агрегацією даних. Реалізовано низку маршрутів, які забезпечують доступ до інформації про транзакції, товари та користувачів, що дозволяють виконувати аналітичні обчислення та формувати узагальнені метрики. Нижче наведено фрагмент коду, що демонструє обробку GET-запиту з динамічною фільтрацією і сортуванням (див. рис. 3.2).

Рисунок 3.2 – Фрагмент коду, що демонструє GET-запит з динамічною фільтрацією і сортуванням

Для отримання аналітики використовуються агрегуючі запити, основані на MongoDB Aggregation Framework[37], які групують дані за різними ідентифікаторами і розраховують потрібні метрики. Нижче наведено фрагмент коду, однієї з функцій, яка обраховує сумарний виторг та кількість (див. рис. 3.3).

```
router.get("/analytics/products", async (req, res) => {
  try {
    const data = await Sale.aggregate([
      {
        $group: {
          _id: "$productId",
          totalQuantity: { $sum: "$quantity" },
          totalRevenue: { $sum: { $multiply: ["$price", "$quantity"] } },
        },
      },
      { $sort: { totalRevenue: -1 } },
    ]);
    res.json(data);
  } catch (err) {
    res.status(500).json({ error: (err as Error).message });
  }
});
```

Рисунок 3.3 – Фрагмент коду, що обраховує сумарний виторг та кількість

Сховищем даних виступає документоорієнтована база MongoDB, яка ідеально підходить для зберігання транзакцій, інформації про клієнтів, товари та інші сутності. Гнучка структура колекцій дозволяє ефективно виконувати фільтрацію та агрегацію, що особливо важливо при роботі з великими обсягами інформації. На малюнку нижче представлена структура колекції Sale, що містить ключові поля: invoiceNo, customerId, stockCode, quantity, invoiceDate, unitPrice (див. рис. 3.4).

```
const saleSchema = new Schema<ISale>({
  invoiceNo: { type: String, required: true },
  customerId: { type: String, required: true },
  stockCode: { type: String, required: true },
  quantity: { type: Number, required: true },
  invoiceDate: { type: Date, required: true },
  unitPrice: { type: Number, required: true },
});
```

Рисунок 3.4 – Структура колекції Sale

Для візуального представлення даних на клієнтській стороні використовується бібліотека Chart.js, яка дозволяє будувати графіки, діаграми і таблиці. Це забезпечує наочність і зручність аналізу, особливо при роботі з часовими рядами і порівнянням показників.

Хоча на поточному етапі додаток не розгорнуто в хмарі, його архітектура передбачає можливість масштабування та розміщення на платформах типу AWS, GCP або Azure[38]. Це забезпечить відмовостійкість, гнучкість і доступність для реального використання в бізнес-середовищі.

Таким чином, система являє собою комплексне рішення, що складається з взаємопов'язаних компонентів: клієнтська частина забезпечує взаємодію з користувачем, серверна – обробку та агрегацію даних, база – надійне зберігання, а візуалізація – зручне представлення результатів. Така структура дозволяє ефективно працювати з великими обсягами інформації та демонструє практичні можливості аналітики у веб-середовищі.

3.2 Реалізація програмного продукту

Реалізацію програмного продукту слід розпочати з нагадування про ключові функціональні завдання, поставлені на етапі проектування. В рамках створюваного вебзастосунку необхідно було реалізувати наступні можливості:

- Розробка веб-інтерфейсу, що дозволяє відображати та аналізувати дані, що зберігаються в базі даних, з урахуванням їх структури та типів. Інтерфейс повинен бути простим, зрозумілим і зручним для користувача.
- Впровадження базових механізмів інтерактивної візуалізації, що забезпечують представлення ключових показників продажів у вигляді графіків, діаграм і таблиць.
- Надання можливості перегляду даних за різними категоріями – такими як товари, клієнти або типи замовлень – без необхідності складної фільтрації або сегментації.
- Реалізація функцій пошуку і сортування, що дозволяють

користувачеві швидко знаходити потрібну інформацію за ключовими параметрами.

Виходячи з цих вимог, був сформований технологічний стек та розроблена архітектура системи. Далі представимо функціонал нашого вебзастосунку в вигляді можливих запитів до серверу та продемонструємо їх за допомогою інструменту для тестування API Insomnia[39] (див. рис. 3.5).

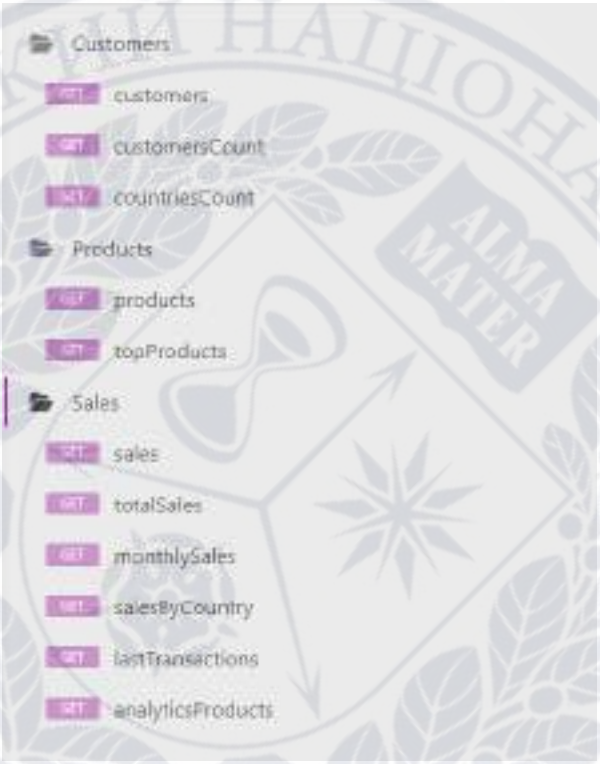


Рисунок 3.5 – Список майбутніх запитів до веб-серверу

Після демонстрації запитів до серверної частини веб-застосунку логічно перейти до опису структури бази даних, яка є основою для зберігання та обробки інформації. В рамках реалізації було спроектовано три основні колекції: Sale, Customer і Product. Кожна з них відповідає за окремий аспект бізнес-логіки електронної комерції.

Колекція Sale містить дані про транзакції. Кожен документ включає номер рахунку, ідентифікатор клієнта, код товару, кількість одиниць, дату покупки і ціну за одиницю (див. рис. 3.6). Така структура дозволяє ефективно виконувати агрегацію, фільтрацію і аналіз продажів за різними параметрами.

```
const saleSchema = new Schema<ISale>({
  invoiceNo: { type: String, required: true },
  customerId: { type: String, required: true },
  stockCode: { type: String, required: true },
  quantity: { type: Number, required: true },
  invoiceDate: { type: Date, required: true },
  unitPrice: { type: Number, required: true },
});
```

Рисунок 3.6 – Колекція Sale

Колекція Customer зберігає інформацію про клієнтів, включаючи унікальний ідентифікатор і країну проживання (див. рис. 3.6). Це дозволяє аналізувати географічний розподіл продажів і виконувати фільтрацію за регіонами.

```
const customerSchema = new Schema<ICustomer>({
  customerId: { type: String, required: true, unique: true },
  country: { type: String, required: true },
});
```

Рисунок 3.6 – Колекція Customer

Колекція Product містить опис товарів: код, найменування та ціну (див. рис. 3.7). Вона використовується для формування звітів про популярність продукції, середню вартість, а також для збагачення транзакційної інформації.

```
const productSchema = new Schema<IProduct>({
  stockCode: { type: String, required: true, unique: true },
  description: { type: String, required: true },
  unitPrice: { type: Number, required: true },
});
```

Рисунок 3.7 – Колекція Product

Всі три колекції мають чітко визначену структуру, що дозволяє легко інтегрувати їх в серверну логіку, виконувати запити з високою продуктивністю і забезпечувати масштабованість системи. В результаті ми

отримали таку логічну структуру колекцій нашої бази даних (див. рис. 3.8).

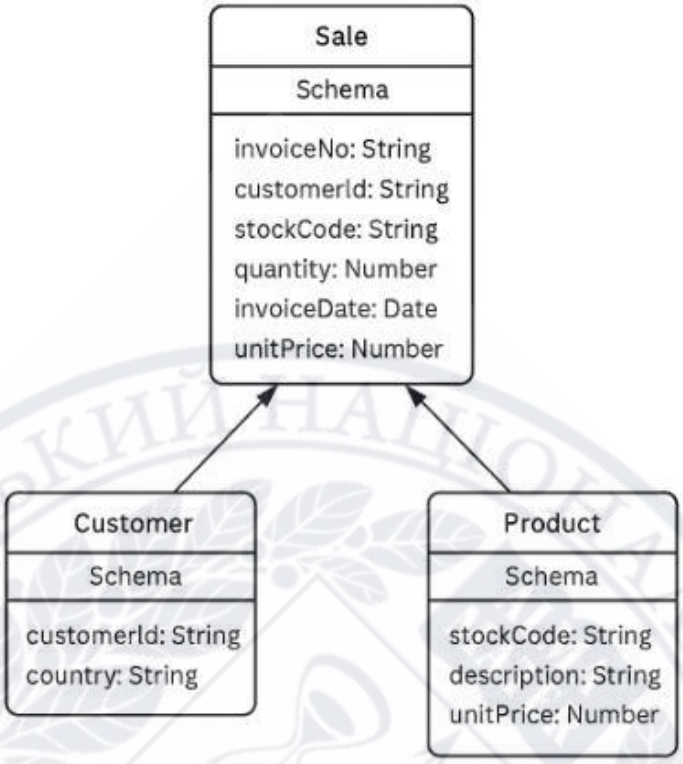


Рисунок 3.8 – Логічна структура колекцій бази даних

Після завершення проектування архітектури та визначення структури бази даних, наступним логічним кроком стає перехід до написання серверної частини застосунку. Розробка починається з ініціалізації проекту, створення базової структури та підключення необхідних інструментів.

Для початку було створено новий Node.js-проект за допомогою команди *npm init*, що дозволило сформувати файл package.json, який містить основну інформацію про додаток та його залежності[40]. На цьому етапі були встановлені ключові бібліотеки, включаючи:

- Express.js – для організації маршрутів і обробки HTTP-запитів.
- Mongoose – для взаємодії з MongoDB.
- dotenv – для роботи зі змінними середовища.
- cors – для спрощення налаштування заголовків CORS.

Такий набір інструментів забезпечує гнучкість, масштабованість і зручність в підтримці коду. В результаті ми отримуємо package.json файл, що показаний на рисунку 3.9.

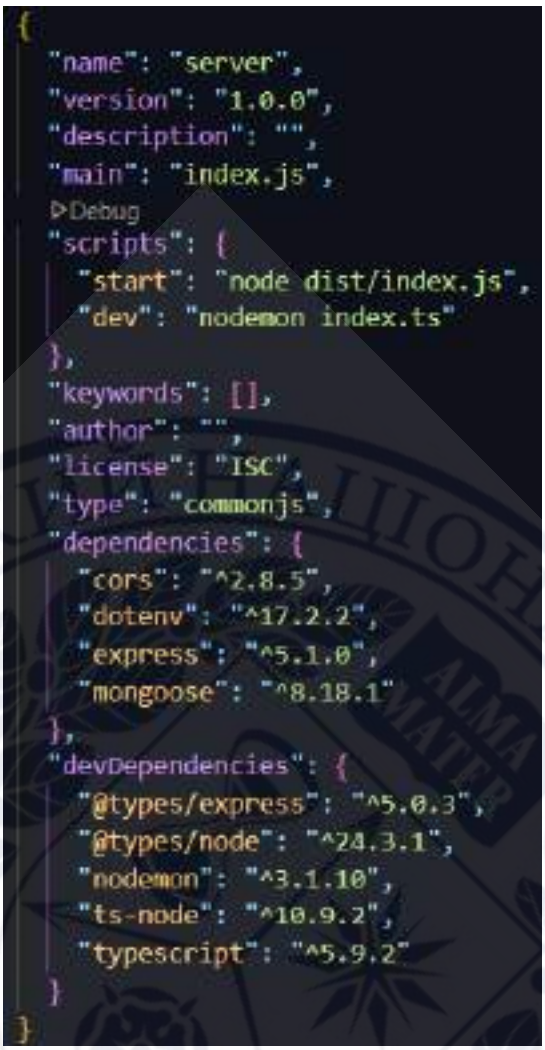


Рисунок 3.9 – Вигляд файлу package.json

Після встановлення залежностей було виконано налаштування середовища. У кореневій директорії проекту створено файл .env [41], в якому задані конфіденційні параметри, такі як URI підключення до бази даних і номер порту, на якому буде запускатися сервер. Це дозволяє легко перемикатися між середовищами розробки і продакшн, не змінюючи вихідний код. Приклад файлу .env показано на рисунку 3.10.

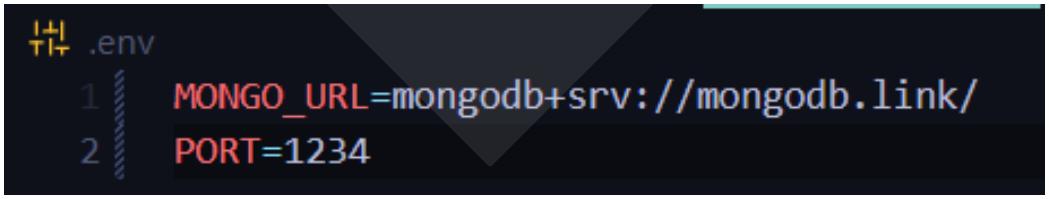


Рисунок 3.10 – Приклад файлу .env

Далі була сформована структура проекту, орієнтована на модульність і читабельність. У корені застосунку розміщені папки routes, models і utils кожна з яких відповідає за певну область логіки. Такий підхід дозволяє ізолювати бізнес-логіку від маршрутів, централізувати роботу з базою даних і спростити масштабування проекту в майбутньому.

У папці models були реалізовані схеми MongoDB для трьох основних колекцій: Sale, Customer і Product. Кожна схема описує структуру відповідного документа, включаючи обов'язкові поля, типи даних і унікальні ідентифікатори. MongoDB автоматично генерує унікальні _id для кожного документа, що спрощує процес зберігання і пошуку інформації [42]. Наприклад, схема Sale включає поля invoiceNo, customerId, stockCode, quantity, invoiceDate і unitPrice, які дозволяють зберігати інформацію про транзакції в структурованому вигляді. Схеми Customer і Product, в свою чергу, містять дані про клієнтів і товари, відповідно, і логічно пов'язані з колекцією Sale через поля customerId і stockCode.

Після визначення моделей даних наступним кроком стала реалізація логіки підключення до бази даних. Для цього був використаний інструмент Mongoose – популярна бібліотека для роботи з MongoDB в середовищі Node.js [43], яка надає зручний API для створення схем, виконання запитів і управління з'єднанням. Підключення до бази даних здійснюється через функцію *mongoose.connect()*, в яку передається URI, отриманий із змінних оточення. Це дозволяє гнучко керувати конфігурацією додатка, перемикаючись між локальною базою і віддаленим сервером без зміни вихідного коду.

Реалізація підключення до бази даних, показана на рисунку 3.11.

Особлива увага була приділена обробці подій підключення. У коді реалізовані слухачі, які відстежують успішне з'єднання (connected) і можливі помилки (error). Це дає можливість оперативно реагувати на збої, логувати проблеми і забезпечувати стабільну роботу додатка.

```
const PORT = process.env.PORT || 5000;

mongoose
  .connect(process.env.MONGO_URL!)
  .then(() => {
    console.log("MongoDB connected");
    app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
  })
  .catch((err) => console.error(err));
```

Рисунок 3.11 – Код для підключення до бази даних

Крім того, при запуску сервера виводиться повідомлення про статус підключення, див. рис. 3.12, що зручно для налагодження і моніторингу.

```
> server@1.0.0 dev
> nodemon index.ts

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node index.ts`
[dotenv@17.2.2] injecting env (2) from .env -- tip: ⚙ override existing env vars with { override: true }
MongoDB connected
Server running on port 5000
```

Рисунок 3.12 – Успішне підключення до бази даних

Після того як з’єднання з базою даних було успішно встановлено, розробка перейшла до створення маршрутів, відповідальних за обробку запитів від клієнта. Відповідно до принципів REST-архітектури [44], кожен маршрут реалізує конкретну операцію: отримання списку транзакцій, фільтрацію за параметрами, агрегацію даних, тощо. Для організації маршрутів була створена папка routes, в якій розміщені файли, що відповідають логічним блокам системи – наприклад, sales.routes.ts, customers.routes.ts, products.routes.ts (див. рис. 3.13).

Всередині кожного маршруту визначаються HTTP-методи [45], із зазначенням URL-шаблонів і параметрів запиту такі як:

- GET – використовується для отримання даних із сервера.
- POST – надсилає дані на сервер для обробки.

- PUT – використовується для повного оновлення або створення ресурсу на сервері.
- DELETE – видаляє ресурс із сервера.

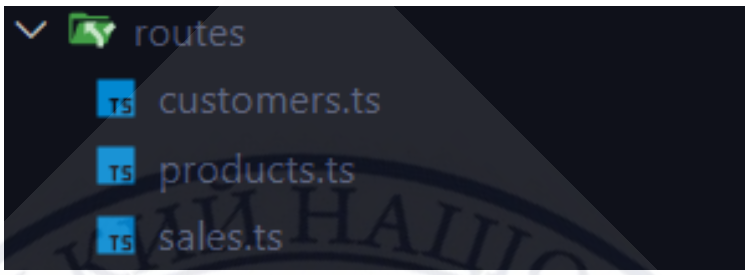


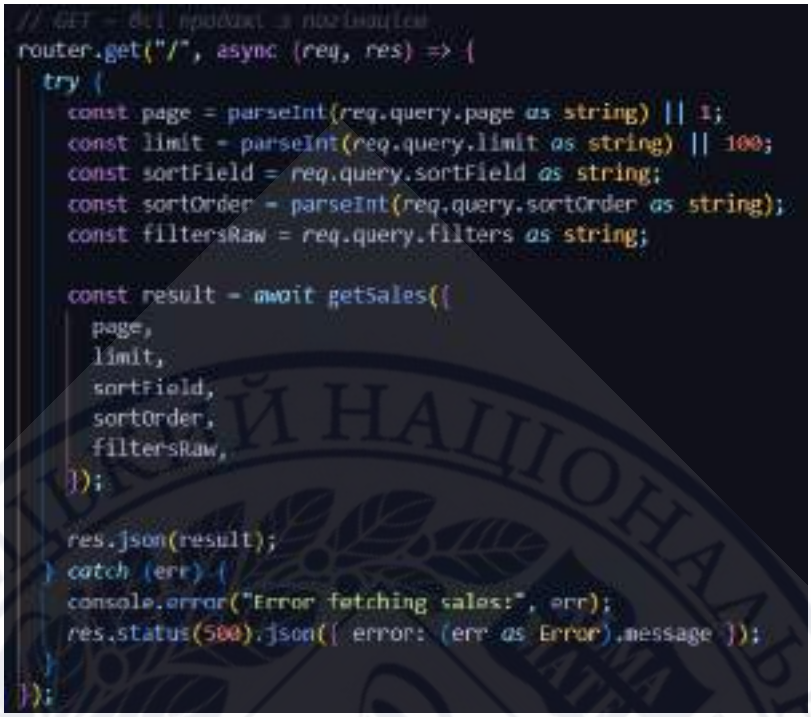
Рисунок 3.13 – Папка routes з файлами роутів

В нашому випадку використовується тільки GET метод. Наприклад, маршрут GET */api/sales* дозволяє отримати список транзакцій, а також підтримує передачу query-параметрів для фільтрації за номером транзакції, номером користувача, датою, товаром, кількістю одиниць або ціною. Для цього в коді реалізована логіка перевірки валідності параметрів, їх обробки та формування відповідного запиту до бази даних.

Важливим архітектурним рішенням стало винесення логіки обробки запитів в окремі утиліти, розміщені в папці utils. Це дозволяє централізувати бізнес-логіку, зробити код більш читабельним і полегшити його тестування. Кожна утиліта являє собою набір функцій, які приймають параметри запиту, виконують необхідні операції з базою даних і формують відповідь. Наприклад, утиліта getSales() обробляє запит на отримання транзакцій, застосовує фільтри, сортування і пагінацію, а потім повертає результат у форматі JSON, придатному для подальшої візуалізації на клієнтській частині.

Також, не менш важливим аспектом реалізації стала обробка помилок. У кожному запиті передбачені блоки try-catch, які дозволяють перехоплювати винятки, що виникають при виконанні запитів, і повертати інформативні повідомлення про помилки (див. рис. 3.14). Це особливо важливо при роботі із зовнішніми даними, де можливі ситуації відсутності записів, некоректних

параметрів або проблем з підключенням до бази.



```
// GET - всі продажі з пагінацією
router.get("/", async (req, res) => {
  try {
    const page = parseInt(req.query.page as string) || 1;
    const limit = parseInt(req.query.limit as string) || 100;
    const sortField = req.query.sortField as string;
    const sortOrder = parseInt(req.query.sortOrder as string);
    const filtersRaw = req.query.filters as string;

    const result = await getSales({
      page,
      limit,
      sortField,
      sortOrder,
      filtersRaw,
    });

    res.json(result);
  } catch (err) {
    console.error("Error fetching sales:", err);
    res.status(500).json({ error: (err as Error).message });
  }
});
```

Рисунок 3.14 – Приклад GET запиту для отримання всіх продаж

Одним з ключових функціональних елементів серверної частини є реалізація гнучкої системи фільтрації, пагінації та агрегації даних, яка дозволяє користувачеві отримувати релевантну інформацію в зручній формі. Для цього в маршрутах, що відповідають за обробку запитів, передбачена можливість передачі параметрів через query-рядок. Наприклад, при запиті списку транзакцій можна вказати код транзакції, код користувача, код товару, кількість одиниць, ціну та дату. Всі ці параметри обробляються на сервері і формують динамічний об’єкт фільтра, який потім використовується в запиті до MongoDB.

У реалізації утиліти використовується динамічне формування запиту, яке базується на перевірці кожного переданого параметра. Якщо значення фільтра є валідним, воно додається до об’єкта query, що використовується для звернення до бази даних. Такий підхід дозволяє уникнути зайвих умов і забезпечує високу гнучкість: наприклад, якщо користувач вказав лише один критерій – дату або ціну – фільтрація буде застосована виключно за ним. Це особливо актуально для аналітичних систем, де сценарії перегляду даних

можуть суттєво відрізнятися залежно від потреб користувача.

Для обмеження обсягу даних, що повертаються, реалізована пагінація. Вона дозволяє отримувати інформацію порціями, задаючи кількість записів на сторінку (limit) та номер сторінки (page). На сервері ці параметри перетворюються у значення skip та limit (див. рис. 3.15), які передаються у запит до бази даних. Це знижує навантаження на систему, покращує швидкість відповіді та дозволяє зручно реалізувати інтерфейс з переходом між сторінками.

```
const total = await Sale.countDocuments(query);
const sales = await Sale.find(query)
  .sort(sort)
  .skip((page - 1) * limit)
  .limit(limit);
```

Рисунок 3.15 – Фрагмент коду з пагінацією

Особлива увага була приділена агрегації даних – одному з найпотужніших інструментів MongoDB. За допомогою агрегаційного фреймворку можна не тільки фільтрувати записи, але й групувати їх за певними критеріями, обчислювати сумарні значення, середні показники, а також формувати структуровані звіти. Наприклад, для аналізу продажів по країнах використовується стадія $group[46] (див. рис. 3.16), яка об’єднує транзакції по полю country і обчислює загальну суму продажів, кількість замовлень і середню ціну. Такі запити дозволяють отримати узагальнену інформацію, яка є основою для побудови графіків та діаграм на клієнтській частині.

Для побудови складних запитів використовується комбінація стадій $match, $group, $sort, $project, що дозволяє не лише агрегувати дані, але й форматувати їх перед поверненням [47]. Наприклад, можна відфільтрувати транзакції за певний період, згрупувати їх за товарами, відсортувати за кількістю продажів і повернути лише необхідні поля – назву товару, кількість, загальну суму. Такий підхід дозволяє реалізувати гнучку аналітику без

необхідності додаткової обробки на клієнті.

Рисунок 3.16 – Фрагмент коду з утиліти getSalesByCountry

Важливо зазначити, що всі запити до бази даних реалізовані з урахуванням продуктивності. Для цього використовуються індекси на полях customerId, stockCode, invoiceDate, що дозволяє значно прискорити виконання фільтрації та агрегації. Крім того, в коді передбачено обмеження на максимальну кількість записів, що повертаються, а також обробку ситуацій, коли запит не дає результату – в таких випадках сервер повертає інформативне повідомлення про відсутність даних. Завдяки реалізованій системі запитів, додаток здатний ефективно обробляти великі обсяги інформації, забезпечуючи гнучкість, високу швидкість відгуку і точність аналітичних операцій. Це створює міцну основу для подальшої візуалізації даних і побудови зручного користувальницького інтерфейсу.

В результаті виконаної роботи над серверною частиною додатка була сформована чітка і логічно побудована структура файлів і папок, що відображає архітектурні принципи проекту. Така організація коду забезпечує зручність навігації, розподіл відповідальності між модулями і легкість

масштабування при додаванні нового функціоналу. У кореневій директорії проекту розташовані основні папки:

- routes – містить маршрути для обробки HTTP-запитів.
- models – описує структуру колекцій MongoDB.
- utils – включає бізнес-логіку, взаємодію з моделями та допоміжні функції.

Крім того, в корені проекту знаходяться конфігураційні файли tsconfig.json, .env, package.json, що відповідають за налаштування середовища, залежності та змінні середовища. На рисунку 3.17 нижче представлений фрагмент структури проекту в середовищі розробки VS Code.

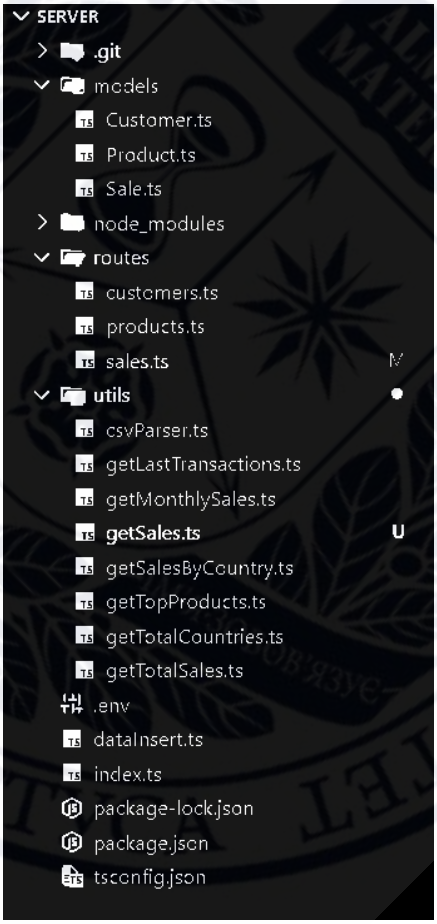


Рисунок 3.17 – Структура проекту в середовищі розробки VS Code

Отже, серверна частина додатка побудована на принципах модульності, структурної ясності та масштабованості. Кожен маршрут відповідає за строго визначене завдання, а логіка обробки винесена в окремі утиліти, що спрощує

супровід і тестування. Взаємодія з базою даних здійснюється через Mongoose, що гарантує надійну і адаптивну роботу всієї системи.

Після завершення реалізації серверної логіки було проведено тестування API, метою якого була перевірка коректності обробки запитів, стабільності маршрутів і відповідності повернутих даних заданим параметрам. Для тестування використовувалося програмне забезпечення – Insomnia, яке дозволяє виконувати HTTP-запити, передавати query-параметри, а також аналізувати структуру відповідей. В рамках тестування були по черзі перевірені ключові маршрути серверної частини. Першим було протестовано маршрут, відповідальний за отримання списків транзакцій. У запиті були передані параметри limit та page, що дозволило отримати дані для 11 сторінки з лімітом в 100 записів на сторінці (див. рис. 3.18).

Рисунок 3.18 – Запит на отримання списка транзакцій, параметри та відповідь

Відповідь сервера містить загальну кількість об’єктів, поточну сторінку, ліміт записів та масив об’єктів, що відповідають заданим критеріям. Структура відповіді є коректною, всі поля присутні, формат даних відповідає очікуванням.

Далі був перевірений запит на отримання списка товарів з сортуванням. У запиті були передані параметри page, limit, sortField та sortOrder, що дозволило отримати дані для 1 сторінки з лімітом в 20 записів та

відсортованими даними в порядку зростання, за полем unitPrice (див. рис. 3.19).

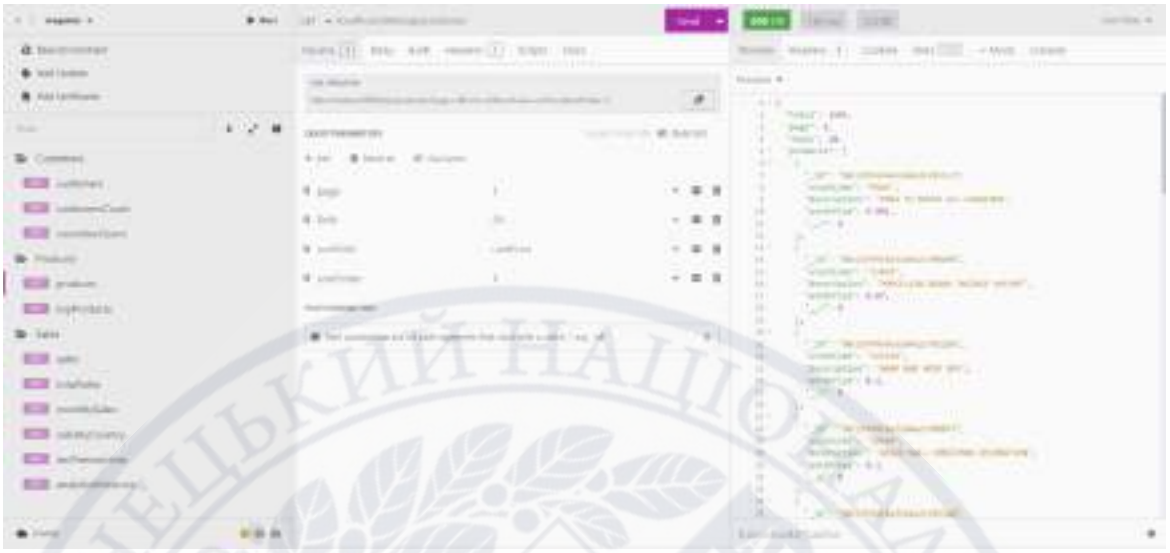


Рисунок 3.19 - Запит на отримання списка товарів, параметри та відповідь

Дані відсортовані коректно, що підтверджує правильну реалізацію логіки сортування на сервері. Для отримання всіх клієнтів був зроблений запит до маршруту *api/customers*, щоб перевірити його коректність. У запиті були передані параметри page, limit, filters, що дозволило отримати дані для 2 сторінки з лімітом в 20 записів та відфільтрованими даними за полем country із значенням Germany (див. рис. 3.20).

Рисунок 3.20 – Запит на отримання списка клієнтів, параметри та відповідь

Відповідь сервера містить всі необхідні дані, що відповідають заданим критеріям. Структура відповіді є коректною, всі поля присутні, формат даних відповідає очікуванням.

Маршрут *api/customers/countriesCount* дозволяє агрегувати дані та отримати кількість унікальних країн клієнтів (див. рис. 3.21).

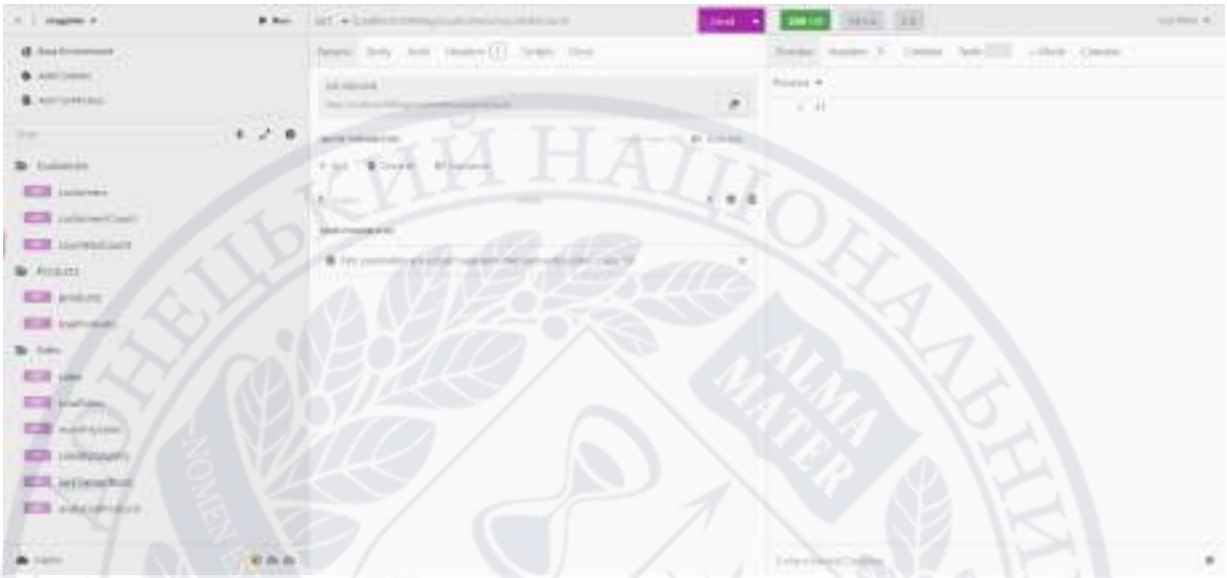


Рисунок 3.21 – Запит на отримання кількості країн та відповідь

В результаті запита повертається число, а саме, кількість країн. Структура відповіді, також, є коректною.

Маршрут *api/customers/customersCount* дозволяє отримати дані з кількістю унікальних клієнтів (див. рис. 3.22).

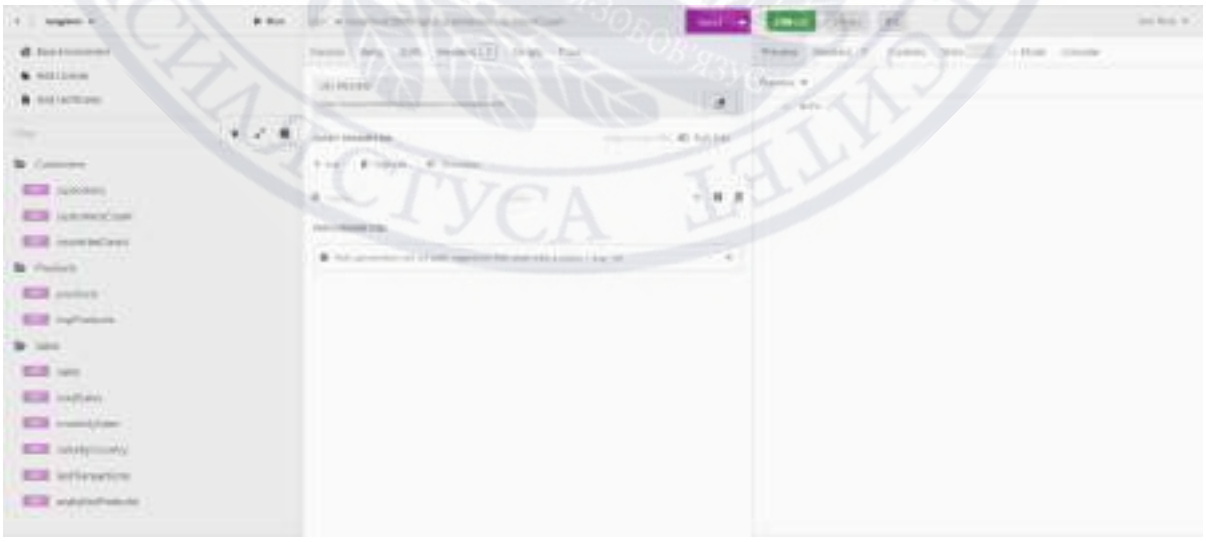


Рисунок 3.22 – Запит на отримання кількості клієнтів та відповідь

Наступним було перевірено запит на отримання списку найкращих товарів. В результаті отримали масив об’єктів, які відповідають визначеній структурі (див. рис. 3.23).

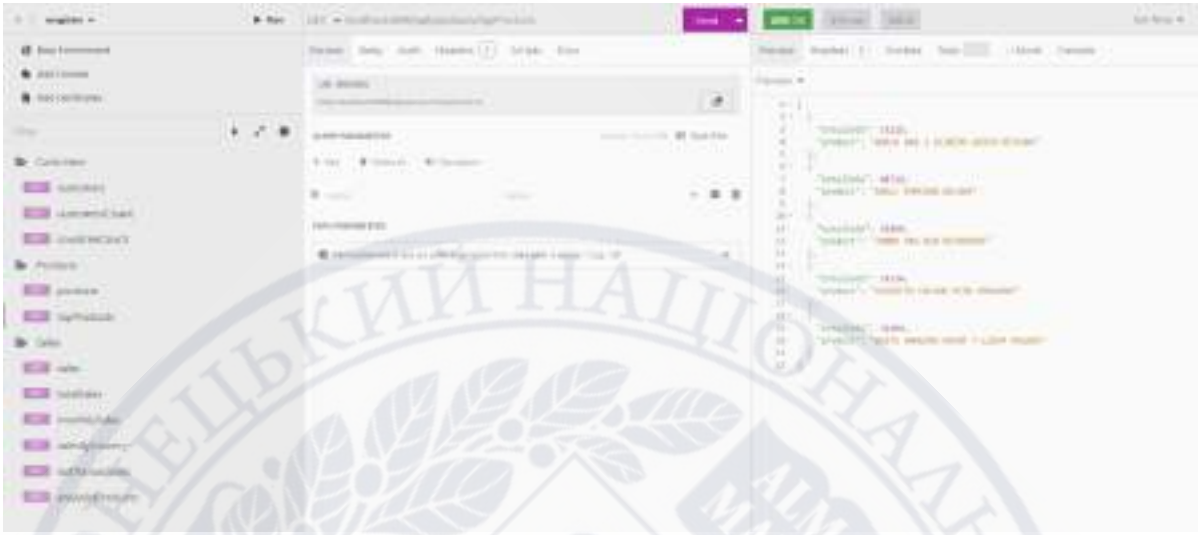


Рисунок 3.23 – Запит на отримання списку найкращих товарів та відповідь

Наступною перевіркою став запит на отримання суми всіх продажів. В результаті повинно повернутись число, а саме сума, що ми і бачимо на рисунку (див. рис. 3.24).

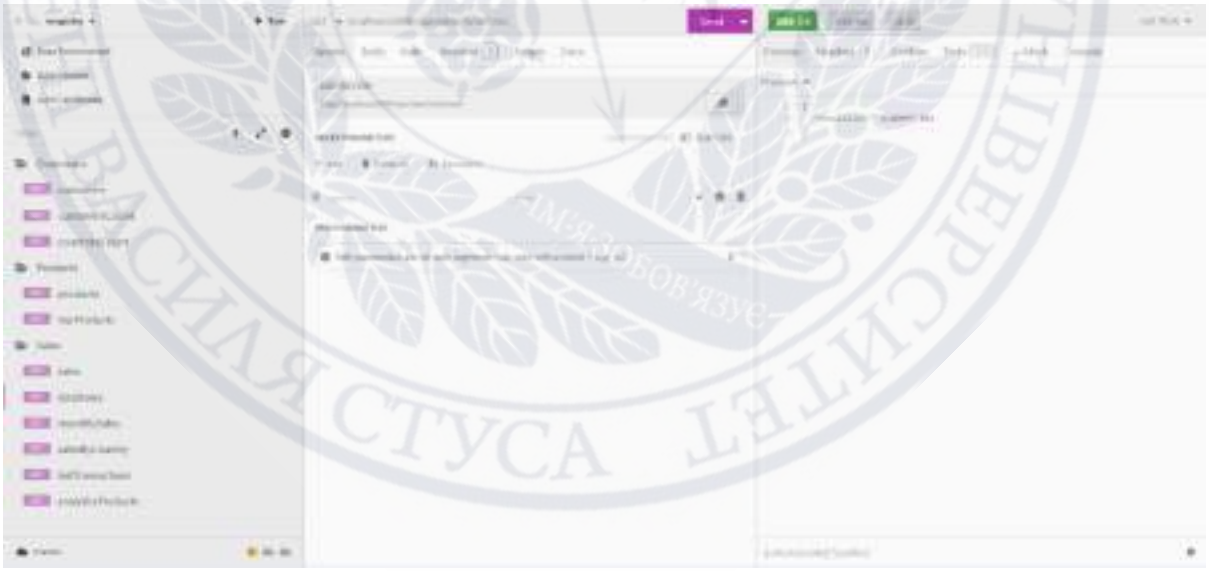


Рисунок 3.24 – Запит на отримання суми всіх продажів та відповідь

Аналогічно був перевірений запит на отримання кількості продажів по місячно. В відповіді отримуємо масив об’єктів, який містить місяць та кількість продажів (див. рис. 3.25).

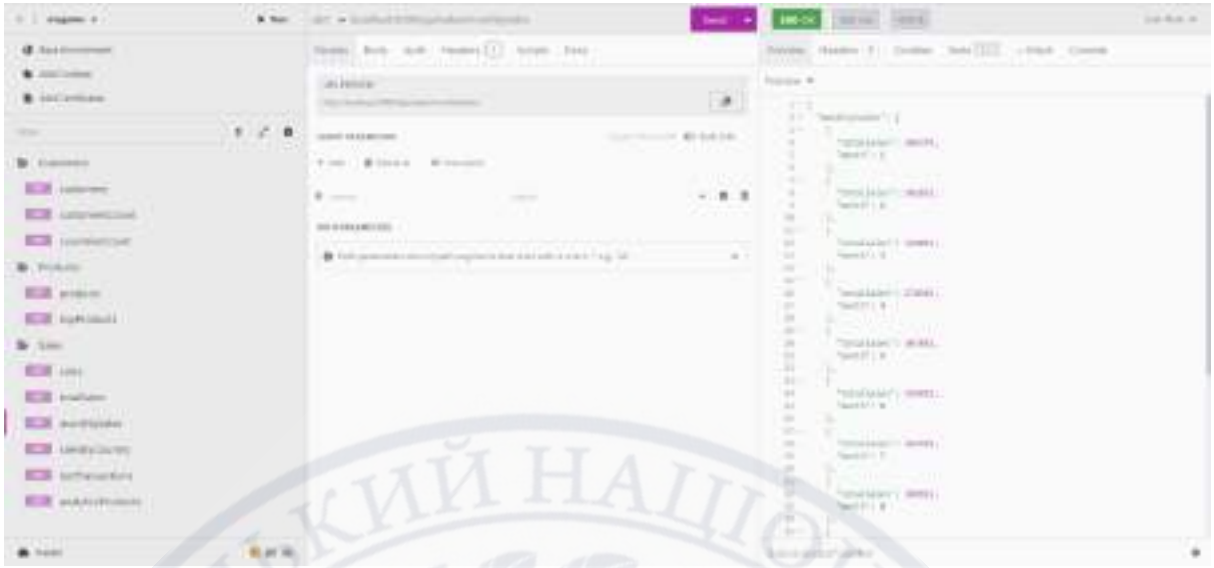


Рисунок 3.25 – Запит на отримання кількості продажів по місячно та відповідь

Далі було перевірено маршрут, відповідальний за отримання продаж по країнам. Відповідь сервера повинна містити масив об’єктів з двома полями: назва країни та кількість продаж (див. рис. 3.26).

Рисунок 3.26 – Запит на отримання кількості продажів по країнам та відповідь

На останок перевіримо маршрут, відповідальний за отримання списка 10 останніх транзакцій. В відповіді від сервера отримуємо масив з 10 об’єктів, які містять відповідні поля визначені в схемі Sale (див. рис. 3.27).

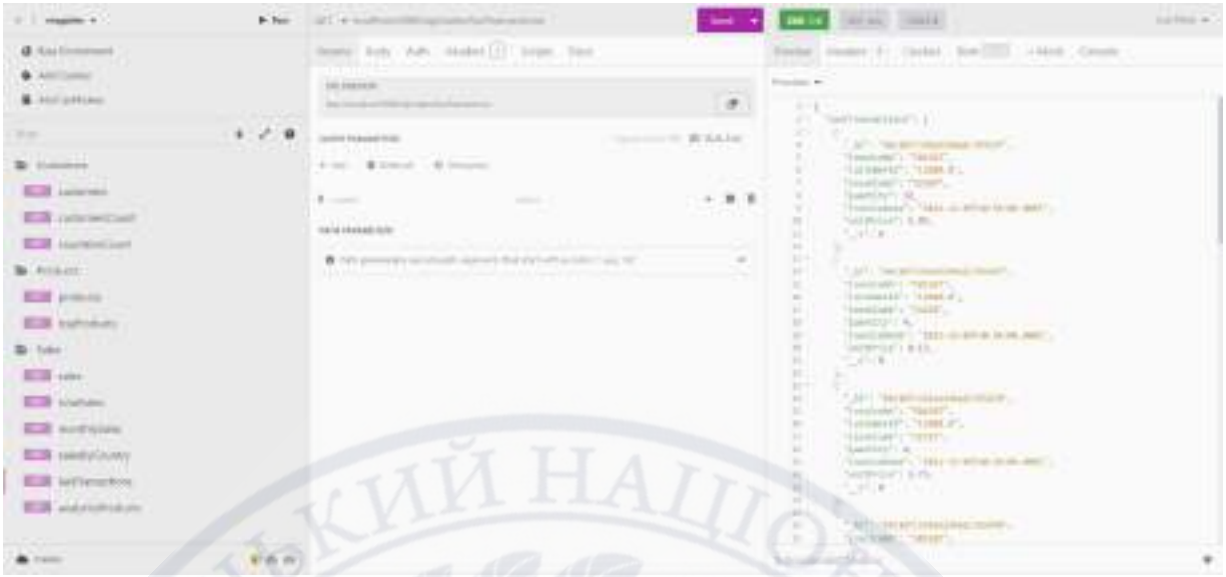


Рисунок 3.27 – Запит на отримання 10 останніх транзакцій та відповідь

Таким чином, тестування API підтвердило коректність реалізації всіх запитів, стабільність роботи серверної частини та відповідність отриманих даних очікуваним параметрам. Кожен запит був перевірений на предмет обробки фільтрації, пагінації, агрегації та виняткових ситуацій, що дозволяє зробити висновок про готовність серверної логіки до інтеграції з клієнтським інтерфейсом.

Далі розглянемо реалізацію фронтенд частини застосунку, яка відповідає за візуалізацію отриманих даних, забезпечення взаємодії користувача з системою та побудову аналітичних компонентів інтерфейсу.

Клієнтська частина вебзастосунку відповідає за візуалізацію аналітичних даних, отриманих з сервера, а також за забезпечення інтерактивної взаємодії користувача з системою. Вона реалізована з використанням сучасного JavaScript-фреймворку Vue.js, який забезпечує реактивність, компонентну архітектуру і високу продуктивність інтерфейсу.

На початковому етапі розробки був створений проект за допомогою інструменту Vite – легкого і швидкого збирача, орієнтованого на сучасні веб-стандарти[48]. Vite забезпечує миттєвий запуск середовища розробки, швидку збірку і зручну інтеграцію з Vue 3, що значно прискорює процес початкового налаштування.

Після ініціалізації проекту були встановлені ключові бібліотеки, необхідні для реалізації клієнтської логіки, взаємодії з API, побудови графіків і управління станом додатка. До основних залежностей відносяться:

- vue – основний фреймворк для побудови інтерфейсу
- axios – інструмент для виконання HTTP-запитів
- vue-router – бібліотека маршрутизації між сторінками
- primevue – бібліотека компонентів для полегшення та пришвидшення процесу розробки
- pinia – сучасне сховище стану, альтернатива Vuex.

Повний список залежностей показаний в файлі package.json (див. рис. 3.28).

Рисунок 3.28 – Файл package.json

Після встановлення необхідних бібліотек була сформована структура

проекту, що відповідає принципам модульності, розподілу відповідальності та зручності супроводу (див. рис. 3.29). У кореневій директорії клієнтської частини розташовані такі основні папки:

- components – повторно використовувані компоненти та окремі сторінки застосунку;
- services – модулі для взаємодії з API;
- store – конфігурація сховища Pinia;
- router – налаштування маршрутизації;
- utils – додаткові функції.

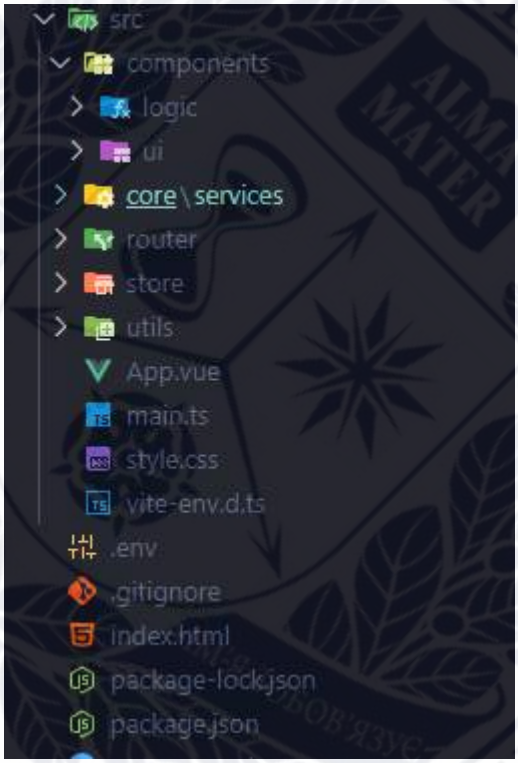


Рисунок 3.29 – Структура проекту

Така організація проекту забезпечує логічний розподіл логіки, спрощує навігацію по коду і сприяє масштабованості додатка.

Після формування структури клієнтської частини наступним логічним етапом стала реалізація механізму взаємодії з серверною частиною додатка. Для цього в проекті була виділена окрема папка *services*, в якій розміщуються модулі, відповідальні за обробку HTTP-запитів і централізовану роботу з API.

Ключовим елементом даної папки є файл ApiService.ts, який інкапсулює

базову логіку для виконання HTTP-запитів з використанням бібліотеки axios. Реалізація виконана у вигляді статичного класу, що дозволяє централізовано ініціалізувати інтеграцію з axios та повторно використовувати методи запитів у будь-якому компоненті додатка (див. рис. 3.30).

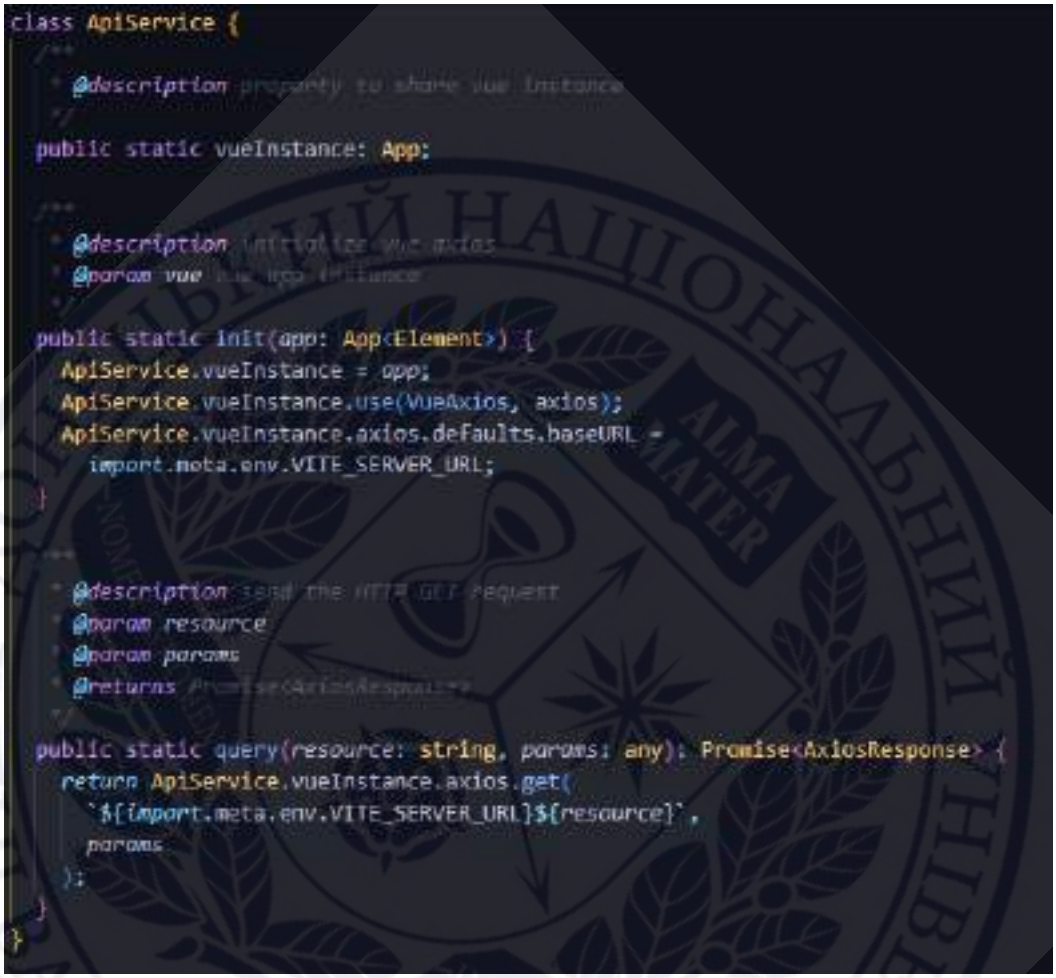


Рисунок 3.30 – Лістинг класу ApiService

Метод *init()* викликається під час ініціалізації додатка і виконує підключення axios як плагіна Vue, а також встановлює базовий URL для всіх наступних запитів. Значення базової адреси береться зі змінної середовища VITE_SERVER_URL, що дозволяє легко змінювати адресу сервера без модифікації коду.

Метод *query()* реалізує універсальний механізм для виконання GET-запитів до будь-якого ресурсу. Він приймає шлях до ресурсу та об’єкт параметрів, які передаються у вигляді query рядків. Такий підхід дозволяє централізовано керувати логікою запитів, спрощує обробку помилок і

забезпечує гнучкість при масштабуванні функціоналу.

Таким чином, реалізація класу ApiService дозволила централізувати логіку взаємодії з серверною частиною і спростити структуру HTTP-запитів в клієнтській частині. Однак для ефективного управління отриманими даними, їх збереження та повторного використання в різних компонентах інтерфейсу, виникла необхідність у впровадженні механізму глобального стану.

З цією метою в проєкті було використано Pinia – офіційне сховище стану для Vue 3 [49], яке забезпечує реактивність, просту інтеграцію з компонентами та зручну структуру для масштабування. В межах клієнтської частини було створено кілька окремих сторінок, кожна з яких відповідає за певну категорію даних: клієнти, товари та транзакції. Нижче розглянемо структуру та функціональність кожного з них.

Стор *customers* відповідає за зберігання та обробку інформації про клієнтів, а також за отримання статистичних даних, пов’язаних із країнами.

Основні елементи (див. рис. 3.31):

- customers – масив клієнтів;
- customersCount – загальна кількість клієнтів;
- countriesCount – кількість унікальних країн;
- isLoading – індикатор завантаження.

```
export const useCustomerStore = defineStore("customers", () => {
  // State
  const customers = ref<Customer[]>([]);
  const customersCount = ref();
  const countriesCount = ref();
  const isLoading = ref(true);
```

Рисунок 3.31 – Змінні сховища стану customers

Методи (actions) включають (див. рис. 3.32):

- getCustomers() – отримання списку клієнтів з параметрами фільтрації;
- getCustomersCount() – запит загальної кількості клієнтів;
- getCountriesCount() – запит кількості країн, представлених у базі.

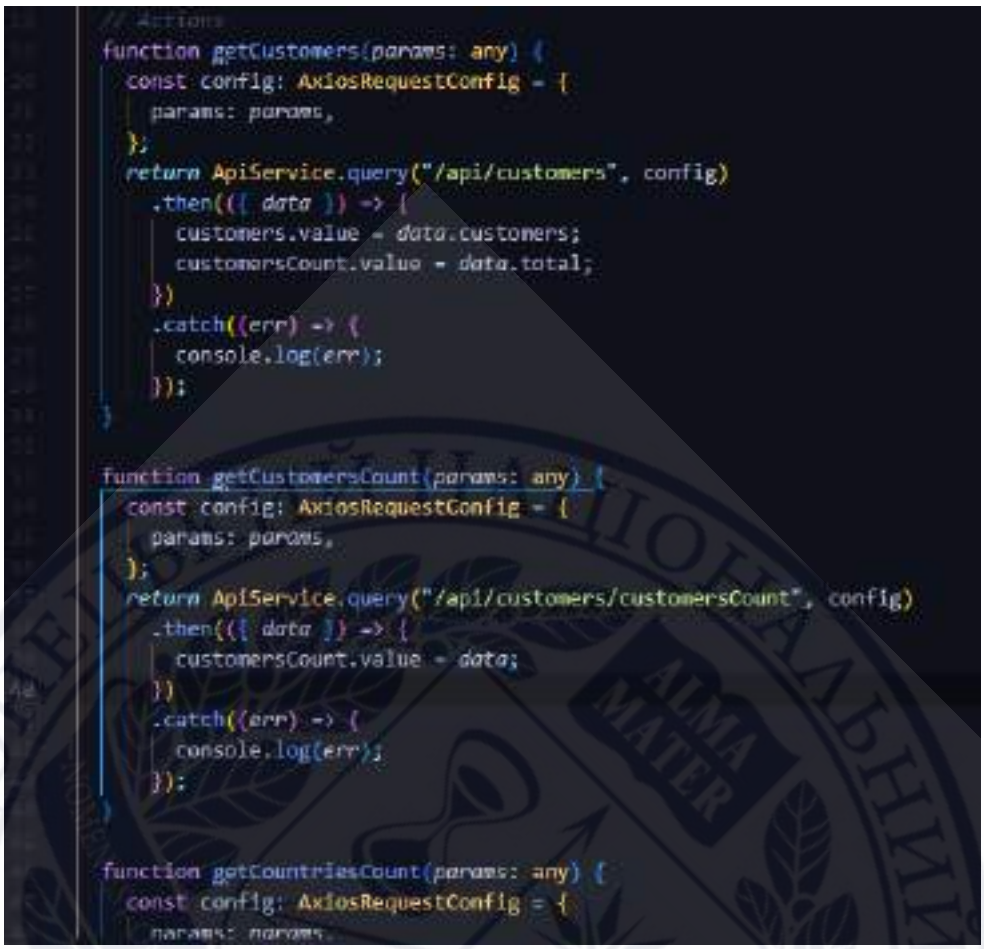


Рисунок 3.32 – Методи сховища стану customers

Усі запити виконуються через ApiService, а отримані дані зберігаються у відповідних реактивних змінних.

Стор *products* відповідає за управління даними про товари, включаючи їх опис, ціну та статистику продажів. Основні елементи стану (див. рис. 3.33):

- products – масив товарів;
- productsCount – загальна кількість товарів;
- topProducts – масив найпопулярніших товарів за кількістю продажів;
- isLoading – індикатор завантаження.

```
export const useProductsStore = defineStore("products", () => {
  // State
  const products = ref<Products[]>([]);
  const productsCount = ref();
  const topProducts = ref<TopProducts[]>([]);
  const isLoading = ref(true);
```

Рисунок 3.33 – Основні елементи сховища products

Методи (див. рис. 3.34):

- getProducts() – отримання списку товарів з параметрами фільтрації;
- getTopProducts() – запит агрегованих даних про найпопулярніші товари.

Рисунок 3.34 – Методи сховища products

Стор *transactions* є найбільш функціонально насиченим і відповідає за обробку транзакцій, а також за отримання агрегованих аналітичних даних.

Основні елементи стану (див. рис. 3.35):

- transactions – масив транзакцій;
- transactionsCount – загальна кількість транзакцій;
- totalSales – загальна сума продажів;
- monthlySales – динаміка продажів по місяцях;
- salesByCountry – обсяг продажів по країнах;
- lastTransactions – останні 10 транзакцій;
- isLoading – індикатор завантаження.

```
export const useTransactionsStore = defineStore("transactions", () => {
  // State
  const transactions = ref<Transactions[]>([]);
  const totalSales = ref();
  const monthlySales = ref();
  const transactionsCount = ref();
  const salesByCountry = ref();
  const lastTransactions = ref();
  const isLoading = ref(true);
```

Рисунок 3.35 – Основні елементи сховища transactions

Методи (див. рис. 3.36):

- getTransactions() – отримання списку транзакцій з фільтрацією, пагінацією та сортуванням;
- getTotalSales() – запит загальної суми продажів;
- getMonthlySales() – отримання динаміки продажів по місяцях;
- getSalesByCountry() – агреговані дані по країнах;
- getLastTransactions() – останні транзакції.

```
// Actions
function getTransactions(params: any) {
  const config: AxiosRequestConfig = {
    params: params,
  };
  return ApiService.query("/api/sales", config)
    .then(({ data }) => {
      transactions.value = data.sales;
      transactionsCount.value = data.total;
    })
    .catch((err) => {
      console.log(err);
    });
}

function getTotalSales(params: any) {
  const config: AxiosRequestConfig = {
    params: params,
  };
  return ApiService.query("/api/sales/totalSales", config)
    .then(({ data }) => {
      totalSales.value = data.totalSales;
    })
    .catch((err) => {
      console.log(err);
    });
}

function getMonthlySales(params: any) {
  const config: AxiosRequestConfig = {
    params: params,
  };
```

Рисунок 3.36 – Декілька методів сховища transactions

Цей стор є центральним джерелом даних для таблиць, графіків та загальних показників у застосунку.

Завдяки використанню Pinia, усі компоненти додатку мають доступ до реактивного стану, що дозволяє ефективно синхронізувати інтерфейс з даними, отриманими з API.

Перш ніж перейти до опису реалізації візуальних компонентів інтерфейсу, таких як таблиці та графіки, слід коротко відзначити наявність допоміжних утиліт, які використовуються в різних частинах клієнтського додатка.

У відповідній директорії utils були реалізовані допоміжні функції, покликані спростити обробку даних і підвищити читабельність коду. Серед них:

- debounce – функція , яка обмежує частоту виклику певних дій, наприклад, запитів до API при зміні параметрів фільтрації (див. рис. 3.37). Це дозволяє зменшити навантаження на сервер і уникнути надмірної кількості запитів при активній взаємодії користувача з формами.

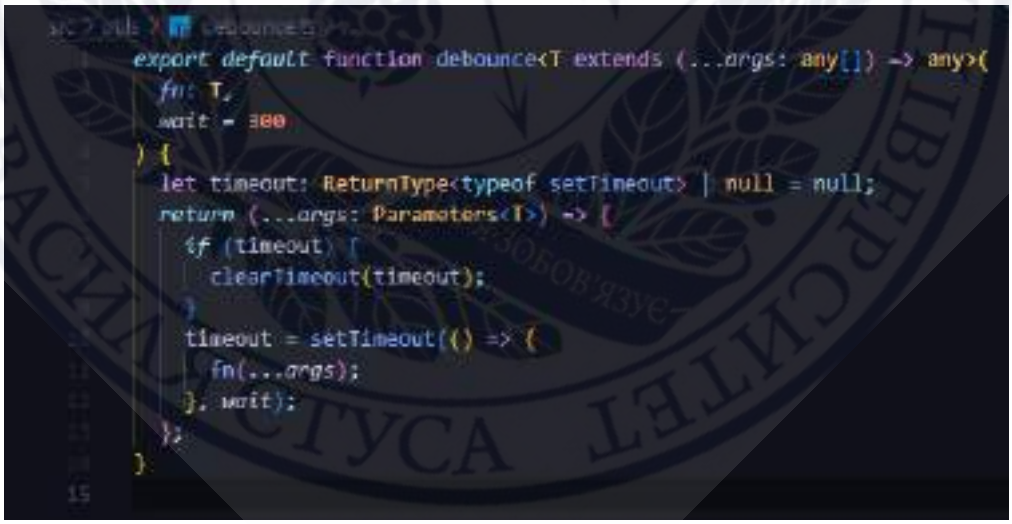


Рисунок 3.37 – Лістинг функції debounce

- formatCurrency – функція, яка відповідає за форматування ціни, отриманої із серверної частини, у зручний для користувача вигляд (див. рис. 3.38).

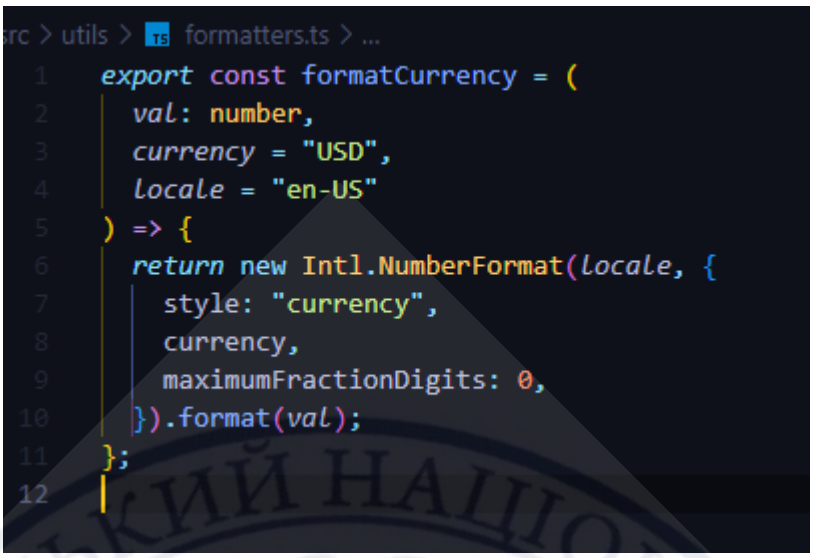


Рисунок 3.38 – Лістинг функції formatCurrency

Ці утиліти не є самостійними модулями, але відіграють важливу роль у забезпеченні стабільної та зручної роботи інтерфейсу. Їх використання дозволяє зберігати чистоту компонентів і уникати дублювання логіки.

Після опису архітектури, сховищ стану та допоміжних утиліт, наступним етапом стала реалізація візуальних компонентів, які забезпечують відображення аналітичних даних та взаємодію користувача з системою. Оскільки додаток містить велику кількість компонентів, не має сенсу детально розглядати кожен з них. Натомість доцільно зосередитися на найбільш репрезентативних елементах, які демонструють загальні принципи побудови інтерфейсу. Тому, ми розглянемо реалізацію сторінки Dashboard, яка є центральним елементом аналітичного представлення, а також компонента LineChart, що відповідає за побудову графіка динаміки продажів.

Сторінка Dashboard реалізована як комплексний компонент, який об’єднує кілька візуальних блоків: статистичні картки, графіки, таблиці та аналітичні діаграми. Вона є прикладом адаптивного макету, що змінює структуру залежно від ширини екрану, забезпечуючи зручність користування на різних пристроях.

Основні елементи сторінки (див. рис. 3.39):

- Статистичні картки (StatsCard) – відображають загальні показники: загальна сума продажів, кількість транзакцій, клієнтів та країн. Дані

отримуються з відповідних сховищ (transactionsStore, customersStore).

- Графік продажів по місяцях (LineChart) – займає основну частину екрану та дозволяє оцінити динаміку продажів протягом року.
- Діаграма продажів по країнах (PieChart) – відображає географічний розподіл транзакцій.
- Горизонтальна діаграма популярних товарів (HorizontalBarChart) – показує найпопулярніші товари за кількістю продажів.
- Таблиця останніх транзакцій (DataTable) – містить 10 останніх записів, отриманих через transactionsStore.getLastTransactions(). Форматування дати здійснюється через функцію dateTemplate.

Рисунок 3.39 – Лістинг сторінки Dashboard

Ініціалізація даних відбувається у функції Dashboard(), яка викликається при монтуванні компонента. Всі запити виконуються паралельно за

допомогою Promise.all, що дозволяє зменшити час завантаження сторінки (див. рис. 3.40).

Рисунок 3.40 – Лістинг логіки сторінки Dashboard

Компонент LineChart відповідає за побудову лінійного графіка, який демонструє зміну обсягу продажів по місяцях (див. рис. 3.41). Він реалізований на основі бібліотеки Chart.js, інтегрованої через primevue/chart [50]. Основні особливості:

- Прийом даних через props – компонент отримує масив об’єктів, кожен з яких містить номер місяця та суму продажів.
- Обчислення даних для графіка – за допомогою computed створюється масив значень, який відповідає кожному місяцю року. Якщо дані відсутні – підставляється нуль.
- Налаштування графіка – кольори, натяг кривої (tension), адаптивність (responsive), стилі осей та легенди.

- Реактивність – графік автоматично оновлюється при зміні вхідних даних.

Такий підхід дозволяє створити гнучкий і адаптивний компонент, який може бути використаний у різних частинах додатка з мінімальними змінами.

Рисунок 3.41 – Графік динаміки продажів по місяцях

Аналогічно описаним прикладам були реалізовані й інші компоненти інтерфейсу, такі як PieChart, HorizontalBarChart, StatsCard і DataTable та сторінки. Вони побудовані за тими ж принципами: прийом даних через props, реактивне оновлення стану та використання сторонніх бібліотек для візуалізації. Їх детальний опис не є обов’язковим, оскільки вони повторюють загальну архітектурну модель, продемонстровану на прикладі LineChart та Dashboard.

В результаті була послідовно реалізована клієнтська частина додатка: від ініціалізації проекту з використанням Vite і установки ключових бібліотек до розробки сервісів для роботи з API, організації сховищ стану на базі Pinia, створення допоміжних утиліт і побудови основних компонентів інтерфейсу.

3.3 Тестування та аналіз результатів роботи системи

Фінальним етапом в розробці нашого вебзастосунку, буде тестування клієнтської частини. Перш за все перевіримо верстку сторінки Dashboard, яку ми отримали (див. рис. 3.42).

Рисунок 3.42 – Підсумковий вигляд сторінки Dashboard

Для перевірки була проведена взаємодія з статистичними картками, графіками та таблицею останніх транзакцій. Усі компоненти коректно відображали дані, отримані із серверної частини, а зміна параметрів призводила до своєчасного оновлення інформації. Таким чином, можна зробити висновок, що інтерфейс функціонує стабільно, забезпечує реактивність та відповідає всім вимогам.

Наступним кроком перевіримо сторінки Products, Customers, Transactions. Сторінка Products успішно отримує та відображає данні (див. рис. 3.43). Також логіка фільтрації, сортування та пагінації працює коректно.

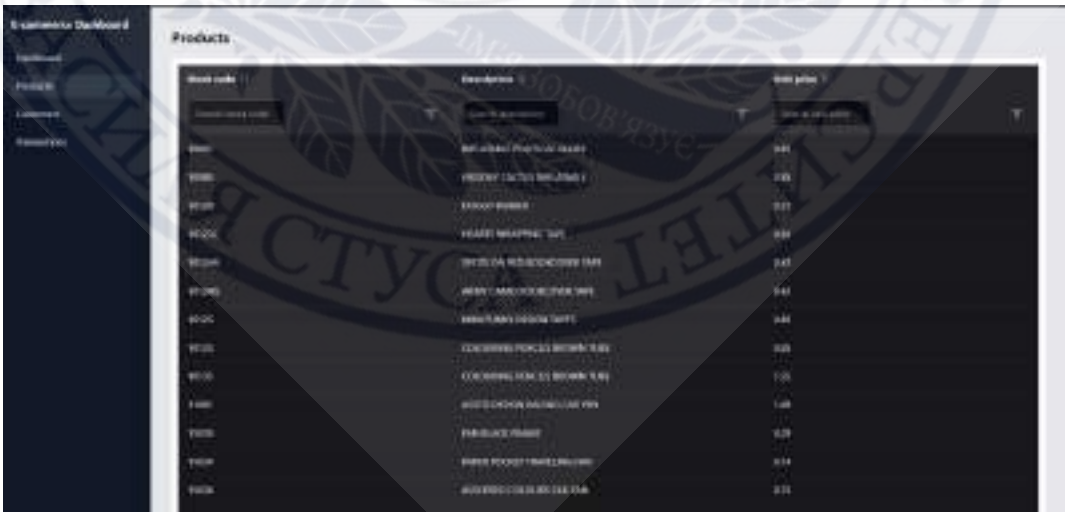


Рисунок 3.43 – Сторінка Products

Сторінка Customers також не викликає ніяких сумнівів в коректності її реалізації, все працює вірно(див. рис. 3.44).

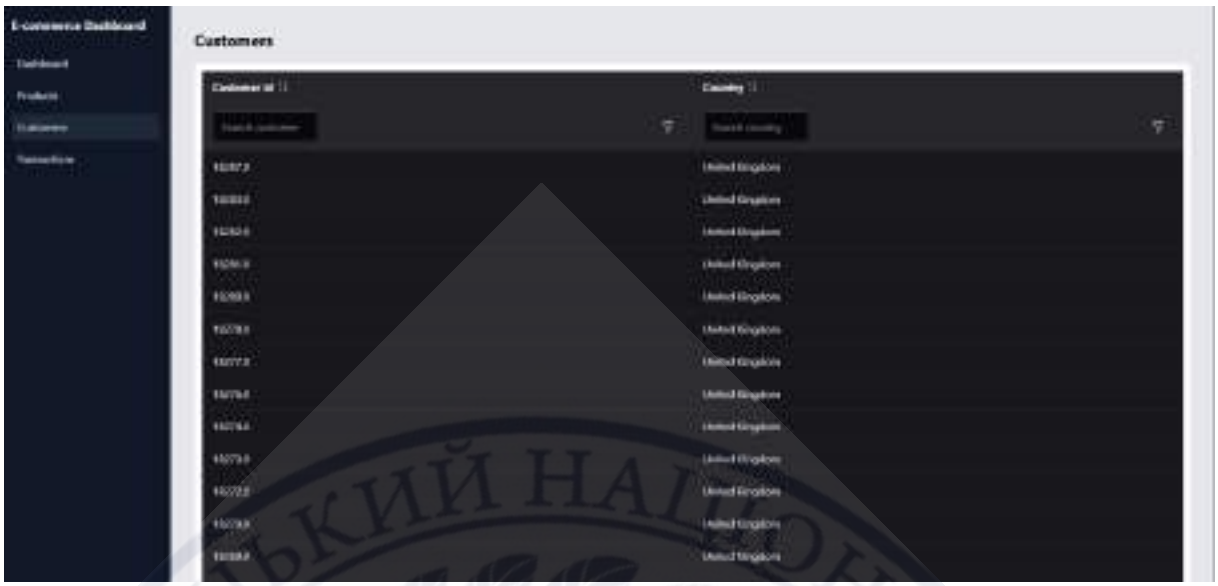


Рисунок 3.44 – Сторінка Customers

Сторінка Transactions також успішно отримує, візуалізує та за необхідності оновлює данні (див. рис. 3.45).

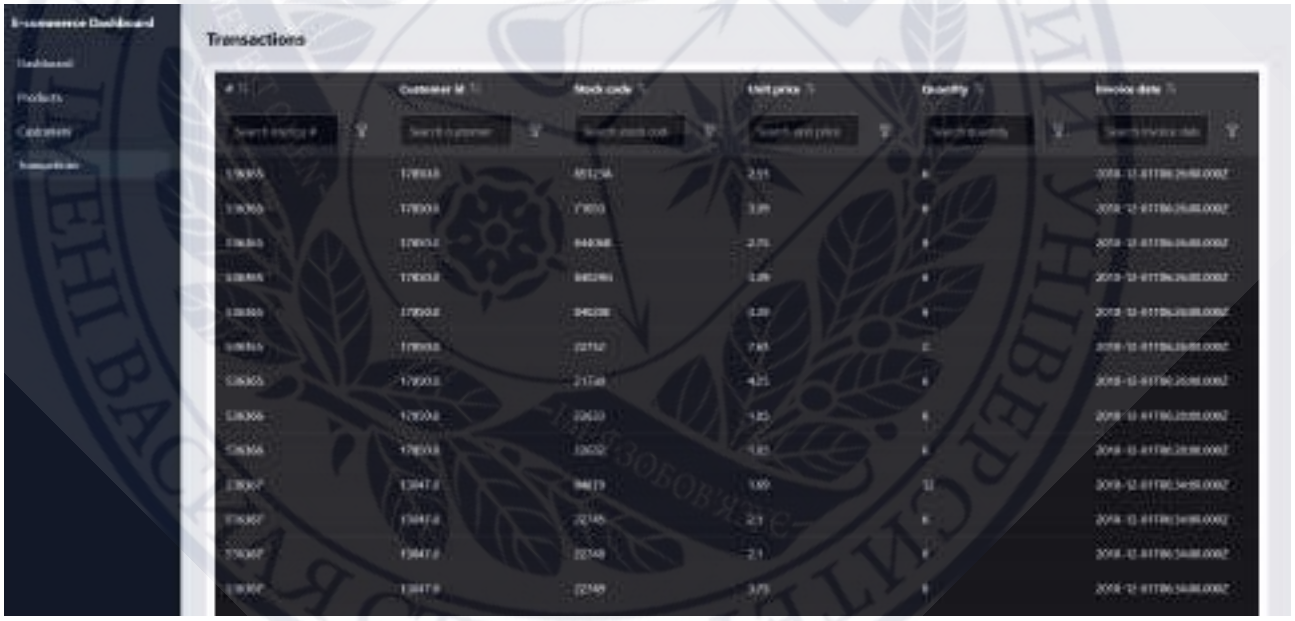


Рисунок 3.45 – Сторінка Transactions

Проведене тестування клієнтської частини вебзастосунку підтвердило коректність роботи основних сторінок та компонентів системи. Усі перевірені елементи – від статистичних карток та графіків на сторінці Dashboard до таблиць і механізмів фільтрації на сторінках Products, Customers та Transactions – функціонують стабільно, своєчасно оновлюють дані та забезпечують зручність взаємодії користувача з інтерфейсом.

Основне практичне значення розробленого вебзастосунку полягає не стільки у створенні візуального інтерфейсу у вигляді дашборда, скільки в реалізації серверної частини, яка забезпечує агрегацію, обробку та надання аналітичних даних. Саме серверна логіка формує основу системи, дозволяючи перетворювати великі масиви транзакційної інформації у структуровані показники, придатні для подальшого аналізу та прийняття управлінських рішень. Розроблені механізми агрегації даних дозволяють: формувати статистику продажів за місяцями і країнами; визначати найбільш затребувані товари і динаміку їх реалізації; розраховувати ключові показники ефективності (загальна сума продажів, кількість транзакцій, кількість клієнтів і країн); надавати дані в зручному форматі для візуалізації та інтеграції із зовнішніми системами.

Таким чином, серверна частина виступає універсальним інструментом аналітики, який може бути адаптований під різні сценарії використання. Зокрема, система може застосовуватися:

- у сфері електронної комерції для моніторингу продажів і поведінки клієнтів
- у бізнес-аналітиці для підтримки управлінських рішень
- в освітніх цілях як приклад реалізації комплексного вебзастосунку з акцентом на обробку даних
- у наукових дослідженнях, пов’язаних з аналізом транзакційних потоків і моделюванням споживчої активності.

У сукупності, розроблений продукт демонструє практичну цінність як платформа для аналітики даних і підтверджує актуальність обраного напрямку дипломного дослідження. Основний акцент на серверній частині та її логіці обробки даних робить систему не просто візуальним інструментом, а повноцінним аналітичним ядром, здатним масштабуватися та адаптуватися під потреби бізнесу та науки.

ВИСНОВКИ З РОЗДІЛУ 3

У третьому розділі представлено результати практичної реалізації вебзастосунку для аналітики продажів у сфері електронної комерції. Реалізовано серверну частину, яка забезпечує обробку великих даних, формування агрегованих показників та передачу їх через API до клієнтської частини. Створено інтерфейс користувача з дашбордом та допоміжними компонентами, що дозволяють візуалізувати статистику продажів, динаміку транзакцій і розподіл даних за країнами та товарами.

Проведене тестування підтвердило коректність роботи основних сторінок та компонентів системи, стабільність оновлення даних та зручність взаємодії користувача з інтерфейсом. Таким чином, практична реалізація довела ефективність обраних архітектурних рішень і технологічного стеку, а також відповідність створеного вебзастосунку поставленим вимогам.

ВИСНОВКИ

В ході виконання магістерської роботи була досліджено технології обробки великих даних у вебзастосунку для аналітики в сфері електронної комерції, що включає серверну частину з логікою агрегації та обробки даних, а також клієнтський інтерфейс для візуалізації результатів. Основна увага приділена серверній складовій, яка забезпечує збір, зберігання та перетворення транзакційної інформації у зручний для аналізу формат. За результатами роботи були отримані наступні висновки:

1. На підставі проведеного аналізу сучасних підходів та інструментів обробки великих даних у вебзастосунках визначено, що даний процес є складним багаторівневим і передбачає використання комплексних технологічних стеків, які забезпечують повний цикл роботи з інформацією – від її збору до візуального представлення результатів аналізу. Повний цикл роботи з даними забезпечують процеси: збір та агрегація інформації з різноманітних джерел, попередня обробка даних, зберігання даних у відповідних форматах, аналітика та візуалізація результатів, інтерактивна взаємодія з користувачем. Проведено аналіз підходів до роботи з Big Data серед яких є: використання розрахунків на основі розподілені обчислювальних моделей (MapReduce, Apache Spark та Apache Flink), хмарні платформи (Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure), компоненти архітектури та інструменти потокової обробки (Data Lakes, ETL-сервіси, серверлес-функції, контейнеризацію). Дані архітектурні підходи дозволяють створювати гнучкі та надійні аналітичні вебзастосунки, адаптовані до вимог сучасного бізнесу.

2. Аналіз методів та інструментів збору, зберігання та візуалізації даних у системах електронної комерції показав, що вони представлені як комерційними продуктами, що пропонують розширений функціонал за ліцензійною моделлю, так і безкоштовними сервісами з відкритим доступом, які дозволяють інтегрувати дані, здійснювати їх візуалізацію та проводити

аналітичні розрахунки у режимі реального часу. Основними серед них є: Google Data Studio, Tableau, Microsoft Power BI, Metabase, JavaScript-бібліотеки для візуалізації.

3. Для розробки вебзастосунку обґрунтовано вибір модульної монолітної архітектури, яка забезпечує оптимальний баланс між простотою і можливістю масштабування та дозволяє швидко приступити до розробки. Даний вид архітектурного рішення не вимагає складної інфраструктури, але при цьому зберігає логічний поділ компонентів, що важливо для супроводу і майбутньої еволюції системи.

4. В розробленому вебзастосунку для аналітики продажів реалізовано серверну частину, технологічні рішення якої забезпечують обробку великих масивів даних і формування агрегованих показників. Створений клієнтський інтерфейс включає дашборд і ряд допоміжних компонентів, які дозволяють візуалізувати статистику продажів, динаміку транзакцій і розподіл даних по країнах і товарах. Для взаємодії з клієнтською частиною розроблено API.

5. Тестування створеної системи показало, що усі компоненти коректно відображають дані, отримані із серверної частини, а своєчасне оновлення інформації відбувається внаслідок зміни відповідних параметрів. Тестування клієнтської частини вебзастосунку підтвердило коректність роботи основних сторінок та компонентів системи. Тестування елементів від статистичних карток та графіків на сторінці Dashboard до таблиць і механізмів фільтрації на сторінках Products, Customers та Transactions, показало їх здатність функціонувати стабільно та своєчасно оновлювати дані при високому рівні забезпечення зручності взаємодії користувача з інтерфейсом.

Розроблене рішення може бути використано для аналізу даних у сфері електронної комерції, а також адаптоване для інших областей, де потрібна обробка транзакційної інформації. Система демонструє можливості інтеграції серверної логіки та клієнтської візуалізації, забезпечуючи комплексний підхід до аналітики.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Ефективність інструменту Big Data в бізнесі: виклики, тренди, перспективи. 2024. URL: https://ukrainiandigital.com/strong-efektyvnist-instrumentu-big-data-v-biznesi-vyklyky-trendy-perspektyvy-strong/ (дата звернення: 03.10.2025)

2. Reinsel D., Gantz J., Rydning J. The Digitization of the World From Edge to Core. IDC White Paper, 2018. 28 с.

3. Singh A., Bawa S. Web Applications: Concepts, Technologies and Security. International Journal of Computer Applications, 2019. 15 с.

4. Моноліт чи мікросервіси: як вибрати архітектуру проєкту та не прогадати. 2025. URL: https://hub.kyivstar.ua/articles/monolit-chi-mikroservisi-yak-vibrati-arhitekturu-proyektu-ta-ne-progadati (дата звернення: 03.10.2025)

5. Що таке хмарна архітектура, простими словами. 2024. URL: https://coworkingclub.com.ua/uk/scho-take-hmarna-arhitektura/ (дата звернення: 03.10.2025)

6. MapReduce, Spark and Flink — Big 3 Processing Frameworks. 2023. URL: https://medium.com/@gjbmurugan/mapreduce-spark-and-flink-big-3-processing-frameworks-f1d7aa264113 (дата звернення: 03.10.2025)

7. AWS, Azure чи GCP: яку хмару вибрати?. 2024. URL: https://itedu.center/ua/blog/comparisons/aws-azure-chi-gcp-yaku-xmaru-vibrati/?srsltid=AfmBOoroXz31raxUoj3nwap1CPmmRMjlpjc9e-Lz9fKLBLnOYglaiNFV (дата звернення: 03.10.2025)

8. Аналіз брокерів повідомлень RabbitMQ та Apache Kafka. URL: https://nauka-online.com/publications/information-technology/2018/6/analiz-brokerov-soobshhenij-rabbitmq-i-apache-kafka/ (дата звернення: 03.10.2025)

9. Gandomi, A., & Haider, M. Beyond the hype: Big data concepts, methods, and analytics. Int. J. Information Management, 2015. 144 с.

10. MongoDB, Official Documentation. URL: https://www.mongodb.com/docs/ (дата звернення: 03.10.2025)

11. Інструменти для бізнес-аналітика: короткий огляд. 2023. URL: https://goit.global/ua/articles/instrumenty-dlia-biznes-analityka-korotkyy-ohliad/ (дата звернення: 03.10.2025)

12. Огляд кращих бібліотек для візуалізації даних у 2024 році. 2024. URL: https://itproger.com/ua/news/obzor-luchshih-bibliotek-dlya-vizualizatsii-dannih-v-2024-godu (дата звернення: 03.10.2025)

13. Google Data Studio vs. Microsoft Power BI vs. Tableau Comparison Chart. URL: https://sourceforge.net/software/compare/Google-Data-Studio-vs-Power-BI-vs-Tableau (дата звернення: 03.10.2025)

14. Google Data Studio. URL: https://datastudio.google.com/ (дата звернення: 03.10.2025)

15. Tableau. URL: https://www.tableau.com/ (дата звернення: 03.10.2025)

16. Microsoft Power BI. URL: https://powerbi.microsoft.com/ (дата звернення: 03.10.2025)

17. Metabase. URL: https://www.metabase.com/ (дата звернення: 03.10.2025)

18. Comparing the most popular open-source charting libraries. 2024. URL: https://www.metabase.com/blog/best-open-source-chart-library (дата звернення: 03.10.2025)

19. Monolith Architecture vs. Microservices. 2025. URL: https://www.ramotion.com/blog/monolithic-architecture-vs-microservice-architecture (дата звернення: 06.10.2025)

20. Pros and cons of monolithic vs. microservices architecture 2020. URL: https://www.techtarget.com/searchapparchitecture/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture (дата звернення: 06.10.2025)

21. Monolith vs Microservices: Understanding the Pros and Cons of each Approach. 2023. URL: https://aquare.la/en/monolith-vs-microservices-understanding-the-pros-and-cons-of-each-approach (дата звернення: 06.10.2025)

22. Node.js vs Python: Real Benchmarks, Performance Insights, and Scalability Analysis. 2025. URL: https://dev.to/m-a-h-b-u-b/nodejs-vs-python-real-

benchmarks-performance-insights-and-scalability-analysis-4dm5 (дата звернення: 06.10.2025)

23. Node.js vs Python: Which Backend Is Better in 2025? 2025. URL: https://www.sevensquaretech.com/nodejs-vs-python-comparison (дата звернення: 06.10.2025)

24. D3 or Chart.js for Data Visualisation? 2021. URL: https://www.createwithdata.com/d3js-or-chartjs (дата звернення: 06.10.2025)

25. Chart.js vs d3 vs highcharts vs echarts 2025. URL: https://npm-compare.com/chart.js,d3,echarts,highcharts (дата звернення: 06.10.2025)

26. Офіційна документація Apache Superset. 2025. URL: https://superset.apache.org/docs/6.0.0/intro (дата звернення: 06.10.2025)

27. Apache Superset: Features, Benefits, and Use Cases. 2024. URL: https://www.coraltechteam.com/apache-superset-features-benefits-and-use-cases (дата звернення: 06.10.2025)

28. Top JavaScript Frameworks to Use in 2025. 2025. URL: https://dev.to/eva_clari_289d85ecc68da48/top-javascript-frameworks-to-use-in-2025-2g9i (дата звернення: 06.10.2025)

29. Top JavaScript Frameworks Comparison for Web Development in 2025. 2025. URL: https://www.brilworks.com/blog/javascript-web-frameworks-comparison (дата звернення: 06.10.2025)

30. Data Modeling for Web Analytics. 2022. URL: https://snowplow.io/blog/data-modeling-for-web-analytics (дата звернення: 11.10.2025)

31. State Management with Pinia vs Vuex. 2025. URL: https://dev.to/robin-ivi/state-management-with-pinia-vs-vuex-4mh (дата звернення: 11.10.2025)

32. Що таке Axios?. 2025. URL: https://axios-http.com/uk/docs/intro (дата звернення: 11.10.2025)

33. When to Use MongoDB? 2025. URL: https://www.geeksforgeeks.org/mongodb/when-to-use-mongodb (дата звернення: 11.10.2025)

34. MongoDB: що це за СУБД, переваги та недоліки? 2025. URL: https://goit.global/ua/articles/mongodb-shcho-tse-take-ta-yak-prazuie/ (дата звернення: 11.10.2025)

35. Difference between SQL and NoSQL. 2025. URL: https://www.geeksforgeeks.org/sql/difference-between-sql-and-nosql (дата звернення: 11.10.2025)

36. MongoDB Schema Design Best Practices and Techniques. 2025. URL: https://www.geeksforgeeks.org/mongodb/mongodb-schema-design-best-practices-and-techniques (дата звернення: 11.10.2025)

37. Aggregation Operations. 2025. URL: https://www.mongodb.com/docs/manual/aggregation (дата звернення: 11.11.2025)

38. AWS Vs Google Cloud Platform Vs Azure. 2025. URL: https://www.geeksforgeeks.org/devops/aws-vs-google-cloud-platform-vs-azure (дата звернення: 11.11.2025)

39. Overview of Insomnia API and The Best Insomnia Alternative. 2025. URL: https://apidog.com/blog/insomnia-api (дата звернення: 11.11.2025)

40. Node First Application. 2025. URL: https://www.geeksforgeeks.org/node-js/node-js-first-application (дата звернення: 22.11.2025)

41. .env Files and the Art of Not Committing Secrets. 2025. URL: https://blog.openreplay.com/env-files-art-not-committing-secrets (дата звернення: 22.11.2025)

42. How is id Generated in MongoDB. 2025. URL: https://www.geeksforgeeks.org/mongodb/how-is-id-generated-in-mongodb (дата звернення: 22.11.2025)

43. Mongoose Tutorial. 2025. URL: https://www.geeksforgeeks.org/node-js/mongoose-tutorial (дата звернення: 22.11.2025)

44. REST API Architectural Constraints. 2025. URL: https://www.geeksforgeeks.org/javascript/rest-api-architectural-constraints (дата звернення: 22.11.2025)

45. HTTP request methods explained. 2025. URL:

https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/HTTP-methods (дата звернення: 22.11.2025)

46. Using $group aggregation stage in MongoDB (with examples). 2025. URL: https://www.slingacademy.com/article/using-group-aggregation-stage-in-mongodb-with-examples (дата звернення: 22.11.2025)

47. Aggregation in MongoDB. 2025. URL: https://www.tutorialsteacher.com/mongodb/aggregation (дата звернення: 22.11.2025)

48. Ти – фронтенд-розробник? Дізнайся, як інструмент Vite спростить тобі роботу. 2025. URL: https://dev.ua/blogs/posts/tarabanov-1736410707 (дата звернення: 22.11.2025)

49. Pinia та з чим його їдять. 2022. URL: https://medium.com/@winato/pinia-та-з-чим-його-їдять-f8dde3281319 (дата звернення: 22.11.2025)

50. PrimeVue Chart component. 2025. URL: https://primevue.org/chart (дата звернення: 22.11.2025)

ДОДАТКИ

ДОДАТОК А

Лістинг основних функцій серверної частини

Код файлу index.ts

```
import express from "express";
import mongoose from "mongoose";
import cors from "cors";
import dotenv from "dotenv";
import productsRouter from "./routes/products";
import customersRouter from "./routes/customers";
import salesRouter from "./routes/sales";
dotenv.config();
const app = express();
app.use(cors());
app.use(express.json());
app.use("/api/products", productsRouter);
app.use("/api/customers", customersRouter);
app.use("/api/sales", salesRouter);
const PORT = process.env.PORT || 5000;
mongoose
  .connect(process.env.MONGO_URL!)
  .then(() => {
    console.log("MongoDB connected");
    app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
  })
  .catch((err) => console.error(err));
```

Код файлу sales.ts

```
import { Router } from "express";
import Sale from "../models/Sale";
import { getTotalSales } from "../utils/getTotalSales";
import { getMonthlySales } from "../utils/getMonthlySales";
import { getSalesByCountry } from "../utils/getSalesByCountry";
import { getLastTransactions } from "../utils/getLastTransactions";
import { getSales } from "../utils/getSales";
const router = Router();
// GET — всі продажі з пагінацією
router.get("/", async (req, res) => {
  try {
    const page = parseInt(req.query.page as string) || 1;
```

```
    const limit = parseInt(req.query.limit as string) || 100;
    const sortField = req.query.sortField as string;
    const sortOrder = parseInt(req.query.sortOrder as string);
    const filtersRaw = req.query.filters as string;
    const result = await getSales({
      page,
      limit,
      sortField,
      sortOrder,
      filtersRaw,
    });
    res.json(result);
  } catch (err) {
    console.error("Error fetching sales:", err);
    res.status(500).json({ error: (err as Error).message });
  }
});
// GET - сума всіх продажів
router.get("/totalSales", async (req, res) => {
  try {
    const totalSales = await getTotalSales();
    res.json({ totalSales });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Failed to calculate total sales" });
  }
});
// GET - кількість продажів по місячно
router.get("/monthlySales", async (req, res) => {
  try {
    const monthlySales = await getMonthlySales();
    res.json({ monthlySales });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Failed to calculate monthly sales" });
  }
});
// GET - продажі по країнам
router.get("/salesByCountry", async (req, res) => {
  try {
    const salesByCountry = await getSalesByCountry();
    res.json({ salesByCountry });
```

```
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Failed to calculate sales by country" });
  }
});
// GET - 10 останніх транзакцій
router.get("/lastTransactions", async (req, res) => {
  try {
    const lastTransactions = await getLastTransactions();
    res.json({ lastTransactions });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Failed to calculate last transactions" });
  }
});
export default router;
```

Код файлу product.ts

```
import { Router } from "express";
import Product from "../models/Product";
import { getTopProducts } from "../utils/getTopProducts";
const router = Router();
// GET — всі продукти
router.get("/", async (req, res) => {
  try {
    const page = parseInt(req.query.page as string) || 1;
    const limit = parseInt(req.query.limit as string) || 100;
    const sortField = req.query.sortField as string;
    const sortOrder = parseInt(req.query.sortOrder as string);
    const filtersRaw = req.query.filters as string;

    // Парсимо фільтри (якщо є)
    const filters = filtersRaw ? JSON.parse(filtersRaw) : {};
    // Побудуємо query-об'єкт
    const query: Record<string, any> = {};
    const andClauses: any[] = [];
    Object.entries(filters).forEach(([key, filter]: any) => {
      if (
        filter == null ||
        filter.value === undefined ||
        filter.value === null ||
        filter.value === ""
```

```
  )
    return;
  // Спеціальна обробка для числового поля unitPrice
  if (key === "unitPrice") {
    if (filter.matchMode === "equals") {
      const n = Number(filter.value);
      query[key] = Number.isNaN(n) ? filter.value : n;
    } else if (filter.matchMode === "contains") {
      // Для contains будемо порівнювати рядкове представлення числа
      andClauses.push({
        $expr: {
          $regexMatch: {
            input: { $toString: `$${key}` },
            regex: String(filter.value),
            options: "i",
          },
        },
      });
    }
    return;
  }
  // Інші поля (stockCode, description) — рядкові
  switch (filter.matchMode) {
    case "contains":
      query[key] = { $regex: String(filter.value), $options: "i" };
      break;
    case "equals":
      query[key] = filter.value;
      break;
    default:
      break;
  }
});
if (andClauses.length) {
  query.$and = (query.$and || []).concat(andClauses);
}
// Сортування
const sort: Record<string, 1 | -1> = {};
if (sortField) {
  sort[sortField] = sortOrder === 1 ? 1 : -1;
} else {
  sort["stockCode"] = 1; // стандартне сортування
```

```
    }
    const total = await Product.countDocuments(query);
    const products = await Product.find(query)
      .sort(sort)
      .skip((page - 1) * limit)
      .limit(limit);
    res.json({ total, page, limit, products });
  } catch (err) {
    res.status(500).json({ error: (err as Error).message });
  }
});
// GET - топ 5 продуктів за продажами
router.get("/topProducts", async (req, res) => {
  try {
    const topProducts = await getTopProducts();
    res.json(topProducts);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Failed to calculate top products" });
  }
});
export default router;
```

Код файлу Sale.ts

```
import { Schema, model, Document } from "mongoose";
export interface ISale extends Document {
  invoiceNo: string;
  customerId: string; // посилання на Customers
  stockCode: string; // посилання на Products
  quantity: number;
  invoiceDate: Date;
  unitPrice: number;
}
const saleSchema = new Schema<ISale>({
  invoiceNo: { type: String, required: true },
  customerId: { type: String, required: true },
  stockCode: { type: String, required: true },
  quantity: { type: Number, required: true },
  invoiceDate: { type: Date, required: true },
  unitPrice: { type: Number, required: true },
});
```

```
export default model<ISale>("Sale", saleSchema);
```

Код файлу Product.ts

```
import { Schema, model, Document } from "mongoose";
export interface IProduct extends Document {
  stockCode: string;
  description: string;
  unitPrice: number;
}
const productSchema = new Schema<IProduct>({
  stockCode: { type: String, required: true, unique: true },
  description: { type: String, required: true },
  unitPrice: { type: Number, required: true },
});

export default model<IProduct>("Product", productSchema);
```

Код файлу getSales.ts

```
import Sale from "../models/Sale";
interface Filter {
  value?: any;
  matchMode?: string;
}

interface Filters {
  [key: string]: Filter;
}
export const getSales = async (params: {
  page?: number;
  limit?: number;
  sortField?: string;
  sortOrder?: number;
  filtersRaw?: string;
}) => {
  const { page = 1, limit = 100, sortField, sortOrder, filtersRaw } = params;
  const filters: Filters = filtersRaw ? JSON.parse(filtersRaw) : {};
  const query: Record<string, any> = {};
  const isValidDate = (v: any) => {
    const d = new Date(v);
    return !isNaN(d.getTime());
  };
  Object.entries(filters).forEach(([key, filter]) => {
    if (
```

```
  !filter ||
  filter.value === undefined ||
  filter.value === null ||
  filter.value === ""
)
  return;
//Обробка фільтра дати
if (key === "invoiceDate") {
  const val = filter.value;
  if (
    Array.isArray(val) &&
    val.length === 2 &&
    isValidDate(val[0]) &&
    isValidDate(val[1])
  ) {
    const start = new Date(val[0]);
    const end = new Date(val[1]);
    start.setHours(0, 0, 0, 0);
    end.setHours(23, 59, 59, 999);
    query[key] = { $gte: start, $lte: end };
  } else if (typeof val === "string" && isValidDate(val)) {
    const d = new Date(val);
    const start = new Date(d);
    start.setHours(0, 0, 0, 0);
    const end = new Date(d);
    end.setHours(23, 59, 59, 999);
    query[key] = { $gte: start, $lte: end };
  } else if (
    typeof val === "object" &&
    isValidDate(val.start) &&
    isValidDate(val.end)
  ) {
    const start = new Date(val.start);
    const end = new Date(val.end);
    start.setHours(0, 0, 0, 0);
    end.setHours(23, 59, 59, 999);
    query[key] = { $gte: start, $lte: end };
  }
  return;
}
//Текстові або числові фільтри
switch (filter.matchMode) {
```

```
      case "contains":
        query[key] = { $regex: filter.value, $options: "i" };
        break;
      case "equals":
        query[key] = filter.value;
        break;
    }
  });
  //Сортування
  const sort: Record<string, 1 | -1> = {};
  if (sortField) {
    sort[sortField] = sortOrder === 1 ? 1 : -1;
  } else {
    sort["date"] = -1;
  }
  //Отримуємо дані
  const total = await Sale.countDocuments(query);
  const sales = await Sale.find(query)
    .sort(sort)
    .skip((page - 1) * limit)
    .limit(limit);
  return { total, page, limit, sales };
};
```

ДОДАТОК Б

Лістинг основних функцій клієнтської частини

Код файлу main.ts

```
import { createApp } from "vue";
import "./style.css";
import App from "./App.vue";
import router from "./router";
import { createPinia } from "pinia";
import PrimeVue from "primevue/config";
import Lara from "@primeuix/themes/lara";
import ApiService from "./core/services/ApiService";
const app = createApp(App);
app.use(createPinia());
app.use(router);
app.use(PrimeVue, {
  theme: {
    preset: Lara,
  },
});
ApiService.init(app);
app.mount("#app");
```

Код файлу App.vue

```
<template>
  <div class="flex h-screen font-[Trebuchet MS]">
    <!-- Sidebar -->
    <aside class="w-64 bg-gray-900 text-white p-4">
      <h2 class="text-lg font-bold mb-6">E-commerce Dashboard</h2>
      <nav class="space-y-2">
        <RouterLink
          to="/"
          class="block hover:bg-gray-700 p-2 rounded"
          active-class="bg-gray-800"
          >Dashboard</RouterLink
        >
        <RouterLink
          to="/products"
          class="block hover:bg-gray-700 p-2 rounded"
          active-class="bg-gray-800"
          >Products</RouterLink
        >
        <RouterLink
          to="/customers"
          class="block hover:bg-gray-700 p-2 rounded"
```

```
          active-class="bg-gray-800"
          >Customers</RouterLink
        >
        <RouterLink
          to="/transactions"
          class="block hover:bg-gray-700 p-2 rounded"
          active-class="bg-gray-800"
          >Transactions</RouterLink
        >
      </nav>
    </aside>
    <!-- Main content -->
    <main class="flex-1 bg-gray-200 overflow-y-auto">
      <router-view />
    </main>
  </div>
</template>
<script setup lang="ts"></script>
<style scoped></style>
```

Код файлу Customers.ts

```
import { defineStore } from "pinia";
import ApiService from "../core/services/ApiService";
import { ref } from "vue";
import type { AxiosRequestConfig } from "axios";
interface Customer {
  customerId: string;
  country: string;
}
export const useCustomerStore = defineStore("customers", () => {
  // State
  const customers = ref<Customer[]>([]);
  const customersCount = ref();
  const countriesCount = ref();
  const isLoading = ref(true);
  // Actions
  function getCustomers(params: any) {
    const config: AxiosRequestConfig = {
      params: params,
    };
    return ApiService.query("/api/customers", config)
      .then(({ data }) => {
        customers.value = data.customers;
```

```
      customersCount.value = data.total;
    })
    .catch((err) => {
      console.log(err);
    });
 }
 function getCustomersCount(params: any) {
  const config: AxiosRequestConfig = {
   params: params,
  };
  return ApiService.query("/api/customers/customersCount", config)
   .then(({ data }) => {
    customersCount.value = data;
   })
   .catch((err) => {
    console.log(err);
   });
 }
 function getCountriesCount(params: any) {
  const config: AxiosRequestConfig = {
   params: params,
  };
  return ApiService.query("/api/customers/countriesCount", config)
   .then(({ data }) => {
    countriesCount.value = data;
   })
   .catch((err) => {
    console.log(err);
   });
 }
 return {
  customers,
  isLoading,
  customersCount,
  countriesCount,
  getCustomers,
  getCustomersCount,
  getCountriesCount,
 };
});
```

Код файлу ApiService.ts

```
import type { App } from "vue";
```

```
import type { AxiosResponse } from "axios";
import axios from "axios";
import VueAxios from "vue-axios";
/**
 * @description service to call HTTP request via Axios
 */
class ApiService {
  /**
   * @description property to share vue instance
   */
  public static vueInstance: App;
  /**
   * @description initialize vue axios
   * @param vue vue app instance
   */
  public static init(app: App<Element>) {
    ApiService.vueInstance = app;
    ApiService.vueInstance.use(VueAxios, axios);
    ApiService.vueInstance.axios.defaults.baseURL =
      import.meta.env.VITE_SERVER_URL;
  }
  /**
   * @description send the HTTP GET request
   * @param resource
   * @param params
   * @returns Promise<AxiosResponse>
   */
  public static query(resource: string, params: any): Promise<AxiosResponse> {
    return ApiService.vueInstance.axios.get(
      `${import.meta.env.VITE_SERVER_URL}${resource}`,
      params
    );
  }
}
export default ApiService;
```

Код файлу Dashboard.vue

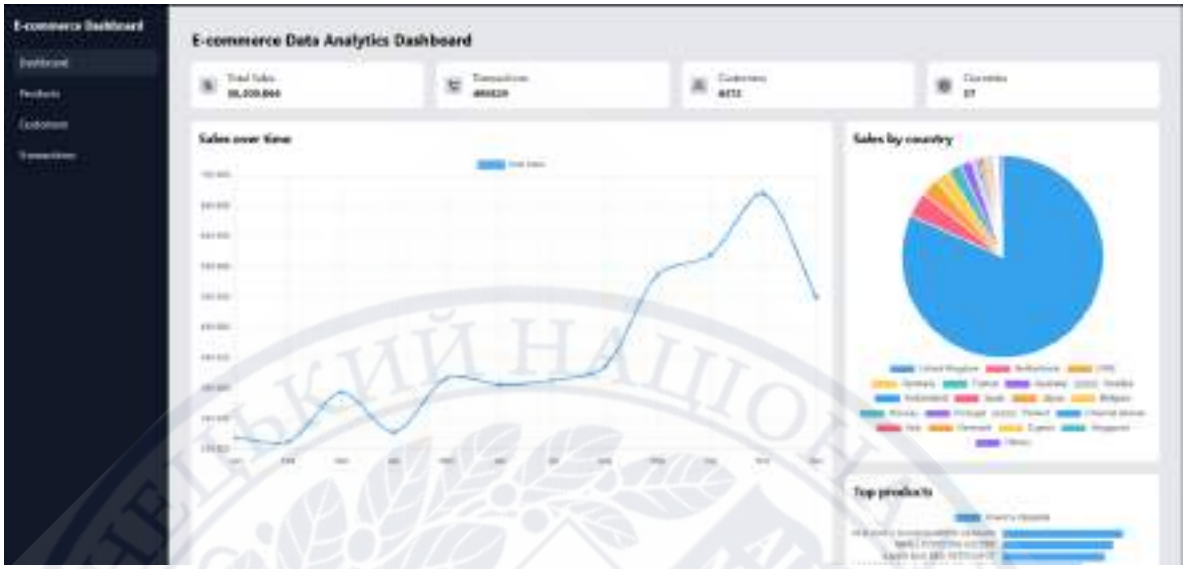
```
<template>
  <div class="p-9">
    <h1 class="text-black text-2xl font-bold mb-5">
      E-commerce Data Analytics Dashboard
    </h1>
```

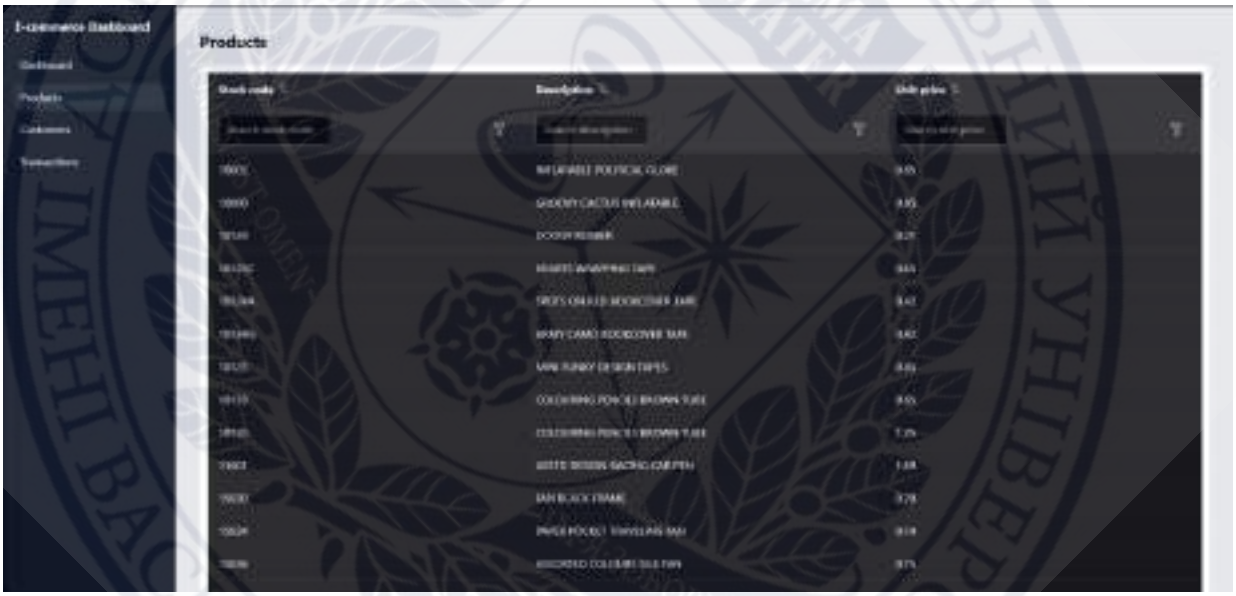
```
<div class="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-4 gap-5 mb-5">
  <StatsCard
    title="Total Sales"
    :data="transactionsStore.totalSales"
    :formatter="(val) => formatCurrency(val)"
  >
    <span
      class="pi pi-dollar text-black bg-gray-300 p-1.5 rounded-md"
    ></span>
  </StatsCard>
  <StatsCard
    title="Transactions"
    :data="transactionsStore.transactionsCount"
  >
    <span
      class="pi pi-shopping-cart text-black bg-gray-300 p-1.5 rounded-md"
    ></span>
  </StatsCard>
  <StatsCard title="Customers" :data="customersStore.customersCount">
    <span class="pi pi-user text-black bg-gray-300 p-1.5 rounded-md"></span>
  </StatsCard>
  <StatsCard title="Countries" :data="customersStore.countriesCount">
    <span
      class="pi pi-globe text-black bg-gray-300 p-1.5 rounded-md"
    ></span>
  </StatsCard>
</div>
<div class="grid grid-cols-1 lg:grid-cols-3 gap-5 mb-5">
  <div class="bg-white rounded-md p-3 lg:col-span-2">
    <h2 class="text-black text-xl font-bold mb-3">Sales over time</h2>
    <LineChart :data="transactionsStore.monthlySales" />
  </div>
  <div class="space-y-5">
    <div class="bg-white rounded-md p-3">
      <h2 class="text-black text-xl font-bold mb-3">Sales by country</h2>
      <PieChart :data="transactionsStore.salesByCountry" />
    </div>
    <div class="bg-white rounded-md p-3">
      <h2 class="text-black text-xl font-bold mb-3">Top products</h2>
      <HorizontalBarChart :data="productsStore.topProducts" />
    </div>
  </div>
```

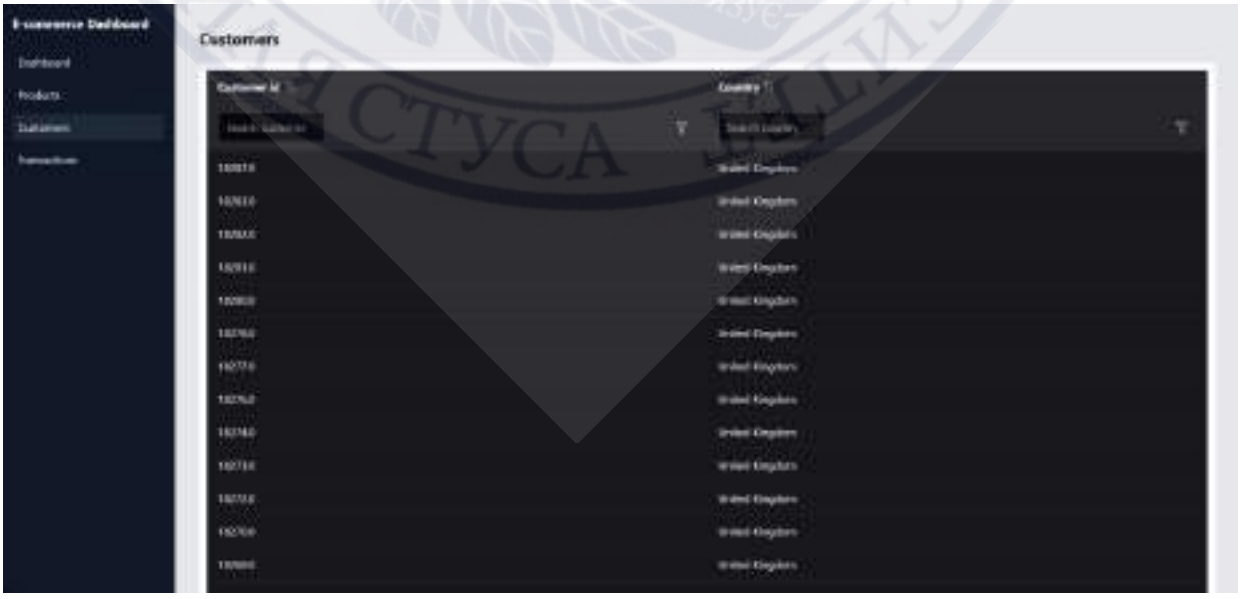
```
    </div>
    <div class="bg-white rounded-md p-3">
     <h2 class="text-black text-xl font-bold mb-3">Latest transactions</h2>
     <DataTable :value="transactionsStore.lastTransactions">
      <Column
       v-for="col of tableColumns"
       :key="col.field"
       :field="col.field"
       :header="col.header"
      >
       <template v-if="col.field === 'invoiceDate'" #body="slotProps">
        {{ dateTemplate(slotProps.data.invoiceDate) }}
       </template>
      </Column>
     </DataTable>
    </div>
  </div>
</template>
<script lang="ts">
import { defineComponent } from "vue";
import { Dashboard } from "../../logic/views/Dashboard";
import StatsCard from "../cards/StatsCard.vue";
import LineChart from "../chart/LineChart.vue";
import HorizontalBarChart from "../chart/HorizontalBarChart.vue";
import PieChart from "../chart/PieChart.vue";
import DataTable from "primevue/datatable";
import Column from "primevue/column";
export default defineComponent({
  name: "Dashboard",
  components: {
    StatsCard,
    LineChart,
    HorizontalBarChart,
    PieChart,
    DataTable,
    Column,
  },
  setup() {
    return Dashboard();
  },
});</script>
```

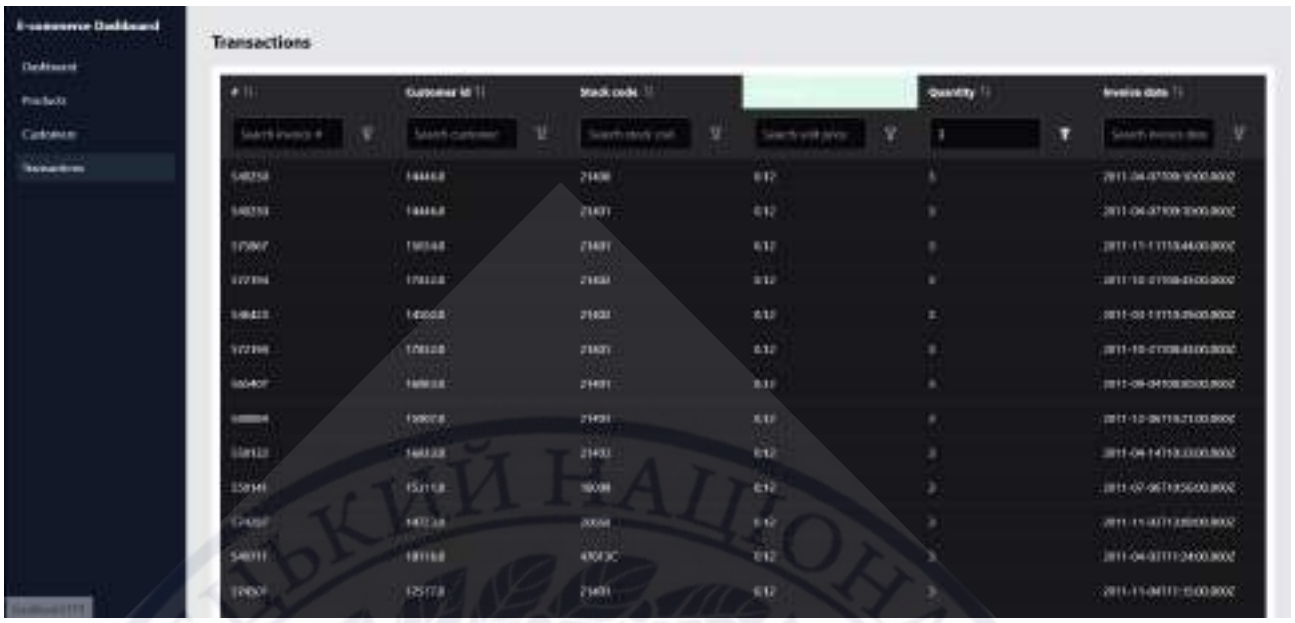
ДОДАТОК В

Приклади роботи вебзастосунку








E-commerce Dashboard
Dashboard
Products
Customers
Transactions
Transactions
Customer ID
Stock code
Quantity
Invoice date

ДОДАТОК Г

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)