

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

"На правах рукопису"
УДК

«До захисту допущено»
Завідувач кафедри

“ ” Наталія АУШЕВА
2025 р.

Магістерська дисертація

на здобуття ступеня другого (магістерського) рівня вищої освіти
за освітньою програмою “Цифрові технології в енергетиці”
зі спеціальності 122 “Комп’ютерні науки”

на тему Технології інтеграції застосунків ASP.NET Core з сучасними фронтенд-інструментами

Виконав: студент 2 курсу, групи ТР-41мп

Падалко Денис Романович

(прізвище, ім'я, по батькові)

(підпис)

Науковий керівник к.ф.-м.н., доцент Юрій ТАРНАВСЬКИЙ

(науковий ступінь, вчене звання, ім'я ПРІЗВИЩЕ)

(підпис)

Рецензент завідувачка каф. теплової та альтернативної енергетики, д.т.н., професорка Черноусенко Ольга Юріївна

(посада, науковий ступінь, вчене звання, ім'я ПРІЗВИЩЕ)

(підпис)

Н.контроль ст. вик., Ольга БЕСПАЛА

(посада, ім'я ПРІЗВИЩЕ)

(підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ - 2025

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”**

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – другий (магістерський)

спеціальність 122 “Комп’ютерні науки”

Освітньо-наукова програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ

Завідувач кафедри ЦТЕ

Наталія АУШЕВА

«__» _____ 2025 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту**

Падалко Денису Романовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Технології інтеграції застосунків ASP.NET Core з сучасними фронтенд-інструментами

керівник роботи Тарнавський Юрій Адамович, к.ф.-м.н., доцент

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 03 листопада 2025 р. № 4779-с

2. Термін подання студентом дисертації 06.12.2025

3. Об’єкт дослідження : процес інтеграції застосунків ASP.NET Core з сучасними фронтенд-інструментами

4. Вихідні дані: платформа ASP.NET Core, мова програмування C#, система керування базами даних PostgreSQL, фреймворки React, Vue, Angular, ORM-бібліотека Entity Framework Core, середовище розробки Visual Studio

5. Перелік завдань: 1) провести аналіз технологій інтеграції фронтенду та бекенду; 2) обґрунтувати вибір інструментів і технологій для реалізації застосунку; 3) спроектувати архітектуру веб-застосунку та REST API; 4) розробити серверну частину інтернет-магазину та клієнтські інтерфейси; 5) здійснити інтеграцію фронтендів з бекендом і провести тестування; 6) підготувати концепцію стартап-проєкту на основі створеної системи.
6. Орієнтований перелік ілюстративного матеріалу : 14 слайдів з наступними заголовками: титульний слайд; актуальність теми; мета та завдання роботи; основні методи інтеграції веб-застосунків; порівняльна характеристика методів інтеграції; програмні засоби та технології проєкту; діаграма класів моделі даних; три слайди з демонстрацією роботи фронтендів; загальна архітектура системи; висновки презентації; слайд з відомостями про автора та апробацію результатів.
7. Орієнтований перелік публікацій : Опубліковані тези на XXII Міжнародній науково-практичній конференції молодих вчених і студентів, 22–25 квітня 2025 року
8. Дата видачі завдання «16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів підготовки магістерської дисертації	Термін виконання	Примітка
1	Аналіз методів дослідження за літературними джерелами.	16.09.2024– 15.12.2024	Виконано
2	Вибір та налаштування засобів реалізації.	16.12.2024 – 31.01.2025	Виконано
3	Програмна реалізація серверної частини засобами ASP.NET Core.	01.02.2025 – 31.05.2025	Виконано
4	Інтеграція сучасних фронтенд-інструментів з серверною частиною.	01.06.2025 – 30.09.2025	Виконано
5	Аналіз результатів тестування та ефективності інтеграції.	01.10.2025 – 23.10.2025	Виконано
6	Захист створеного програмного забезпечення.	24.10.2025	Виконано
7	Оформлення магістерської дисертації.	24.10.2025 – 23.11.2025	Виконано
8	Передзахист.	27.11.2025	Виконано
9	Подання готової роботи на кафедру.	28.11.2025 – 04.12.2025	Виконано
10	Захист.	16.12.2025-29.12.2025	Виконано

Студент

(підпис)

Денис ПАДАЛКО

(ім'я, ПРИЗВИЩЕ)

Керівник роботи

(підпис)

Юрій ТАРНАВСЬКИЙ

(ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

Актуальність теми. Сучасні веб-застосунки дедалі частіше будуються як розподілені системи, у яких серверна частина та інтерфейс користувача розробляються за допомогою різних технологій. Для успішного функціонування таких систем необхідна коректна інтеграція між клієнтом і сервером, що забезпечує узгоджений обмін даними, масштабованість і гнучкість розвитку продукту. З огляду на широке використання ASP.NET Core у корпоративних рішеннях і популярність фреймворків React, Angular і Vue.js на фронтенді, дослідження особливостей їх інтеграції є актуальним та має практичну цінність.

У роботі розглянуто й реалізовано прототип веб-застосунку IntegraShop, який демонструє архітектурні підходи та технічні рішення інтеграції сучасних SPA-фреймворків із серверною платформою ASP.NET Core на прикладі спрощеної моделі інтернет-магазину.

Метою роботи є дослідження методів інтеграції фронтенд-застосунків із бекенд-частиною на базі ASP.NET Core та розроблення прототипу системи електронної комерції, що демонструє взаємодію різними засобами інтеграції.

Завдання дослідження:

- провести аналіз методів та технологій інтеграції клієнтських веб-застосунків із серверними API;
- розробити архітектуру серверної частини IntegraShop та реалізувати REST API засобами ASP.NET Core;
- створити три фронтенд-клієнти (React, Angular, Vue) та забезпечити їх повноцінну інтеграцію з бекендом;
- провести тестування інтеграції, оцінити коректність обміну даними та працездатність прототипу;
- обґрунтувати можливість подальшого використання розробленої системи в контексті стартап-проєкту.

Об’єкт дослідження – процес інтеграції застосунків ASP.NET Core з сучасними фронтенд-інструментами.

Предмет дослідження – методи, технології та архітектурні підходи до побудови інтеграційних взаємодій між серверною частиною ASP.NET Core та клієнтськими SPA-фреймворками.

Практична цінність отриманих результатів полягає у створенні робочого прототипу інтегрованої системи електронної комерції, який може використовуватися як основа для подальшої розробки повноцінного веб-продукту. Реалізований підхід забезпечує модульність, масштабованість та можливість гнучкого підключення різних клієнтських додатків. Окрім того, результати роботи можуть бути використані при проєктуванні реальних комерційних систем, впровадженні API-орієнтованих підходів та навчанні веб-розробці.

Методи дослідження. У роботі застосовано аналітичні методи для порівняння технологій інтеграції, методи проєктування програмних систем, моделювання архітектури, а також практичні методи програмної реалізації та експериментальної перевірки працездатності створених компонентів веб-застосунку.

Апробація результатів дисертації. Основні положення роботи були представлені у вигляді опублікованих тез на XXII Міжнародній науково-практичній конференції молодих вчених і студентів (22–25 квітня 2025 року).

Структура та обсяг роботи. Дисертація складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. Загальний обсяг становить 103 сторінок, включаючи 85 сторінок основного тексту, 1 додатку, 6 рисунків, 3 таблиці та список використаних джерел із 13 найменувань.

Ключові слова: ASP.NET Core, REST API, інтеграція, React, Angular, Vue.js, веб-розробка, електронна комерція, клієнт-серверна архітектура.

ABSTRACT

Relevance of the topic. Modern web applications are increasingly built as distributed systems in which the server side and the user interface are created using different technological stacks. Successful operation of such systems requires correct integration between the client and the server, ensuring consistent data exchange, scalability, and flexibility of product development. Considering the wide adoption of ASP.NET Core in enterprise solutions and the popularity of React, Angular, and Vue.js on the frontend, studying the specifics of their integration is both relevant and practically significant.

The thesis presents the analysis and implementation of a prototype web application IntegraShop, which demonstrates architectural approaches and technical solutions for integrating modern SPA frameworks with an ASP.NET Core backend using a simplified e-commerce model.

The aim of the research is to investigate the methods of integrating frontend applications with a backend built on ASP.NET Core and to develop a prototype e-commerce system that demonstrates interaction through various integration mechanisms.

Research tasks:

- to analyze existing methods and technologies of integration between client-side web applications and server-side APIs;
- to design the architecture of the server side of IntegraShop and implement a REST API using ASP.NET Core;
- to develop three frontend clients (React, Angular, Vue) and ensure their full integration with the backend;
- to test the integration, evaluate the correctness of data exchange and the operability of the prototype;
- to justify the potential application of the developed system within a startup project.

Object of research – the integration process between ASP.NET Core applications and modern frontend tools.

Subject of research – methods, technologies, and architectural approaches that ensure integration between an ASP.NET Core backend and SPA frameworks.

Practical significance. The work provides a functional prototype of an integrated e-commerce system that can serve as a foundation for developing a full-scale web product. The implemented approach ensures modularity, scalability, and flexibility in connecting various frontend applications. The results may also be applied in real commercial systems, API-centric architectural design, and educational purposes in web development.

Methods of research. The research employs analytical methods for comparing integration technologies, software system design techniques, architectural modelling, as well as practical methods of software implementation and experimental verification of the developed web application components.

Approbation of results. The main findings of the thesis were presented as published conference abstracts at the XXII International Scientific and Practical Conference of Young Scientists and Students (April 22–25, 2025).

Structure and volume of the thesis. The thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices. The total volume is 103 pages, including 85 pages of main text, 1 appendice, 6 figures, 3 tables and a reference list of 13 sources.

Keywords: ASP.NET Core, REST API, integration, React, Angular, Vue.js, web development, e-commerce, client–server architecture.

ЗМІСТ

ВСТУП.....	11
1 ОГЛЯД МЕТОДІВ ІНТЕГРАЦІЇ ВЕБ-ЗАСТОСУНКІВ	14
1.1 Технологія REST API.....	14
1.2 Мова запитів GraphQL.....	17
1.3 Технологія SPA	20
1.4 Технологія Blazor	22
1.5 Бібліотека SignalR для реального часу	25
1.6 Протокол gRPC.....	29
1.7 Обслуговування статичних файлів.....	33
1.8 Порівняння методів інтеграції	35
1.9 Висновки до розділу 1	42
2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ	44
2.1 Постановка задачі.....	44
2.2 Архітектура системи.....	46
2.3 Вибір технологій та засобів реалізації	49
2.4 Розроблення REST API сервісу	51
2.5 Тестування та результати	54
2.6 Висновки до розділу 2	57
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ	58
3.1 Загальна архітектура системи.....	58
3.2 Архітектура бекенд-системи.....	59
3.3 Модель даних.....	61
3.4 Алгоритм обробки клієнтських запитів.....	62
3.5 Взаємодія фронтенд-застосунків з REST API.....	63
3.6 Висновки до розділу 3	64

4 ВСТАНОВЛЕННЯ, ЗАПУСК І ВИКОРИСТАННЯ ПРОГРАМНОГО

ЗАСТОСУНКУ	65
4.1 Вимоги до апаратного та програмного середовища.....	65
4.2 Структура програмної системи.....	66
4.3 Процедура інсталяції та налаштування	66
4.3.1 Налаштування бази даних	67
4.3.2 Запуск серверної частини.....	67
4.3.3 Запуск фронтендів.....	67
4.4 Основний функціонал застосунку	68
4.5 Висновки до розділу 4	69
5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ	70
5.1 Формування ідеї стартап-проєкту	71
5.1.1 Сутність ідеї.....	71
5.1.2 Напрями застосування.....	72
5.1.3 Загальні дані стартап-проєкту	73
5.2 Аналіз ринку	74
5.2.1 Оцінка місткості ринку.....	74
5.2.2 Цільові сегменти стартапу	75
5.3 Аналіз конкурентів.....	76
5.3.1 Основні конкуренти.....	76
5.3.2 Визначення характеристик ідеї проєкту	78
5.4 Аудит технологій	80
5.4.1 Технологічна основа продукту	80
5.4.2 Оцінка технологічної здійсненності	81
5.5 Бізнес-модель та монетизація	82
5.6 План впровадження стартап-проєкту.....	83
5.6.1 Стадія MVP (Minimum Viable Product).....	83
5.6.2 Стадія Launch (запуск на ринок)	84
5.6.3 Стадія Growth (ріст і масштабування)	85

5.7 Аналіз ризиків	87
5.7.1 Ринкові ризики	87
5.7.2 Технологічні ризики	89
5.7.3 Організаційні ризики	90
5.7.4 Фінансові ризики.....	91
5.7.5 Заходи мінімізації ризиків	93
5.8. Перспективи розвитку стартапу	94
5.9. Висновки до розділу 5	95
ВИСНОВКИ.....	96
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	98
ДОДАТОК А.....	100

ВСТУП

У сучасних умовах цифрової трансформації веб-застосунки стали ключовою технологічною основою більшості бізнес-процесів та сервісів. Вони забезпечують роботу систем електронної комерції, банківських платформ, сервісів управління контентом, освітніх порталів, корпоративних інформаційних систем і безлічі інших рішень, що працюють через браузер або мобільні клієнти. Спільною рисою таких платформ є поділ їхньої архітектури на клієнтську (frontend) та серверну (backend) частини. Від того, наскільки ефективно організована взаємодія між ними, залежить якість роботи всієї системи: її продуктивність, масштабованість, безпека та зручність супроводу.

Тривалий час найбільш розповсюдженим та практичним способом інтеграції була архітектура REST API, заснована на використанні HTTP-методів і передачі структурованих JSON-даних. Проте розвиток користувацьких інтерфейсів, ускладнення бізнес-логіки та поява нових вимог до взаємодії в режимі реального часу призвели до появи альтернатив. GraphQL, запропонований Facebook, забезпечує можливість формувати гнучкі запити, отримуючи рівно ті дані, які потрібні клієнту — ні більше, ні менше. У свою чергу, gRPC пропонує високошвидкісний двосторонній обмін повідомленнями з мінімальними накладними витратами, що робить його привабливим для мікросервісних систем та високонавантажених застосунків. Паралельно технологія SignalR відкрила можливість ефективного push-спілкування між клієнтом і сервером, що стало значним проривом для чатів, онлайн-сповіщень, кооперативних редакторів та інших сценаріїв.

Поруч із серверними технологіями стрімко розвивалися й клієнтські. Односторінкові застосунки (Single Page Applications, SPA) стали стандартом де-факто для сучасних веб-інтерфейсів. Фреймворки Angular, React та Vue дозволяють створювати складні, інтерактивні UI, що працюють без перезавантаження сторінок. Поява Blazor від Microsoft розширила можливості розробників на .NET, дозволивши

створювати SPA-подібні інтерфейси мовою C#, як на стороні клієнта (WebAssembly), так і на стороні сервера. Усе це формує широке й досить складне поле технологій, у якому розробнику необхідно орієнтуватися для вибору оптимальної технологічної зв'язки «клієнт–сервер».

Саме в цьому контексті й полягає актуальність даної роботи. Інтеграція між frontend та backend стала не просто технічним аспектом — вона є фундаментальною частиною архітектурного проектування сучасних веб-систем. Від вибору методу інтеграції залежить те, наскільки швидко працюватиме система, як просто її можна буде масштабувати і які обмеження накладатиме клієнтське середовище (браузер, мобільний додаток чи сторонній сервіс). На практиці багато компаній стикаються з дилемою: залишатися на перевіреному REST API, переходити на GraphQL, використовувати gRPC для продуктивності, комбінувати рішення з SignalR, чи будувати фронтенд на традиційному JavaScript-стеку або довірити інтерфейс Blazor. Робота покликана систематизувати ці підходи, порівняти їх і продемонструвати один з них на практиці.

Метою магістерської роботи є визначення ефективного підходу до інтеграції клієнтської та серверної частин веб-застосунку та реалізація програмного прототипу, який демонструє можливості обраного методу. У межах дослідження було проведено аналіз сучасних технологій інтеграції, сформовано порівняльні характеристики підходів (REST, GraphQL, gRPC, SignalR, SPA-фреймворки, Blazor), розроблено архітектуру веб-застосунку та реалізовано REST API на основі ASP.NET Core. Практична частина роботи включає створення прототипу інтернет-магазину з декількома клієнтськими інтерфейсами (React, Angular, Vue), що звертаються до спільного серверного API.

Об'єктом дослідження є процес інтеграції сервера та клієнта у веб-застосунках. Предметом дослідження — сучасні методи, протоколи та архітектурні підходи інтеграції, а також їхній вплив на продуктивність, гнучкість і масштабованість застосунку. Для досягнення поставленої мети використано методи аналізу та синтезу

наукових і технічних джерел, порівняльний аналіз технологій, методи програмної інженерії (моделювання архітектури, об'єктно-орієнтоване проєктування), а також експериментальний підхід через тестування створеного REST API та взаємодію з клієнтськими частинами.

Практичне значення результатів полягає у можливості використання створеної системи як навчального або проєктного прототипу для розроблення повноцінного веб-продукту. Реалізований сервер на ASP.NET Core демонструє підхід до побудови API, який може масштабуватися та доповнюватися іншими інтерфейсами. Прототип інтернет-магазину може бути розширений або адаптований під інші сценарії, а сформовані рекомендації щодо вибору методів інтеграції можуть використовуватися у реальних командах розробників при проєктуванні складних веб-рішень.

Отримані результати були апробовані під час XXII Міжнародної науково-практичної конференції молодих учених і студентів (22–25 квітня 2025 року), на якій були представлені тези за тематикою доповіді (Додаток А). Це підтверджує відповідність дослідження сучасним тенденціям та практичну цінність отриманих висновків.

Робота складається зі вступу, п'яти розділів, висновків, списку використаних джерел та додатків. Загальний обсяг становить 103 сторінок, містить 6 рисунків, 3 таблиці та 1 додаток. Основний текст викладено на 85 сторінках. У списку використаних джерел — 13 найменувань, з них 12 іноземною мовою. У першому розділі проведено огляд сучасних методів інтеграції веб-застосунків та проаналізовано ключові технології взаємодії клієнта і сервера. У другому розділі виконано проєктування та реалізацію серверної частини застосунку на основі REST API. Третій розділ присвячено детальній програмній реалізації системи. У четвертому розділі описано вимоги до середовища, структуру програмної системи, процедуру інсталяції, запуску та використання основного функціоналу застосунку. П'ятий розділ містить розроблення стартап-проєкту: сформовано його концепцію, проаналізовано ринок і конкурентів, виконано технологічний аудит, визначено бізнес-модель, ризики та перспективи подальшого розвитку продукту.

1 ОГЛЯД МЕТОДІВ ІНТЕГРАЦІЇ ВЕБ-ЗАСТОСУНКІВ

У цьому розділі здійснено огляд основних технологій та підходів, що використовуються для інтеграції клієнтської та серверної частин веб-застосунку. Під інтеграцією мається на увазі організація обміну даними між фронтом (інтерфейсом, що працює у браузері користувача) та бекендом (серверною логікою, базою даних тощо). Розглянуто як протоколи обміну даними (REST, GraphQL, gRPC, SignalR), так і архітектурні підходи до побудови користувацької частини (SPA, Blazor), а також базовий випадок статичного контенту. Для кожного методу описано принцип роботи, наводяться його сильні та слабкі сторони, а також типові сценарії використання. Наприкінці розділу подано узагальнююче порівняння розглянутих підходів.

1.1 Технологія REST API

REST API (Representational State Transfer Application Programming Interface) — це архітектурний стиль проєктування програмних інтерфейсів, що визначає принципи побудови розподілених систем, насамперед веб-служб. Він не є протоколом, а формує набір правил взаємодії між клієнтом і сервером. Концепція REST була сформульована Роєм Філдіном у 2000 році й стала основою сучасної веб-архітектури, на якій базується HTTP [1]. У межах REST дані подаються у вигляді ресурсів, доступних через уніфіковані URL-адреси, а взаємодія з ними здійснюється за допомогою стандартних HTTP-методів (GET, POST, PUT, DELETE), що відповідають операціям читання, створення, оновлення та видалення.

До ключових принципів REST належать безстановість взаємодії, коли кожен запит містить повну інформацію, необхідну для його обробки; єдиний уніфікований

інтерфейс, який забезпечує однакові механізми роботи з ресурсами; чітке розмежування клієнтської та серверної частин, що дозволяє їм розвиватися незалежно; а також підтримка кешування, завдяки якому значно зменшується навантаження на сервер [1]. У практичному сенсі під REST API зазвичай розуміють веб-інтерфейс, що передає дані у форматі JSON через HTTP, забезпечуючи просту та інтуїтивну взаємодію.

REST-підхід став надзвичайно популярним завдяки використанню стандартних веб-технологій, зрозумілим URL-адресам і легкості роботи з JSON-даними[13]. Більшість сучасних публічних API (Facebook, Twitter, Google Maps тощо) реалізовані у форматі REST, оскільки такі сервіси легко тестувати, відлагоджувати та інтегрувати з різними клієнтськими застосунками. Крім того, безстановий характер REST спрощує горизонтальне масштабування: додавання нових серверних вузлів не потребує перенесення стану між ними.

Серед переваг REST API варто відзначити передусім простоту та стандартизованість. Оскільки REST спирається на HTTP-методи й стандартні коди стану, розробникам не потрібно винаходити окремі механізми обробки запитів, а налагодження стає значно швидшим. Другою важливою перевагою є логічна та зрозуміла структура ресурсів: побудова URL-адрес за ієрархічним принципом підвищує читабельність і полегшує навігацію. Третьою перевагою виступає ефективне кешування, підтримуване на рівні HTTP-протоколу, що суттєво зменшує кількість повторних звернень до сервера. Окрім того, REST має найширшу підтримку серед мов програмування, бібліотек та інструментів, включаючи автоматично згенеровану документацію OpenAPI/Swagger, що робить його універсальним рішенням для інтеграції клієнтських застосунків [5].

Попри поширеність REST API, цей підхід має низку обмежень, які стають помітними у проєктах зі складною структурою даних. Однією з найбільш відомих проблем є феномен надлишкового та недостатнього отримання даних. Оскільки структура REST-ендпоінтів є фіксованою, сервер часто повертає більше інформації,

ніж потрібно клієнту, або, навпаки, вимагає від клієнта виконати декілька окремих запитів, щоб зібрати всі необхідні дані. Наприклад, для відображення профілю користувача разом із трьома останніми публікаціями може знадобитися послідовне звернення до різних ресурсів, що збільшує затримки та мережевий трафік. Саме ці недоліки стали передумовою появи GraphQL — технології, орієнтованої на гнучке формування запитів і уникнення надлишкових відповідей [2].

Ще одним недоліком REST є відсутність суворої типізації. На відміну від gRPC, який спирається на строго описані контракти даних (формат Protocol Buffers), або GraphQL із його схемою типів, REST використовує декларативну документацію (наприклад, OpenAPI), яка не гарантує повної сумісності між клієнтом і сервером. Це може ускладнювати автоматичну генерацію клієнтських SDK та збільшувати ризик помилок, пов'язаних із невідповідністю форматів.

Проблемним є також питання версіонування. Будь-які суттєві зміни у структурі ресурсів — додавання нових полів, зміна форматів або поведінки ендпоїнтів — потенційно можуть порушити роботу клієнтів, які використовують попередню версію API. Тому підтримка кількох версій REST API часто потребує дублювання логіки та ускладнює супровід проєкту.

Ще одною обмежувальною характеристикою REST є його запитно-орієнтована модель. Оскільки сервер не може ініціювати передачу даних самотійно, побудова реального часу вимагає окремих технологій, які виходять за межі REST-підходу. Наприклад, для push-повідомлень або потокового оновлення використовуються WebSocket, Server-Sent Events або спеціалізовані фреймворки на кшталт SignalR, що інтегрують ці механізми в екосистему .NET [6].

Попри зазначені обмеження, REST API продовжує широко застосовуватися у різноманітних веб-системах. Він добре підходить для публічних сервісів із чіткою структурою даних, стабільними бізнес-вимогами та необхідністю забезпечити просту інтеграцію з широким колом клієнтів. REST також залишається фундаментальним компонентом багатьох корпоративних систем і мікросервісних архітектур, де важливі

стандартизованість, зрозумілість та легкість налагодження. У випадках, коли потрібна більша гнучкість запитів або висока пропускна здатність, можливо обрати альтернативні рішення — GraphQL чи gRPC [7][8], однак саме REST завдяки своїй універсальності та зрілості все ще є найбільш розповсюдженим механізмом інтеграції клієнтських застосунків.

1.2 Мова запитів GraphQL

GraphQL є мовою запитів до API та середовищем виконання цих запитів, розробленою компанією Facebook з метою вирішення типових обмежень REST-підходу. Уперше технологію використано у внутрішніх системах компанії у 2012 році, а публічну специфікацію оприлюднено у 2015 році. На відміну від REST, де для отримання різних ресурсів використовуються різні URL-ендпоїнти, GraphQL працює через єдину кінцеву точку (зазвичай /graphql), у яку клієнт надсилає запити із чітко визначеною структурою даних, що мають бути повернуті сервером [4]. Це означає, що клієнт сам формує необхідний набір полів та вкладених об'єктів, а сервер повертає рівно ту інформацію, що була запитана, у форматі JSON.

GraphQL можна розглядати як інструмент, що надає клієнту повний контроль над формою та обсягом отримуваних даних. Якщо у REST для отримання пов'язаної інформації часто потрібно виконувати кілька запитів (наприклад, спочатку отримати дані користувача, а потім — список його постів), то у GraphQL ці дані можуть бути отримані в межах одного запиту. Клієнт визначає структуру відповіді, вказуючи лише потрібні поля та обмеження. Завдяки цьому усувається проблема надлишкового або недостатнього завантаження даних — одна з найбільш поширених вад REST API, на яку GraphQL із самого початку відповідав як на ключову проблему [2][3]. Таким чином, GraphQL дозволяє оптимізувати взаємодію з сервером, зменшуючи кількість HTTP-викликів та обсяг переданої інформації.

Фундаментальною частиною GraphQL є суворо типізована схема — формальний опис усіх типів даних, їхніх полів, зв'язків та запитів, які можна виконувати. Схема відіграє роль контракту між клієнтом і сервером: вона задає структуру API, дозволяючи інструментам виконувати перевірки ще до надсилання запиту. Наприклад, клієнт не може звернутися до неіснуючого поля — це буде виявлено під час компіляції запиту, а сервер поверне структуровану інформацію про помилку. При цьому, відповідно до специфікації, навіть коли запит містить логічні помилки, HTTP-відповідь зазвичай має код 200, оскільки сама операція була виконана протоколом коректно, а помилка є частиною бізнес-логіки API.

Підтримка суворої типізації, єдиного ендпоїнта, можливість виконувати складні запити в одному зверненні та усунення проблеми надлишкових даних роблять GraphQL особливо ефективним у складних клієнтських застосунках і системах з великою кількістю взаємопов'язаних сутностей. Завдяки цьому GraphQL активно впроваджується у великих компаніях та у проектах, де REST виявляється недостатньо гнучким для оптимального обміну даними.

GraphQL має низку суттєвих переваг, що пояснюють його швидке поширення у складних веб-системах. Однією з ключових є гнучкість формування запитів: клієнт сам визначає, які саме поля слід повернути, що дозволяє оптимізувати обсяг переданих даних. Це особливо важливо для мобільних клієнтів і SPA-застосунків, де пропускна здатність мережі може бути обмеженою, а різні екрани потребують різних наборів інформації. Завдяки можливості вкладених запитів зменшується кількість HTTP-викликів: дані, які у REST доводиться отримувати кількома окремими зверненнями, у GraphQL надходять одним пакетом. Таким чином скорочується сумарна затримка та навантаження на мережеві канали. Додатковою перевагою є суворі типізація: GraphQL-схема точно описує можливі операції та структуру даних, а інструменти на кшталт GraphiQL чи GraphQL Playground дозволяють досліджувати API інтерактивно. На основі схеми можна автоматично генерувати документацію та клієнтські типи, що мінімізує ризик помилок сумісності. Окремо важливою

властивістю є наявність єдиного endpoint'a — це спрощує налаштування маршрутизації та політик безпеки. У разі зміни структури моделі даних GraphQL дозволяє додавати нові поля, не порушуючи існуючі клієнтські запити; старі поля можуть позначатися як застарілі, що зменшує потребу у версіонуванні API [2][4].

Попри значні переваги, GraphQL має і помітні недоліки. Гнучкість запитів призводить до більшої складності на боці сервера: необхідно визначати резолвери для кожного поля та контролювати навантаження, яке може створювати надто глибокий або широкий запит. Сервер має реалізовувати механізми захисту, зокрема ліміти глибини, обмеження складності запитів і системи кешування для повторюваних структур. Також виникають труднощі з кешуванням на рівні HTTP, оскільки всі запити надсилаються через одну й ту саму URL-адресу; це унеможливорює використання класичних CDN-підходів і потребує використання спеціальних технік, таких як persisted queries чи кешування окремих резолверів [2][3]. Для простих CRUD-сценаріїв GraphQL може бути навіть надмірним рішенням: початкове налаштування, створення схеми, резолверів та інфраструктурні витрати можуть перевищувати вигоду від використання. Додатковою перешкодою є вищий поріг входження: розробники повинні опанувати нову модель роботи з API, зокрема поняття запитів, мутацій та підписок, а також специфічні засоби розробки, які ще не мають такої зрілості, як екосистема REST.

Разом із тим GraphQL найкраще проявляє себе у сценаріях, де клієнти мають різноманітні й складні інформаційні потреби. Для SPA-застосунків і мобільних клієнтів, де кожен екран споживає власну комбінацію даних, GraphQL дозволяє уникати створення численних спеціалізованих REST-ендпоінтів. Він також ефективний у системах, що агрегують дані з різних джерел: один GraphQL-сервер може об'єднувати інформацію з декількох мікросервісів або баз даних під єдиною схемою, підвищуючи узгодженість API. Саме тому великі компанії, такі як GitHub, Shopify, Netflix чи Pinterest, активно застосовують GraphQL у своїх продуктах. У той же час для простих загальнодоступних API, які мають виконувати мінімальний набір

операцій, REST залишається більш зручним: він не потребує складної інфраструктури і легко інтегрується з широкою екосистемою HTTP-інструментів.

Окрім стандартних операцій отримання даних, GraphQL підтримує зміни (mutation) та механізм підписок (subscription), які забезпечують передачу подій у реальному часі через WebSocket-з'єднання. Завдяки цьому GraphQL може покривати практично весь спектр інтеграційних сценаріїв між клієнтом і сервером, включаючи динамічне оновлення інтерфейсу, синхронізацію даних і складні багатоджерельні запити.

1.3 Технологія SPA

Односторінковий веб-застосунок (Single Page Application, SPA) являє собою архітектурний підхід до побудови клієнтських інтерфейсів, у якому веб-сторінка завантажується лише один раз, а всі подальші зміни інтерфейсу виконуються без повного перезавантаження. На відміну від традиційних багатосторінкових веб-сайтів, де кожен перехід супроводжується отриманням нового HTML-документа від сервера, SPA завантажує базову сторінку та необхідні скрипти під час першого входу користувача, після чого управління інтерфейсом повністю здійснюється на стороні браузера. Усі зміни контенту забезпечує JavaScript, який оновлює структуру DOM та, за потреби, змінює адресу в рядку браузера за допомогою технології History API. Дані для формування інтерфейсу отримуються через асинхронні запити до сервера — здебільшого це REST API чи GraphQL, що передають інформацію у форматі JSON. Таким чином, сервер у SPA-архітектурі відповідає лише за надання даних, а не за формування HTML-сторінок [5].

Односторінкові застосунки характеризуються значною інтерактивністю, оскільки відсутність повного перезавантаження забезпечує плавний та швидкий відгук на дії користувача. Така модель дозволяє реалізовувати складні інтерфейси, які

потребують частих локальних змін — від динамічних форм до інтерактивних карт та редакторів. Крім того, перенесення більшої частини логіки на клієнт зменшує кількість роботи, яку повинен виконувати сервер, адже замість повторного рендерингу сторінок він лише передає дані, що нерідко знижує навантаження на серверну інфраструктуру. SPA також сприяє розділенню роботи між фронтенд- і бекенд-командами: сервер може розвиватися як незалежний API-сервіс, тоді як клієнт реалізується у вигляді повноцінного застосунку на фреймворках Angular, React чи Vue. Завдяки такому підходу API, створений для SPA, може бути повторно використаний у мобільних застосунках або інших клієнтах [11].

Проте підхід SPA має й низку обмежень. Одним з них є триваліше початкове завантаження, оскільки браузер мусить отримати значний обсяг JavaScript-коду перед початком роботи. Хоча сучасні методи оптимізації — зокрема, поділ коду на модулі та динамічне завантаження — частково компенсують цю проблему, перша взаємодія користувача все одно може бути повільнішою, ніж у традиційних веб-сайтах. Додатковою складністю є SEO-оптимізація: незважаючи на те, що сучасні пошукові системи можуть індексувати JavaScript, SPA-додатки все ще потребують SSR або prerendering для повноцінної індексації. Певні труднощі виникають і з маршрутизацією: застосунок має коректно обробляти переміщення назад/вперед у браузері та зберігати узгодженість URL із поточним станом інтерфейсу. До цього додаються підвищені вимоги до безпеки, адже велика частина логіки виконується на клієнті, а отже, потрібно ретельно захищати від XSS-вразливостей та забезпечувати перевірку прав доступу на стороні сервера.

Незважаючи на певні недоліки, SPA стали одним із ключових стандартів у сучасній веб-розробці, особливо для застосунків, що вимагають високої інтерактивності й динамічного оновлення контенту. Приклади таких рішень включають поштові клієнти, адміністративні панелі, інструменти керування проєктами та соціальні сервіси. Для проєктів із мінімальними взаємодіями чи переважно статичним контентом SPA зазвичай є надмірно складним рішенням,

оскільки традиційні серверні підходи забезпечують швидше початкове завантаження та простішу реалізацію. Щодо .NET-екосистеми, SPA може інтегруватися з ASP.NET Core як окремий застосунок, що розгортається разом із серверною частиною або на стороні CDN. Існують офіційні шаблони Microsoft, що надають інтеграцію Angular чи React з ASP.NET Core, де бекенд виконує роль хоста статичних файлів та одночасно надає REST API [12]. Такий підхід дозволяє поєднувати гнучкість SPA із повноцінними можливостями серверної платформи.

1.4 Технологія Blazor

Blazor — це сучасний веб-фреймворк від Microsoft, інтегрований у платформу ASP.NET Core, що дає змогу створювати інтерактивні користувацькі інтерфейси, використовуючи C# замість JavaScript. Назва технології поєднує поняття Browser та Razor, підкреслюючи її здатність рендерити Razor-компоненти безпосередньо в браузері. Основна ідея Blazor полягає у тому, що логіка інтерфейсу може бути реалізована на C#, а компоненти — повторно використовувані на боці клієнта або сервера залежно від обраної моделі хостингу. Такий підхід дозволяє розробникам .NET створювати повноцінні SPA-застосунки, не змінюючи технологічного стеку та зберігаючи єдину мову програмування для всієї веб-системи [11].

Моделі хостингу Blazor:

- Blazor WebAssembly (WASM).

У цій моделі застосунок повністю виконується на стороні клієнта: браузер завантажує WebAssembly-модуль із вбудованим .NET Runtime, після чого завантажуються збірки застосунку у форматі DLL. Компоненти виконуються локально — у пісочниці WebAssembly — і взаємодіють із DOM так само, як JavaScript-код. Для отримання даних використовується стандартна комунікація через HTTP-запити до REST API або GraphQL, а для роботи у режимі реального часу — SignalR.

Застосунок може працювати автономно після першого завантаження, що дозволяє реалізовувати PWA-сценарії та офлайн-режим. Суттєвою перевагою є можливість повторного використання коду між клієнтом і сервером, зокрема моделей даних та бізнес-логіки у вигляді спільних бібліотек. До недоліків належить початковий розмір завантаження, адже браузеру потрібно отримати .NET Runtime і збірки, що може уповільнити старт. Проте сучасні оптимізації WebAssembly та .NET (зокрема tree shaking та trimming) значно зменшують ці накладні витрати. Blazor WASM доцільний у випадках, коли потрібен автономний або високонавантажений клієнтський застосунок, але зберігається бажання використовувати C# на всіх рівнях.

- Blazor Server.

Цей варіант працює за принципово іншою моделлю: компоненти виконуються на сервері, а браузер виступає «тонким клієнтом», який лише відображає інтерфейс і приймає оновлення через SignalR-підключення в режимі реального часу [6]. Після завантаження сторінки встановлюється постійне WebSocket-з'єднання, через яке браузер передає події (натискання кнопок, зміни елементів) на сервер. Там відбувається виконання логіки компонентів і обчислення нового стану, який повертається клієнту у вигляді diff-оновлень DOM. Це дозволяє забезпечити швидкий початковий рендеринг, оскільки на клієнта надходить лише мінімальний обсяг HTML і JavaScript. Модель Blazor Server є безпечнішою: у браузер не передаються збірки, конфіденційні дані залишаються на сервері, а логіка доступу повністю контролюється бекендом. Разом з тим, існують обмеження: необхідність підтримувати стабільне мережеве з'єднання, збільшене навантаження на сервер (оскільки кожен користувач утримує активну SignalR-сесію) та потенційні затримки у взаємодії при повільному інтернеті. Попри це, Blazor Server підходить для корпоративних систем, внутрішніх панелей керування та сценаріїв, де важливі швидкий старт і централізований контроль логіки.

Обидві моделі Blazor демонструють еволюцію підходів до веб-розробки в екосистемі .NET та дають змогу створювати SPA-застосунки без використання

JavaScript-фреймворків. Вибір між ними залежить від вимог до продуктивності, автономності, безпеки та можливостей масштабування.

Blazor має низку суттєвих переваг, які роблять його привабливою альтернативою традиційним JavaScript-фреймворкам. Насамперед він забезпечує можливість використовувати єдиний технологічний стек .NET на клієнті та сервері, що дозволяє створювати інтерфейс та бізнес-логіку мовою C# і повторно застосовувати значну частину вже існуючих бібліотек та інструментів. Такий підхід значно знижує поріг входження для команд, орієнтованих на .NET, і дозволяє розробникам без глибоких знань JavaScript створювати сучасні веб-інтерфейси. Завдяки спільному використанню класів, моделей та валідаторів між різними частинами системи суттєво зменшується дублювання коду, особливо у Blazor WebAssembly, де клієнтський застосунок може безпосередньо використовувати моделі доменної логіки, спільні з бекендом. Додатковою перевагою є вбудована підтримка інтерактивності в режимі реального часу у Blazor Server — оскільки цей режим використовує SignalR, зміни інтерфейсу автоматично передаються клієнту без додаткового програмування механізмів push-сповіщень [6]. Важливою також є тісна інтеграція Blazor з екосистемою .NET: у варіанті Blazor Server серверні бібліотеки доступні безпосередньо, а за потреби можна застосовувати JavaScript-функції через механізм JS-interop, що забезпечує гнучкість при роботі з існуючими сторонніми засобами.

Водночас Blazor має й певні недоліки. Для Blazor WebAssembly характерним є відносно великий початковий обсяг завантаження — запуск .NET Runtime та клієнтських збірок може займати додатковий час, що небажано для легких або високочастотних веб-сторінок. Хоча оптимізації .NET значно зменшили ці витрати, початкове завантаження залишається важливим фактором [11]. Іншим обмеженням є порівняно невелика зрілість екосистеми: хоча Blazor активно розвивається, кількість готових UI-бібліотек або спеціалізованих компонентів для нього менша, ніж у традиційних JavaScript-фреймворків, а інтеграція зі складними JS-інструментами

інколи потребує додаткових зусиль. Blazor WebAssembly також успадковує обмеження браузерної пісочниці, подібно до JavaScript, що унеможливорює прямий доступ до системних ресурсів. Натомість Blazor Server, хоч і позбавлений великих завантажень, має інше ключове обмеження — залежність від стабільного з'єднання з сервером: у разі переривання зв'язку застосунок стає недоступним, а також збільшується навантаження на сервер через необхідність утримання активних сесій SignalR.

У підсумку Blazor можна розглядати як потужну альтернативу JavaScript-фреймворкам для команд, орієнтованих на .NET, що прагнуть реалізувати SPA-підхід без зміни технологічного стеку. Вибір між Blazor WebAssembly та Blazor Server залежить від вимог конкретного проєкту: перший підходить для автономної роботи та складних клієнтських сценаріїв, тоді як другий оптимальний для систем, де важливі швидке завантаження і централізоване виконання логіки. Blazor також сумісний із REST API та іншими методами інтеграції, що робить його зручним інструментом у багатокомпонентних рішеннях і дозволяє будувати як гібридні, так і повністю клієнтські застосунки.

1.5 Бібліотека SignalR для реального часу

SignalR — це технологія з екосистеми ASP.NET, призначена для реалізації функціональності обміну даними в режимі реального часу, коли сервер має можливість негайно надсилати оновлення клієнтам без необхідності їх постійного опитування. У традиційній моделі взаємодії HTTP-запити завжди ініціює клієнт, тому для отримання актуальних даних (наприклад, у веб-чаті) клієнт змушений регулярно звертатися до сервера з періодичними запитами. Такий підхід створює затримки, генерує зайве навантаження на мережу й погано масштабується у випадку великої кількості користувачів. SignalR вирішує ці проблеми, забезпечуючи постійний канал

зв'язку між клієнтом і сервером, завдяки якому сервер може надсилати повідомлення в момент їх появи, підтримуючи тим самим справжню модель real-time-взаємодії [6].

У центрі роботи SignalR знаходиться концепція хаба — спеціального серверного компонента, який інкапсулює логіку обміну даними та дозволяє виконувати методи як на сервері, так і на клієнті. Клієнтські застосунки (JavaScript, .NET або інші платформи, що мають відповідні бібліотеки) встановлюють підключення до хаба, після чого обидві сторони можуть надсилати повідомлення одна одній у двонапрямленому режимі. Для забезпечення стабільної роботи SignalR автоматично вибирає оптимальний транспорт: у сучасних браузерах це майже завжди WebSocket, протокол, який надає повноцінне двостороннє TCP-з'єднання. Якщо ж WebSocket недоступний, система прозоро переходить на альтернативні механізми — такі як Server-Sent Events чи Long Polling. Завдяки цьому SignalR залишається сумісним із широким спектром платформ і умов виконання, забезпечуючи однакову модель програмування незалежно від того, на якому транспорті фактично працює зв'язок [10].

Таким чином, SignalR відіграє роль потужного інструмента для створення інтерактивних веб-застосунків — чатів, панелей моніторингу, багатокористувацьких систем та будь-яких рішень, де важлива миттєва синхронізація даних. Його здатність абстрагувати технічні деталі та автоматично керувати встановленням і підтримкою каналу зв'язку значно спрощує розробку real-time-функціональності у веб-середовищі.

Практичний сценарій демонструє роботу SignalR, наприклад у багатокористувацьких іграх або системах, що мають забезпечувати миттєве оновлення даних у всіх відкритих сесіях — таких як лічильники відвідувачів чи індикатори активності. У таких випадках сервер може ініціювати надсилання змін усім підключеним клієнтам, а клієнтська частина одразу відобразить оновлення інтерфейсу. Завдяки можливості групувати підключення або адресувати повідомлення окремим користувачам, SignalR спрощує реалізацію як широкомовної,

так і цільової комунікації, усуваючи необхідність розробляти власні механізми маршрутизації повідомлень поперх WebSocket-з'єднань. Саме завдяки цьому підходу розробник отримує значно вищий рівень абстракції, ніж при використанні WebSocket безпосередньо [6].

Серед ключових переваг SignalR можна виділити зручність програмної моделі: розробнику не потрібно взаємодіяти з низькорівневими транспортами чи забезпечувати серіалізацію даних вручну — ці завдання бере на себе інфраструктура фреймворку, яка підтримує як текстові, так і бінарні формати. Тісна інтеграція з ASP.NET Core дозволяє застосовувати автентифікацію та авторизацію, використовувати механізми впровадження залежностей та організовувати доступ до бізнес-логіки безпосередньо із хабів. Іншою важливою перевагою є масштабованість: SignalR підтримує розподіл навантаження між кількома серверами за допомогою backplane-рішень, а у хмарних середовищах, зокрема Azure, доступна керована служба Azure SignalR Service, яка фактично знімає із застосунку проблему обробки великої кількості одночасних з'єднань. Крім того, екосистема клієнтів SignalR є багатоплатформенною, тому одним хабом можуть користуватися браузерні, мобільні та настільні застосунки.

Поряд із перевагами, SignalR має низку специфічних обмежень. Зокрема, постійне підтримання з'єднань створює додаткове навантаження на сервер, що може стати критичним при великій кількості одночасних клієнтів; у таких випадках потрібне грамотне масштабування. Реалізація асинхронних сценаріїв також ускладнює налагодження, оскільки поведінка системи залежить від великої кількості паралельних подій, а відключення клієнтів, повторні підключення або тимчасові збої в мережі потребують окремого опрацювання. Інфраструктурні вимоги також можуть створювати труднощі — робота WebSocket інколи потребує спеціальних налаштувань проксі, балансувальників чи SSL-термінаторів. Нарешті, SignalR прив'язаний до .NET-екосистеми, що обмежує його застосування у проєктах, побудованих на інших

серверних технологіях, хоча існують альтернативи на кшталт Socket.IO у середовищі Node.js [10].

Сфери застосування SignalR охоплюють будь-які системи, які вимагають миттєвої синхронізації даних між користувачами. Це можуть бути чат-платформи та засоби онлайн-комунікації, системи сповіщень, панелі моніторингу, де дані оновлюються в реальному часі, сервіси спільної роботи над документами або інтерактивні навчальні ресурси. У веб-іграх SignalR використовується для синхронізації стану гри між усіма учасниками, а в бізнес-додатках — для відображення змін у режимі реального часу, наприклад, оновлення статусів замовлень чи результатів аналітики. У всіх цих випадках бібліотека забезпечує розробникам простий, уніфікований механізм організації real-time-взаємодії без складних налаштувань низькорівневих мережевих протоколів.

У контексті продуктивності SignalR варто враховувати, що різні технології реального часу демонструють свої сильні сторони залежно від типу навантаження. Практичні порівняння, проведені спільнотою .NET, показують, що при звичайних одиничних запитах класичні REST-виклики можуть працювати швидше завдяки мінімальним накладним витратам HTTP та відсутності постійного з'єднання. Однак у сценаріях, де йдеться про масову публікацію повідомлень усім підключеним клієнтам, SignalR демонструє помітно кращу ефективність, оскільки WebSocket-підключення дозволяє передавати дані миттєво та без повторних мережевих рукоштовувань [9]. Водночас у випадках, коли потрібна безперервна потокова передача великих обсягів даних — наприклад, телеметрія, відео або інтенсивні дані сенсорів — на перший план виходять можливості gRPC, який забезпечує оптимізоване бінарне кодування й ефективний стрімінг завдяки своїй низькорівневій архітектурі [7, 11].

Таким чином, продуктивність рішень реального часу залежить від конкретного сценарію: SignalR добре підходить для систем, де потрібні широкомовні або групові повідомлення з помірною частотою оновлень, тоді як gRPC доцільніше застосовувати в разі інтенсивних поточкових обчислень або коли ключову роль відіграє максимальна

пропускна здатність та мінімальна затримка. Це підтверджує необхідність диференційованого вибору технології, виходячи з природи навантаження та вимог до інтерактивності.

1.6 Протокол gRPC

gRPC – це сучасний високопродуктивний фреймворк для віддаленого виклику процедур (Remote Procedure Call), створений компанією Google та опублікований як проєкт з відкритим кодом. Назва є рекурсивним скороченням (“gRPC Remote Procedure Calls”). На відміну від традиційних веб-API, gRPC працює поверх протоколу HTTP/2 і використовує бінарний формат серіалізації Protocol Buffers (protobuf). Завдяки цьому клієнт і сервер обмінюються компактними двійковими повідомленнями, структура яких суворо визначена контрактом у файлі .proto [7].

Однією з ключових особливостей gRPC є підхід schema-first: розробник спочатку описує сервіс, методи, параметри та типи повернення у protobuf-схемі, після чого автоматично генерується і серверний, і клієнтський код. Такий підхід забезпечує типобезпечність та синхронність моделей на всіх платформах. Іншою важливою перевагою є ефективність бінарного формату, який значно компактніший і швидше десеріалізується, ніж JSON, що робить gRPC особливо вигідним для високонавантажених або ресурсно обмежених систем.

Фреймворк також використовує можливості HTTP/2, такі як мультиплексування потоків (одночасне виконання кількох запитів через одне TCP-з’єднання), двосторонній стрімінг, стиснення заголовків та оптимізоване використання каналу зв’язку. Це дозволяє виконувати багато паралельних викликів без створення нових з’єднань та з мінімальними накладними витратами. Завдяки офіційним генераторам коду під C#, Java, Go, Python та інші мови програмування,

gRPC добре підходить для мікросервісних систем, де сервіси можуть бути створені на різних технологіях, але взаємодіяти через спільний protobuf-контракт [10].

Серед ключових переваг gRPC насамперед виділяють його високу продуктивність та низьку затримку. Завдяки бінарному формату даних Protocol Buffers і використанню HTTP/2, виклики gRPC у багатьох сценаріях працюють значно швидше, ніж традиційні REST-запити на основі JSON. У практиці розробки неодноразово зазначалося, що серіалізація protobuf може бути в декілька разів швидшою, а обсяг переданих даних — суттєво меншим, що особливо помітно при передачі малих або часто повторюваних повідомлень. Такі властивості роблять gRPC ефективним рішенням у сервісах, де критичними є мінімальні затримки та висока пропускна здатність.

Ще однією сильною стороною gRPC є сувора типізація та автоматична генерація коду. Контракт у форматі .proto дозволяє чітко визначити структуру сервісів, методів та повідомлень, а інструменти генерації створюють цілком готові клієнтські та серверні обгортки для різних мов програмування. Такий підхід значно зменшує ризик несумісностей і помилок, характерних для систем із неформальними контрактами, та забезпечує відчутно вищу надійність. Для розробника виклики gRPC часто виглядають як звичайні асинхронні методи, хоча фактично вони виконуються на віддаленому сервері, що спрощує інтеграцію та значно підвищує продуктивність праці [7].

Важливою особливістю gRPC є його розширені можливості для передачі даних у потоковому режимі. На відміну від REST, який здебільшого обмежений моделлю запит-відповідь, gRPC підтримує односторонній та двосторонній стрімінг. Це означає, що сервер може надсилати клієнту послідовність повідомлень у рамках одного виклику, клієнт може робити те саме у відповідь, або обидві сторони можуть одночасно обмінюватися потоками даних у режимі реального часу [10]. Така модель дозволяє реалізовувати складні сценарії, як-от онлайн-спілкування, телеметрія,

трансляція оновлень або передача великих файлів частинами без додаткових надбудов.

Крім того, gRPC має низку вбудованих механізмів, які у REST доводиться реалізовувати вручну або через зовнішні бібліотеки. До таких належать управління таймаутами та дедлайнами, можливість каскадного скасування запитів, автоматична підтримка повторних викликів для ідемпотентних операцій та інші інструменти, що спрощують розробку надійних розподілених систем. Сукупність цих особливостей робить gRPC не просто альтернативою REST, а повноцінним інструментом для створення високопродуктивних сервісів із чітко визначеним контрактом і розширеними можливостями взаємодії.

Попри очевидні переваги, gRPC має низку суттєвих обмежень, які впливають на вибір цієї технології у веб-інтеграції. Найпомітнішою перепорою є обмежена підтримка в браузері: сучасні браузери не дозволяють JavaScript-коду напряду працювати з HTTP/2 на низькому рівні та відправляти двійкові повідомлення у вигляді, потрібному для «чистого» gRPC. Саме тому було створено проміжний стандарт gRPC-Web, який адаптує виклики до формату, сумісного з браузером, і доставляє їх на сервер через проксі або спеціальний серверний адаптер. У середовищі .NET така можливість реалізована на рівні ASP.NET Core, однак це збільшує складність інфраструктури й робить взаємодію менш прямою та прозорою [10]. Через це gRPC у веб-клієнтських сценаріях зазвичай програє REST або GraphQL за простотою інтеграції, тоді як для мобільних застосунків чи внутрішньої взаємодії мікросервісів він усе ще залишається оптимальним варіантом.

Додатковою складністю є відлагодження та аналіз роботи сервісів. На відміну від REST-запитів, які легко переглядати через браузерні інструменти чи Postman, gRPC використовує двійковий формат поверх HTTP/2, що робить його менш прозорим. Для дослідження таких запитів часто потрібні спеціалізовані інструменти на кшталт Wireshark із protobuf-декодером або консольні утиліти, такі як grpcurl. Хоча

вони надають необхідну функціональність, поріг входження для аналізу все одиниц, ніж у випадку традиційних текстових протоколів JSON.

Певні обмеження пов'язані і з версіонуванням. Хоч Protocol Buffers підтримують додавання нових полів без порушення зворотної сумісності, зміна типів або вилучення полів може спричинити приховані помилки, якщо клієнт і сервер працюють із різними версіями контракту. Цей підхід є чутливішим, ніж у текстових форматах, де невідомі поля просто ігноруються. Тому планування структур protobuf вимагає більших зусиль і уважності до нумерації полів.

Ще одним технічним аспектом є необхідність обов'язкової генерації коду із *.proto-файлів. Хоча у .NET процес інтегрований у систему збірки проєктів і здебільшого виконується автоматично, при частих змінах контрактів потрібно синхронно оновлювати схеми та повторно генерувати клієнтський і серверний код, що ускладнює підтримку та змінює характер розробки.

Попри зазначені недоліки, gRPC займає вагоме місце у сучасних розподілених системах. Він широко використовується у великих мікросервісних інфраструктурах, включно з платформами Google, де ефективність обміну між сервісами є критичною. У мобільній розробці gRPC також демонструє переваги завдяки можливості напряду використовувати protobuf і отримувати швидшу передачу даних, ніж при взаємодії через REST. В екосистемі .NET ця технологія стала офіційною заміною для застарілого WCF і поступово витісняє громіздкі REST-підходи в сценаріях міжсервісної взаємодії [7]. При цьому для публічних веб-API вона використовується значно рідше, адже вимагає від інтеграторів додаткових знань і спеціальних клієнтів, на відміну від універсального й загальнодоступного REST. Таким чином, gRPC найчастіше виступає «внутрішнім» високопродуктивним транспортом, який забезпечує ефективну роботу складних бекенд-систем, тоді як REST і GraphQL залишаються домінуючими методами публічної інтеграції.

1.7 Обслуговування статичних файлів

Найпростішою формою взаємодії між клієнтом і сервером є сценарій, у якому сервер не виконує жодної бізнес-логіки, а лише повертає статичні ресурси: HTML-розмітку, таблиці стилів, зображення та скрипти. У такому випадку інтеграція фактично відсутня, оскільки між клієнтом і сервером немає динамічного обміну даними. Такий підхід характерний для класичних статичних сайтів, лендингових сторінок або інших веб-ресурсів, де контент змінюється рідко і попередньо формується розробником. Перевагою є максимальна простота та швидка доставка – статичні файли можуть кешуватися браузером або CDN, тому вміст завантажується миттєво. Проте очевидним недоліком є неспроможність відображати актуальний або персоналізований контент без повторного розгортання сайту, що робить модель малопридатною для застосунків, де інформація повинна оновлюватися регулярно.

Окремий, більш сучасний варіант використання статичних ресурсів пов'язаний із розгортанням односторінкових веб-застосунків. У цьому випадку клієнтська частина (створена на Angular, React, Vue або у вигляді Blazor WebAssembly) компілюється у статичний набір файлів, які сервер лише видає користувачеві. Після початкового завантаження саме браузерний застосунок здійснює інтеграцію з бекендом через REST або інші види API, тоді як роль сервера зводиться до мінімального хостингу зібраного фронтенду. В екосистемі ASP.NET Core це реалізується за допомогою middleware Static Files, що дозволяє розмістити зібраний клієнтський застосунок у каталозі `wwwroot`. Така модель добре масштабується: статичні файли можуть бути розміщені на CDN або у хмарному сховищі, що знімає навантаження з бекенда та зменшує затримки при доставці клієнтського коду. Після цього вся інтерактивність і комунікація відбувається через API, тоді як сервер зосереджується лише на обробці даних, а не на рендерингу інтерфейсу.

У статичних моделях взаємодії клієнта і сервера інтеграція або фактично відсутня, або зведена до мінімально необхідного рівня. У випадку повністю

статичного сайту сервер просто повертає наперед підготовлений HTML-контент, не беручи участі у жодних динамічних процесах під час роботи користувача. Це робить такий підхід максимально простим та швидким, адже сторінки можна кешувати на рівні CDN і доставляти з мінімальними затримками. Проблемою є те, що відсутній механізм оновлення даних без повторної публікації ресурсу, тому модель придатна лише для сторінок, що не потребують зміни або персоналізації.

Другий варіант статичного підходу пов'язаний із розгортанням SPA-застосунків, коли сервер надає користувачу набір статичних файлів, що містять клієнтську логіку (HTML, стилі та JavaScript). Після початкового завантаження саме браузерний застосунок бере на себе всю інтерактивність, а сервер надалі використовується виключно як API-провайдер. Цей сценарій характерний для сучасних розробок на React, Vue, Angular або Blazor WebAssembly. У такій архітектурі сервер зводиться до функції хостингу зібраного фронтенду, а динамічна інтеграція забезпечується вже іншими механізмами, зокрема REST, GraphQL або WebSocket-з'єднаннями. Незважаючи на те, що процес взаємодії фактично спирається на API, з погляду структури доставлений клієнтові контент залишається статичним набором файлів.

Обидва підходи демонструють, що статичний контент можна вважати базовою точкою відліку у порівнянні методів інтеграції. Будь-яке ускладнення взаємодії — наприклад, включення динамічного API або серверного рендерингу — є компромісом між початковою швидкістю завантаження та функціональністю. Традиційний серверний рендеринг (Razor Pages, MVC) можна розглядати як проміжний варіант: він забезпечує динамічне формування HTML і краще підходить для SEO-орієнтованих сторінок або комбінованих рішень, де SPA працює разом із SSR.

У контексті нашої роботи статичний підхід відіграє допоміжну роль. ASP.NET Core здатний одночасно виконувати функції API-сервера і хостити фронтенд, якщо це потрібно. Зібраний клієнтський застосунок можна помістити в каталог `wwwroot` і налаштувати SPA-fallback, щоб сервер коректно повертав `index.html` для будь-якого

маршруту клієнтської навігації. Раніше така інтеграція підтримувалась окремими механізмами SpaServices, але у сучасних версіях .NET вони визнані застарілими, і Microsoft рекомендує розділяти бекенд і фронтенд-проекти. У межах нашого прототипу припускається спрощений варіант: бекенд може роздавати або мінімальну статичну сторінку для демонстрації роботи API, або зовнішній SPA, що взаємодіє через REST-ендпоїнти.

Узагальнюючи, статичні файли є найпродуктивнішим способом первинної доставки контенту, однак не забезпечують інтеграції даних після початкового рендерингу. Саме тому вони застосовуються або в повністю статичних сайтах, або як контейнер для SPA-додатків. У рамках подальших порівнянь (розділ 1.8) цей підхід буде розглянуто як еталон з точки зору простоти та часу завантаження, на тлі якого оцінюються інші, більш інтерактивні методи інтеграції.

1.8 Порівняння методів інтеграції

Розглянувши окремо кожен із популярних підходів та технологій інтеграції, зведемо їх основні характеристики в узагальнену таблицю. Для порівняння вибрано такі критерії:

- Характер обміну даними: як клієнт отримує дані від сервера (запит-відповідь, push, потоковий і т.д.).
- Продуктивність і трафік: відносна швидкодія/ефективність використання мережі.
- Складність розробки: як складно реалізувати і підтримувати даний підхід (включно з доступністю інструментів, налагодження).
- Тип клієнтів: де можна використовувати (браузер, мобільний застосунок, сервер-сервер).

- Підтримка реального часу: чи підтримує пряме push-оновлення від сервера.
- Підтримка кешування: можливість використати стандартне кешування (HTTP) чи потрібні спеціальні рішення.
- Приклади типових сценаріїв: де технологія оптимально підходить.

Таблиця 1.1 – Порівняння методів інтеграції фронкенду і бекенду

Критерій	REST API	GraphQL	SignalR (WebSocket)	gRPC	SPA	Blazor	Статичні файли
Спосіб взаємодії	Запити до множинних URL, фіксовані ресурси (CRUD)	Запит до єдиного endpoint, гнучкий запит із зазначенням полів	Постійне двостороннє з'єднання, сервер пушить повідомлення клієнту	Запити RPC до єдиного сервісного endpoint, виклики методів (можливість потокового передачі)	Немає чіткого «обміну» – клієнтська логіка сама вирішує, коли звертатись до API	Може працювати як SPA (WASM) або через SignalR (Server); у WASM – запити до API за даними, у Server – виклики методів на сервері через SignalR	Одноразова передача контенту (сторінок) при початковому запиті або при навігації на інший статичний URL

Продовження таблиці 1.1

Продуктивність, трафік	Невеликі накладні витрати на кожен запит (HTTP заголовки), текстовий JSON – більший за бінарні формати. Для великих даних може бути повільніше.	Один запит може замінити кілька REST – менше мережових звернень. Але серверна обробка може бути важкою. JSON – передача.	Постійне з'єднання не потребує установлення нових сесій. Повідомлення легкі (можна бінарні). Ефективно для частих дрібних оновлень.	Дуже високий – HTTP/2 мультимплексування, protobuf мінімізує трафік. Низькі затримки. Ефективний при будь-якому обсязі даних, особливо великих.	Після завантаження, робить запити (REST/GraphQL) за потреби. Може оптимізувати трафік (пакетні запити). Початкове завантаження JS може бути важким.	Blazor WASM : значний початковий завантаження runtime, далі трафік як у SPA (виклики API). Blazor Server: трафік невеликими порціями по SignalR, але всі події йдуть через мережу (можливі затримки).	Максимально швидко видача за рахунок CDN/кешування. Дуже малий трафік (лише готовий HTML/зображення). Але при змінах сторінок – кожен раз завантажує весь контент.
------------------------	---	--	---	---	---	---	--

Продовження таблиці 1.1

Складність реалізації	Дуже підтримується: безліч фреймворків, інструментів. Простий для початку. Але потрібне документування версій.	Вище: необхідно підняти GraphQL-сервер або бібліотеку (наприклад, HotChocolate для .NET), написати резолвери. Потрібні знання схеми, мови запитів. Налагодження складніше (GraphQL IDE допомагають).	Середня: у .NET – просте додавання SignalR (кілька рядків). Потрібно писати методи хабів. Нюанси з масштабуванням і відновленням з'єднання.	Висока: треба визначити .proto, налаштувати генерацію коду. Необхідно забезпечити gRPC-Web для браузерів. Інструменти налагодження менш зручні (спеціалізовані утиліти).	Висока на фронтенді: вимагає знання сучасного JS/фреймворку. На бекенді – зазвичай стандартна реалізація API. Розробка фактично двох окремих частин.	Середня: для .NET-розробників – легше, ніж SPA (можна майже не писати JS). Проте треба враховувати обмеження обраної моделі (WASM vs Server). Менш широке ком'юніті, ніж у React/Angular.	Найпростіша: достатньо HTML+CSS. Будь-який сервер здатний роздати файли. Практично не потребує програмування (лише верстка). Обмежена функціональність.
-----------------------	--	--	---	--	--	---	---

Продовження таблиці 1.1

Де використовується (клієнти)	Браузери, мобільні додатки, інші сервери – універсально. Будь-що, що вміє HTTP.	Браузер (через AJAX запити), мобільні бібліотеки), сервери. Підтримка майже як у REST (тільки що клієнт має формувати запит).	Браузер, .NET-клієнти, Java etc. В браузері потрібні можливості WebSocket. Для server-to-server можна теж, але там часто gRPC краще.	Сервер ↔ сервер (ідеально), мобільні (Android/iOS), десктопні. У браузері – лише з gRPC-Web через AJAX (не нативно).	Браузерні веб-додатки. (Мобільні/десктоп через WebView якщо потрібно).	Браузер. (Є Blazor Hybrid для десктопу, але це інший випадок).	Будь-де, де є браузер або навіть без інтернету (статичні сторінки можна зберегти).
Реальний час (push)	Не підтримує нативно (потрібні окремі механізми: WebSocket/SignalR). Клієнт мусить опитувати, щоб отримати оновлення.	Підтримує GraphQL Subscriptions, але це знову ж таки WebSocket під капотом; не так поширено. Основний сценарій – запит по потребі.	Так, спеціально для цього створений – сервер може будь-коли надіслати дані клієнту.	Підтримує streaming: серверний, клієнтський, двосторонній – тобто можна реалізувати push-повідомлення в рамках виклику або постійний потік.	Може реалізувати через використання SignalR або WebSocket в середині самого SPA (SPA – лише архітектура, вона не дає push з коробки, але можна додати).	Blazor Server – так. Blazor WASM – ні, потрібно окремо додавати SignalR або опитування (як у звичайному SPA).	Ні, статичний контент не змінюється сам по собі. Будь-яке оновлення вимагатиме перезавантаження або скриптів (яких в чисто статичному немає).

Кінець таблиці 1.1

Кешування	Повністю підтримує HTTP-кешування (за ETag, Last-Modified, Cache-Control). Можна ефективно кешувати GET-запити на проксі та клієнті.	Стандартне HTTP-кешування складне (запити різні за body, URL, однакові). Потрібні свої стратегії (наприклад, кеш на рівні Apollo Client у браузері).	Не кешується звичайними засобами – це живе з'єднання. Кешувати можна хіба дані, які пересилаються (всередині логіки). Але типового проксі-кешування немає.	Можна кешувати на рівні окремих повідомлень якщо продумати, але стандартно – ні, оскільки все йде через єдине з'єднання. Швидкість з gRPC знижує потребу в кеші в багатьох випадках.	Кешування даних залежить від того, що SPA викликає (якщо REST – тоді може кешувати GET). Самі ж SPA-файли кешуються браузером як звичайна статика.	Статичні файли Blazor (DLL, WASM, JS) можуть агресивно кешуватися (з версіонуванням). Дані, що запитуються – за принципом використаного протоколу (REST/...).	Так, CDN і браузері можуть кешувати контент, поки він не зміниться (Cache-Control). Для оновлення контенту потрібно змінити файл або заголовки.
Типові сценарії	Публічні API, типові CRUD-сервіси, коли важлива простота і сумісність; міжсервісна взаємодія при невисоких вимогах до продуктивності; мобільні додатки, Web-інтерфейси.	Складні запити з різнорідними даними; коли клієнти сильно різні і потрібно уникнути створення багатьох варіантів API; використовується всередині клієнтських застосунків з багатьма варіаціями відображення.	Чат, нотифікації, спільні редактори, будь-що, де потрібне миттєве доставлення подій користувачам. Також ігри, трекінг (напр. відслідковувати місцезонавання/стан). У .NET часто використовуються в адмін-панелях для оновлення даних без перезавантаження.	Високонвантажні мікросервіси (де кожна мілісекунда важлива); внутрішні комунікації в датацентрі; мобільні і настільні клієнти з вимогою швидкого відгуку (стрімінгові сервіси, відеоконференції). Менш підходить для відкритих веб-API.	Великі веб-застосунки: поштові клієнти, соцмережі, де користувацький досвід має бути як у desktop-додатку. Там, де користувач проводить багато часу на одній сторінці, взаємодія є безперервною.	міграція старих WebForms на сучасне; додатки з вимогами до безпеки. Також підходить для внутрішніх інструментів, де швидкість розробки .NET важливіша за ідеальну оптимізацію фронтенду.	Прості веб-сайти, лендінги, блоги, документація. Також коли критично важлива швидкість завантаження першої сторінки – статичний генератор + CDN дають мінімальний час відклику.

Аналіз порівняльної таблиці свідчить, що універсального та абсолютно “найкращого” методу інтеграції не існує — кожен підхід орієнтований на свої сценарії застосування та має специфічні переваги. Раціональний вибір залежить від пріоритетів, вимог і обмежень конкретного проєкту. Узагальнюючи отримані результати, можна зазначити:

- Якщо першочерговим є забезпечення максимальної продуктивності та мінімальних затримок у взаємодії між серверами, саме gRPC демонструє найкращі характеристики.
- У випадках, коли важлива гнучкість формування клієнтських запитів та можливість точно визначати структуру потрібних даних, оптимальним вибором є GraphQL.
- Для досягнення сумісності з найширшим спектром споживачів, простоти реалізації та передбачуваності поведінки найефективнішим залишається REST.
- Для задач, що потребують обміну оновленнями в режимі реального часу, найбільш придатними є механізми на основі WebSocket, зокрема SignalR у .NET-екосистемі.
- Створення насичених, інтерактивних інтерфейсів у браузері доцільно реалізовувати через SPA-фреймворки або технологію Blazor.
- Для сайтів із переважно статичним контентом та високими вимогами до SEO підходять статичні сторінки або традиційні серверні рендерингові моделі.

У контексті даної дипломної роботи основним завданням було розробити прототип веб-застосунку та продемонструвати один з сучасних підходів до інтеграції клієнтської і серверної частин. Враховуючи баланс між складністю впровадження, зрозумілістю архітектури, сумісністю зі сторонніми клієнтами та доступністю інструментів тестування, було прийнято рішення реалізувати інтеграцію на основі REST API. Цей підхід є найбільш практичним для навчального та експериментального

прототипу, оскільки дозволяє чітко продемонструвати принципи побудови API, структурування ресурсів та обробку запитів.

Інші розглянуті технології — GraphQL, gRPC, SignalR — докладно проаналізовано у теоретичному розділі, і вони можуть бути інтегровані в майбутньому, оскільки архітектура розробленого застосунку дозволяє поступове розширення без кардинальної зміни структури. Наприклад, додавання GraphQL-ендпоінта може забезпечити гнучкіші запити для клієнтів, а інтеграція SignalR — можливість надсилати події в реальному часі. Проте впровадження таких механізмів вимагає додаткових ресурсів, часу та зміни архітектури взаємодії, тому у межах даного прототипу було сфокусовано увагу саме на REST як базовому, надійному та добре підтримуваному рішенні.

1.9 Висновки до розділу 1

У даному розділі було здійснено комплексний огляд сучасних підходів до інтеграції клієнтської та серверної частин веб-застосунків. Проаналізовано ключові архітектурні моделі, що формують користувацький досвід, включаючи традиційні багатосторінкові рішення (MPA), високодинамічні односторінкові застосунки (SPA) та інноваційний фреймворк Blazor. Також було досліджено та порівняно технології обміну даними, які забезпечують комунікацію між рівнями, зокрема REST, GraphQL, SignalR і gRPC. Порівняльний аналіз чітко показав, що кожна технологія має власні сильні сторони та оптимальні сфери застосування: REST вирізняється простотою та універсальністю, надаючи широкі можливості кешування та стандартизації взаємодії, тоді як GraphQL забезпечує високу гнучкість запитів, значно зменшуючи проблему надмірної видачі даних (over-fetching). gRPC демонструє найкращу продуктивність у взаємодії сервіс-сервіс завдяки використанню бінарного формату Protocol Buffers та протоколу HTTP/2, SignalR надає критично важливі механізми реального часу, а SPA

та Blazor орієнтовані на створення високоінтерактивного та швидкого інтерфейсу користувача.

Враховуючи поставлені у роботі цілі та наявні обмеження, було обґрунтовано вибір REST API як основного методу інтеграції для подальшої реалізації програмного прототипу. REST надає необхідний баланс між простотою впровадження, високою сумісністю зі сторонніми клієнтами та достатньою функціональністю, що робить його оптимальним рішенням для експериментального сервера і ефективної демонстрації інтеграційного підходу. Обраний підхід забезпечує надійну та легко підтримувану комунікацію, що є пріоритетом на етапі прототипування. При цьому проведений аналіз підтвердив потенційну можливість подальшого розширення системи іншими технологіями — такими як GraphQL чи SignalR — у разі появи додаткових вимог до оптимізації трафіку або реалізації функціоналу в режимі реального часу, забезпечуючи таким чином гнучкість та масштабованість архітектури.

Подальший розділ роботи присвячений розробці програмного прототипу, у якому буде безпосередньо реалізовано обраний REST API, детально описано прийняті архітектурні рішення, роботу основних компонентів та, що є ключовим, представлено результати тестування системи.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ

У цьому розділі представлено процес проєктування та реалізації веб-застосунку IntegraShop — спрощеного інтернет-магазину, створеного для демонстрації інтеграції клієнтських SPA-фронтедів із серверною частиною на основі REST API. На початку визначено постановку задачі та сформульовано функціональні й нефункціональні вимоги, які висуваються до системи. Далі наведено опис архітектури застосунку, зокрема структуру бекенд-модуля, що включає модель даних, механізми доступу до бази даних, систему авторизації та контролери API, а також принципи взаємодії трьох незалежних фронтенд-клієнтів (React, Angular, Vue) з єдиним сервером.

Особливу увагу приділено обґрунтуванню вибору технологій, що були використані в процесі розроблення: середовища ASP.NET Core, бібліотек для роботи з JWT-автентифікацією, СУБД PostgreSQL, ORM-фреймворку Entity Framework Core та сучасних SPA-інструментів. У відповідних підрозділах детально подано концепцію проєктування, логіку реалізації ключових компонентів, а також фрагменти коду, які ілюструють роботу окремих механізмів системи.

На завершення розділу розглянуто процес тестування створеного прототипу, оцінено якість виконаних інтеграційних рішень та визначено, наскільки реалізація відповідає заданим вимогам і демонструє практичну коректність застосованого підходу.

2.1 Постановка задачі

Для демонстрації практичного застосування різних методів інтеграції клієнтських застосунків із серверною частиною було вирішено реалізувати

спрощений веб-застосунок інтернет-магазину IntegraShop. Така тематика є інтуїтивно зрозумілою, містить природну структуру даних та охоплює достатню кількість типових сценаріїв взаємодії між фронтендом і бекендом. Це робить її зручним прикладом для аналізу інтеграційних підходів. У межах системи реалізовано роботу з товарами, користувачами та замовленнями — трьома центральними сутностями, які забезпечують повноцінний цикл функціонування спрощеної e-commerce платформи.

Основними бізнес-об'єктами виступають товари (Product), зареєстровані користувачі (User) та замовлення (Order). Кожен товар містить у собі унікальний ідентифікатор, назву, опис, ціну, категорію, зображення та інші супровідні дані. Користувачі проходять реєстрацію та автентифікацію за допомогою JWT-токенів, що дозволяє обмежувати доступ до окремих операцій. Адміністратор отримує розширені можливості керування товарами, а звичайні користувачі можуть переглядати каталог та створювати замовлення. Таким чином, система охоплює прикладні сценарії як безавторизаційної взаємодії, так і доступу до захищених маршрутизованих ресурсів.

Функціональні вимоги до системи передбачають реалізацію REST API, яке має забезпечувати:

- отримання списку всіх товарів (HTTP GET /api/products);
- отримання детальної інформації про товар (GET /api/products/{id});
- створення нового товару (POST /api/products, лише для адміністратора);
- редагування та видалення товарів (PUT/DELETE /api/products/{id}, тільки для адміністратора);
- реєстрацію та авторизацію користувачів (POST /api/auth/register, /login);
- створення замовлення (POST /api/orders, лише для автентифікованих користувачів);
- отримання списку замовлень та перегляд їх деталей (GET /api/orders/, /{id}).

Всі відповіді сервера повертаються у форматі JSON, а також супроводжуються коректними HTTP-кодами: 200 OK, 201 Created, 400 Bad Request, 401 Unauthorized,

403 Forbidden, 404 Not Found тощо. API повинно повністю підтримувати CRUD-операції над товарами та частково над замовленнями (створення та перегляд). Для авторизації застосовується механізм JWT, який інтегрується в усі захищені запити.

Вимоги до клієнтської частини мінімальні: три незалежні SPA-застосунки (React, Angular і Vue) повинні виконувати однакові запити до API та коректно обробляти відповіді. Інтерфейс не є критичним у межах роботи, оскільки основна увага приділяється саме інтеграції, а не дизайну. Тому реалізація кожного фронтенда може бути спрощеною, але функціонально повною.

Нефункціональні вимоги зосереджено на коректності інтеграції та структурованості коду. Оскільки прототип не призначений для високих навантажень, вимоги до продуктивності були мінімальними, однак система повинна забезпечувати стабільну роботу при типовій кількості запитів. Важливо також дотримуватися принципів REST: логічна структура ресурсів, ідемпотентність запитів PUT/DELETE, зрозумілі URI, однозначні статус-коди та чіткі контракти даних. Веб-API має бути повністю документованим за допомогою Swagger/OpenAPI, що спрощує взаємодію фронтенд-клієнтів із сервером.

Таким чином, постановка задачі визначила створення функціонального прототипу інтернет-магазину, який демонструє практичну реалізацію REST-орієнтованої інтеграції. Подальші підрозділи розділу присвячено опису архітектури, вибору технологічного стеку та реалізації ядра системи IntegraShop.

2.2 Архітектура системи

Загальна архітектура застосунку IntegraShop побудована за принципами трирівневої моделі веб-системи, яка включає рівень збереження даних, рівень бізнес-логіки та рівень презентації, що представлений REST API. Такий підхід забезпечує

чітке розділення відповідальності між компонентами, гнучкість у масштабуванні та можливість взаємодії з кількома клієнтськими застосунками одночасно.

На рівні даних використовується реляційна база PostgreSQL, у якій створено кілька основних таблиць: Users, Products, Orders, OrderItems. Взаємозв'язки між цими сутностями реалізовано через Entity Framework Core, що виконує роль ORM і дозволяє працювати з даними у вигляді C#-класів. Модель даних включає властивості, необхідні для повноцінної роботи інтернет-магазину, такі як атрибути товару, інформацію про користувача, структуру замовлення та склад його позицій.

Рівень бізнес-логіки реалізовано через набір сервісів, які інкапсулюють роботу з відповідними сутностями. Наприклад, ProductService відповідає за створення, редагування та видалення товарів; OrderService — за формування та обробку замовлень; AuthService — за реєстрацію, логін та формування JWT-токенів. Така організація дозволяє централізовано перевіряти вхідні дані, застосовувати правила доступу (наприклад, обмеження адміністративних операцій), а також забезпечує можливість підміни сервісів під час тестування або розширення системи. Усі сервіси реєструються в контейнері залежностей ASP.NET Core як scoped-служби, що дозволяє автоматизувати їхнє створення та життєвий цикл.

На верхньому рівні розміщено REST API, реалізоване за допомогою контролерів ASP.NET Core. Кожен контролер відповідає за роботу окремої групи ресурсів: ProductsController, OrdersController, AuthController, UsersController. Контролери приймають HTTP-запити, викликають відповідні методи сервісів та повертають відповіді у форматі JSON. Атрибут [ApiController] забезпечує автоматичну валідацію моделей даних, а також формує стандартизовані повідомлення про помилки. Маршрутизація API побудована за REST-принципами, наприклад /api/products, /api/orders, /api/auth/login, що забезпечує логічність і передбачуваність інтерфейсу.

Схема взаємодії основних компонентів у типових сценаріях така:

1. Клієнтський застосунок (React, Angular або Vue) надсилає запит, наприклад GET /api/products або POST /api/orders.
2. Контролер на сервері приймає запит, перевіряє JWT-токен (для захищених операцій) і викликає відповідний метод бізнес-сервісу.
3. Сервіс виконує бізнес-логіку: звертається до DbContext, керує транзакціями, перевіряє наявність записів, формує замовлення тощо.
4. Після обробки запиту сервіс повертає результат контролеру.
5. Контролер формує HTTP-відповідь з кодом стану (200, 201, 400, 401, 404 тощо), а ASP.NET Core автоматично серіалізує модель у формат JSON.
6. Клієнтський SPA-застосунок отримує результат та відображає його користувачу.

Для роботи з помилками використано middleware для глобальної обробки винятків, яке перехоплює непередбачені ситуації, формує єдиний формат відповіді про помилку та запобігає витоку внутрішньої інформації. Це підвищує безпеку та стабільність застосунку.

Клієнтська частина у цьому проєкті представлена трьома незалежними SPA-застосунками, створеними з використанням React, Angular та Vue. Усі вони працюють з одним і тим самим REST API, що дозволяє продемонструвати універсальність серверної частини та порівняти особливості інтеграції в різних фреймворках. Функціональність фронтенду включає перегляд списку товарів, роботу з деталями продукту, реєстрацію та логін, управління товарами (для адміністратора), а також оформлення замовлень.

Документація API також є важливою частиною архітектури. Для її генерації використано Swagger/OpenAPI, який автоматично будує інтерфейс на основі контролерів та моделей. Це дозволяє тестувати API без створення окремого клієнта, а також допомагає розробникам SPA-застосунків швидко зрозуміти структуру ендпоінтів.

На рисунку 2.1 наведено спрощену схему архітектури IntegraShop, яка демонструє ключові компоненти: клієнтські застосунки, REST API контролери, бізнес-сервіси та рівень доступу до даних на основі PostgreSQL.

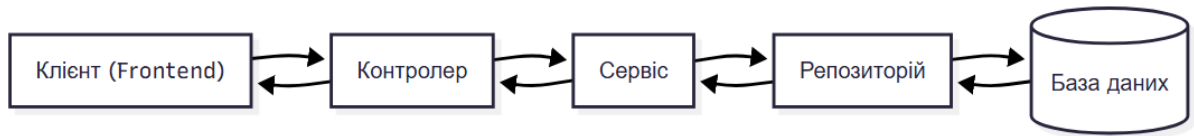


Рисунок 2.1 – Спрощена архітектура застосунку: взаємодія клієнта, контролера, сервісу та бази даних

Показана схема узагальнює логіку роботи всієї системи та демонструє чітке розділення відповідальностей між компонентами. Завдяки такій архітектурі IntegraShop може стабільно взаємодіяти з кількома фронтенд-клієнтами одночасно, забезпечуючи передбачувану поведінку незалежно від використаного фреймворку. Представлена модель є високорівневим оглядом структури системи, тоді як більш детальну архітектуру та конкретні реалізаційні аспекти програмного рішення наведено у розділі 3.1, де подано розширені схеми взаємодії компонентів та внутрішню організацію серверної частини.

2.3 Вибір технологій та засобів реалізації

Вибір технологічного стеку для реалізації системи IntegraShop ґрунтувався як на вимогах до функціональності, так і на попередньому досвіді автора, що дало змогу оптимізувати процес розробки та забезпечити високу якість сервісної частини. Основою серверної логіки обрано платформу ASP.NET Core, оскільки вона поєднує продуктивність, модульність та зручні вбудовані механізми для реалізації RESTful API. Платформа надає готові шаблони проєктів, підтримку середовищ

конфігурації, built-in dependency injection, гнучку маршрутизацію та високу швидкодію під час обробки HTTP-запитів, що робить її оптимальною для побудови веб-застосунків із розділеною клієнтсько-серверною архітектурою. Альтернативи на кшталт Node.js/Express або Python/Flask також розглядалися, але їх застосування потребувало б суттєвих додаткових витрат часу на налаштування та адаптацію під вимоги системи.

Для збереження даних обрана реляційна база даних PostgreSQL, яка забезпечує надійність, розширюваність та відповідність сучасним вимогам до обробки даних у веб-системах. PostgreSQL підтримує складні транзакції, індексацію, розширення типів даних і добре масштабується, що робить її придатною не лише для прототипу, а й для подальшого використання в реальній комерційній розробці. Інтеграція з ASP.NET Core здійснюється через офіційний провайдер Npgsql, що забезпечує сумісність зі стандартними механізмами .NET.

На рівні взаємодії з базою даних використано ORM-фреймворк Entity Framework Core, який значно скорочує кількість рутинного SQL-коду та підвищує прозорість роботи з моделями. EF Core дає можливість використовувати міграції для автоматичного створення та модифікації структури таблиць, працювати з LINQ-запитами та легко змінювати СУБД у майбутньому без зміни бізнес-логіки. У контексті IntegraShop через EF Core реалізовано всі основні моделі домену — товари, категорії, кошик, замовлення, користувачі та ролі.

Форматом обміну даними між сервером та клієнтами використано JSON, що є стандартом де-факто для REST API. ASP.NET Core забезпечує швидку серіалізацію та десеріалізацію даних через бібліотеку System.Text.Json, а фронтенд-застосунки (React, Angular та Vue) мають нативну підтримку роботи з JSON через fetch, Axios та інші інструменти[13].

Особливе місце в системі займає питання автентифікації та авторизації, без яких неможливе створення безпечного інтернет-магазину. Для цього використано механізм JWT (JSON Web Token), який дозволяє клієнтам отримувати маркер доступу після

успішної авторизації. Токен містить зашифровану інформацію про користувача (ID, ім'я, роль), що передається у заголовок `Authorization`. Обмеження доступу до API реалізовано через атрибути `[Authorize]` та ролеві політики (`[Authorize(Roles = "Admin")]`), що забезпечує сегрегацію функціоналу між звичайними користувачами та адміністраторами. Такий підхід дозволяє безпечно виконувати критичні операції — оформлення замовлень, керування товарами, зміну облікових даних тощо.

Для документування та інтерактивного тестування API інтегровано Swagger, який автоматично будує OpenAPI-специфікацію. Swagger UI дає змогу виконувати всі запити до API прямо з браузера, включно із захищеними ендпоінтами після введення JWT-токена в режимі "Authorize". Це значно пришвидшило налагодження API та стало зручним інструментом для тестування фронтенд-клієнтів.

У процесі розробки використано середовище Visual Studio 2022 з налагодженням у режимі Debug. Додатково застосовувався Postman для перевірки різних сценаріїв взаємодії API з точки зору типових, граничних і помилкових запитів.

Підсумовуючи, обраний технологічний стек — ASP.NET Core, Entity Framework Core та PostgreSQL у поєднанні з JWT-механізмом безпеки й Swagger-документацією — забезпечив оптимальний баланс між продуктивністю, простотою, безпекою та масштабованістю. Це дало можливість реалізувати повноцінну бекенд-частину інтернет-магазину та створити універсальний REST API, який однаково успішно використовується трьома незалежними SPA-клієнтами. У наступному підрозділі наведено опис програмної реалізації ключових компонентів системи.

2.4 Розроблення REST API сервісу

Реалізація серверної частини IntegraShop базується на принципах REST-архітектури, що дозволило створити передбачуваний, модульний та інтуїтивно зрозумілий API для взаємодії з клієнтськими застосунками. У процесі розроблення

було дотримано вимог, окреслених у постановці задачі, а структура проєкту організована таким чином, щоб забезпечити чітке відокремлення відповідальностей між рівнями застосунку та можливість подальшого розширення системи без порушення вже реалізованої функціональності.

Усі функціональні можливості бекенду згруповані у вигляді тематичних контролерів, кожен з яких обслуговує конкретний ресурс системи — користувачів, товари, замовлення, кошик, авторизаційні операції тощо. Кожен контролер реалізує повний набір CRUD-операцій над відповідною сутністю, дотримуючись REST-підходу як у структурі URL, так і у використанні HTTP-методів і кодів стану. Це забезпечило однорідний стиль API й дозволило фронтед-клієнтам легко інтегруватися із сервером незалежно від технологічного стеку — React, Angular чи Vue працюють з однаковими ендпоінтами без необхідності адаптації.

Комунікація між контролерами та даними здійснюється через сервісний шар застосунку. Сервіси реалізують основну бізнес-логіку: саме вони відповідають за перевірку коректності даних, обробку складних сценаріїв (наприклад, створення замовлення з одночасним оновленням стану кошика), формування структурованих результатів та взаємодію з Entity Framework Core. Використання сервісів дозволило мінімізувати кількість логіки у контролерах — вони фактично перетворилися на «тонкі» елементи архітектури, чия основна функція полягає у маршрутизації запитів і поверненні клієнту відповідей у стандартизованому форматі. Такий підхід відповідає принципам Clean Architecture і забезпечує легкість тестування та підтримки системи.

Важливою частиною реалізації стала побудова моделі даних та відокремлення внутрішніх сутностей від об'єктів, які надходять від клієнта або повертаються у відповідь. Для цього застосовано розділення на DTO (Data Transfer Object) та ентиті. DTO дозволяють суворо контролювати набір полів, які клієнт може передавати, а також забезпечують коректну валідацію даних на рівні ASP.NET Core ще до того, як запит досягне бізнес-логіки. Наявність таких механізмів суттєво зменшує ймовірність помилок та гарантує єдину поведінку API незалежно від того, який клієнт ініціює

запит. Ентиті, у свою чергу, відображають структуру бази даних та оперують лише внутрішніми даними — це робить архітектуру гнучкою і стійкою до змін у схемі бази.

Інтеграція з PostgreSQL здійснюється за допомогою Entity Framework Core. Контекст бази даних інкапсулює набір таблиць та їхніх зв'язків, а також надає інструменти для виконання запитів, створення, оновлення й видалення даних. Механізм міграцій EF Core дав змогу формувати структуру бази у автоматизований спосіб та уникнути ручного написання SQL-коду на етапі розроблення. Така взаємодія із СУБД є не лише надійною, але й масштабованою — у разі потреби застосунок може бути перенесений на іншу СУБД без значного переписування коду.

Не менш важливою складовою серверної частини є механізми автентифікації та авторизації. Враховуючи, що IntegraShop є інтернет-магазином, доступ до багатьох операцій — наприклад, створення чи редагування товарів, оформлення замовлення, робота з кошиком — має бути обмежений автентифікованими користувачами. Для цього впроваджено сучасний механізм авторизації на основі JWT-токенів. Після успішного входу користувач отримує токен, який містить інформацію про його роль і суттєві атрибути, а всі подальші запити до захищених ендпоінтів супроводжуються надсиланням цього токена у стандартному заголовку Authorization. Система самостійно розпізнає роль користувача та дозволяє або забороняє виконання певних операцій. Завдяки цьому, роботу з товарами можуть виконувати лише адміністратори, тоді як пересічні користувачі мають доступ лише до перегляду інформації та оформлення замовлень.

Коректна обробка помилок є одним із ключових аспектів якісного REST API. У процесі реалізації передбачено стандартний набір відповідей сервера для типової поведінки: повідомлення про відсутність ресурсу, помилки валідації, відсутність авторизації або прав доступу, а також внутрішні помилки, які перехоплюються глобальним middleware. Така стандартизована система обробки помилок дозволяє фронтенд-клієнтам однаково трактувати поведінку API, а отже — забезпечує стабільність усієї системи та полегшує інтеграцію різних клієнтських платформ.

Особливу увагу під час реалізації серверної частини приділено забезпеченню сумісності API з різними SPA-фреймворками. Тестування взаємодії з React-, Angular- та Vue-клієнтами підтвердило, що API є достатньо універсальним, аби працювати однаково ефективно з будь-яким фронтом, незалежно від підходу до стану, маршрутизації або архітектури компонентів. Це ще раз підкреслює гнучкість і продуманість архітектури IntegraShop, яка свідомо проєктувалася таким чином, щоб не прив'язуватися до конкретного інструменту на клієнтському рівні.

Усі основні сценарії використання були ретельно протестовані — від базових CRUD-операцій над товарами до складних бізнес-процесів, таких як оформлення замовлення із перевіркою користувача та валідацією даних. Система демонструє стабільність, передбачуваність і повну відповідність вимогам, що висуваються до REST API. Застосунок успішно витримує паралельні запити, коректно обробляє одночасні звернення до бази даних та повертає клієнтам структуровані та валідні відповіді.

У підсумку, реалізація серверної частини IntegraShop продемонструвала можливість створення повноцінного, масштабованого та безпечного REST API, який може бути використаний у реальному веб-застосунку. Архітектура бекенду дозволяє легко розширювати систему, впроваджувати нові модулі та експериментальні методи інтеграції (GraphQL, gRPC, SignalR), не порушуючи цілісності наявних компонентів. Система відповідає сучасним вимогам веб-розроблення та демонструє якісну інтеграцію клієнтської і серверної частин за допомогою технологій ASP.NET Core.

2.5 Тестування та результати

Після завершення реалізації серверної частини було проведено комплексне тестування з метою перевірки працездатності побудованого REST API, коректності взаємодії між компонентами системи та відповідності результатів поставленим

вимогам. Тестування охоплювало як функціональні аспекти роботи застосунку, так і елементи поведінки системи в умовах помилкових чи граничних сценаріїв. Окрему увагу приділено перевірці інтеграції клієнтських частин із бекендом, оскільки саме цей аспект є ключовим для досліджуваної теми.

Першим етапом перевірки стала ручна оцінка роботи API через Swagger UI — інтерактивну документаційну панель, яка автоматично генерується з контролерів і моделей ASP.NET Core. Через Swagger було послідовно протестовано всі основні ендпоінти: операції над товарами, авторизація користувачів, робота з кошиком та створення замовлень. Веб-інтерфейс дозволив виконувати запити в реальному часі та спостерігати за їхніми результатами, що є особливо важливим для оцінки структури JSON-відповідей, кодів стану HTTP, поведінки в умовах відсутності авторизації чи передавання некоректних даних. Swagger підтвердив коректність опису моделей, автоматичне застосування обмежень валідації та узгодженість роботи API.

Для більш детальної перевірки взаємодії з сервером було використано інструменти Postman і командний рядок (через curl). Ці засоби дали можливість вручну формувати різні сценарії: надсилати запити без необхідних полів, отримувати дані за неіснуючими ідентифікаторами, тестувати авторизовану й неавторизовану взаємодію, перевіряти поведінку API при повторюваних діях (наприклад, повторному видаленні ресурсу). У всіх цих випадках сервер передбачувано повертав стандартизовані коди стану HTTP, що підтвердило коректність реалізації REST-концепції.

Крім ручних перевірок, частина логіки сервісного шару була протестована за допомогою модульних тестів на основі xUnit та InMemory-провайдера Entity Framework Core. Такий підхід дозволив виконувати перевірки без запуску веб-сервера, ізолюючи роботу сервісів та зосереджуючись на їхній внутрішній логіці. Модульні тести підтвердили, що сервіси правильно працюють з даними, коректно обробляють неуспішні сценарії та надійно взаємодіють з контекстом бази.

Окремо було перевірено поведінку серверної частини під невеликим навантаженням. Послідовне виконання серій GET-та POST-запитів показало стабільний час відповіді та відсутність деградації при повторюваних зверненнях до бази даних. Хоча прототип не передбачав повноцінного навантажувального тестування, отримані результати продемонстрували достатню продуктивність для цілей інтеграційної моделі.

Тестування інтеграції з клієнтськими застосунками підтвердило універсальність API: усі три фронтенд-застосунки (React, Angular, Vue) успішно здійснювали запити до сервера та відображали отримані дані. Це означає, що API побудований достатньо незалежно від конкретної технології на клієнті, а всі необхідні механізми — авторизація, передавання JSON-даних, формування запитів — працюють коректно та однаково для різних SPA-фреймворків.

Проведене тестування також показало, що система коректно обробляє помилкові дані. Наприклад, запити з пропущеними обов'язковими полями автоматично відхилялися механізмами валідації ASP.NET Core, а спроби доступу до захищених ендпоїнтів без токена завершувалися поверненням коду 401 Unauthorized. Ця передбачуваність поведінки є важливою характеристикою REST API, оскільки дозволяє клієнтським застосункам легко інтерпретувати результати запитів і впроваджувати відповідні сценарії обробки.

У підсумку, тестування створеного прототипу підтвердило правильність реалізації архітектурних рішень, відповідність API поставленим функціональним та нефункціональним вимогам і демонструє можливість подальшого масштабування системи. Застосунок продемонстрував стабільну роботу, коректність інтеграції з базою даних PostgreSQL та повну сумісність з трьома різними SPA-клієнтами. Отримані результати показують, що побудований REST API може слугувати основою для розширення функціональності, впровадження альтернативних методів інтеграції (GraphQL, SignalR, gRPC) та еволюції застосунку в більш комплексну систему.

2.6 Висновки до розділу 2

У другому розділі було представлено процес проектування та реалізації серверної частини веб-застосунку, що слугує прототипом інтеграції клієнта і сервера за допомогою REST API. На основі поставлених вимог обґрунтовано вибір технологічного стеку: ASP.NET Core як платформи для створення веб-сервісів, PostgreSQL як надійної реляційної СУБД та Entity Framework Core як ORM-інструмента, що забезпечує ефективну взаємодію з базою даних. Обраний стек дав змогу сформувати гнучку та прозору архітектуру.

У межах розділу було розроблено структуру бекенд-системи, що включає модель даних, шар доступу до бази, бізнес-логіку та API-контролери, організовані відповідно до принципів REST. Реалізовані ендпоїнти підтримують повний набір CRUD-операцій, а використання DTO-класів підвищило контроль над передаванням даних і забезпечило зручність інтеграції з клієнтами.

Тестування прототипу підтвердило його працездатність: сервер коректно обробляє валідні та невалідні запити, повертає передбачувані HTTP-коди та стабільно взаємодіє з базою даних. Отримані результати показали, що REST-підхід є ефективним для інтеграції з різними клієнтськими застосунками, а структура системи дозволяє без значних змін розширювати її можливості.

Таким чином, реалізований прототип успішно демонструє обраний метод інтеграції та може слугувати основою для подальшого розвитку, включаючи впровадження альтернативних технологій взаємодії та розширення функціональності застосунку.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

Розділ присвячений опису структури, архітектури та програмної реалізації застосунку, створеного в рамках дипломної роботи для демонстрації інтеграції клієнтських інтерфейсів із серверною частиною за допомогою REST API. Реалізація охоплює серверну платформу ASP.NET Core, реляційну базу даних PostgreSQL, а також три незалежні клієнтські SPA-застосунки, розроблені з використанням фреймворків React, Angular та Vue. Основною метою цього розділу є послідовне пояснення внутрішньої організації системи, способів взаємодії її компонентів та особливостей реалізації архітектурних рішень.

3.1 Загальна архітектура системи

Загальна структура була побудована як класична багаторівнева система з чітким розмежуванням відповідальностей між рівнями. На верхньому рівні знаходяться три незалежні клієнтські додатки (React, Angular та Vue), кожен з яких забезпечує взаємодію користувача з інтерфейсом, але не містить логіки доступу до даних. Клієнтські застосунки надсилають HTTP-запити до єдиного серверного API, що гарантує однакову поведінку всіх інтерфейсів незалежно від використовуваного фреймворку чи платформи.

Серверна частина побудована на платформі ASP.NET Core Web API та реалізує повноцінну бізнес-логіку системи: обробку операцій над товарами, замовленнями, обліковими записами та автентифікацію користувачів. Дані зберігаються в PostgreSQL, а доступ до них здійснюється через ORM-бібліотеку Entity Framework Core, що забезпечує зручність роботи з моделями та гарантує узгодженість даних.

Взаємодію компонентів представлено на рисунку нижче.

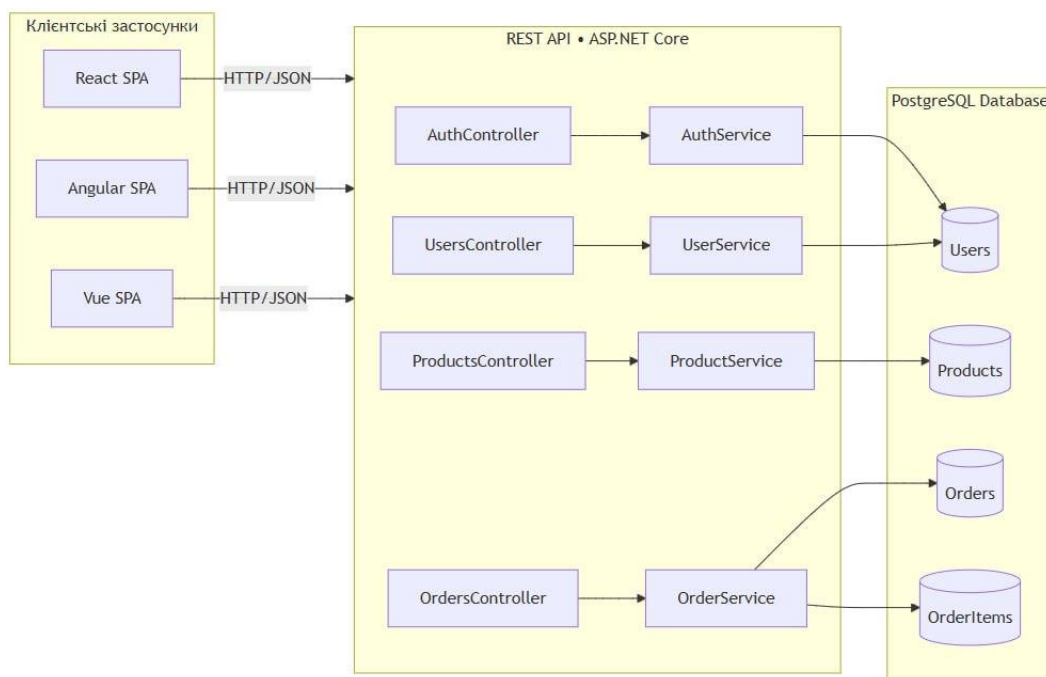


Рисунок 3.1 – Загальна архітектура застосунку

На рисунку наведено загальну архітектуру системи, яка демонструє основні компоненти веб-застосунку та взаємодію між ними. Вона відображає послідовність проходження запиту від клієнтського інтерфейсу до серверної частини та бази даних, а також повернення відповіді користувачу. Така схема дозволяє узагальнено побачити структуру програмного рішення, зрозуміти розподіл відповідальностей між рівнями системи та слугує основою для подальшого деталізованого проєктування окремих компонентів.

3.2 Архітектура бекенд-системи

Бекенд побудовано як модульну систему, у якій кожен компонент виконує чітко визначену роль. На рівні прийому запитів працюють контролери — публічні точки доступу REST API. Вони відповідають за маршрутизацію запитів, виклик відповідних сервісів та формування HTTP-відповідей.

Наступним рівнем є сервісна логіка — об'єкти, що інкапсулюють бізнес-процеси інтеграції. Сервіси не взаємодіють безпосередньо з HTTP-контекстом, тому їх легко тестувати та перевикористовувати. Саме в сервісах зосереджена основна функціональність: оброблення операцій створення замовлення, оновлення товару, обробки логіну користувача, перевірки прав доступу тощо.

Рівень збереження даних представлений контекстом БД (ApplicationDbContext) та наборами моделей, що відповідають сутностям у PostgreSQL. Контекст відповідає за всі операції читання та запису, а також забезпечує автоматичне створення схеми БД через застосування міграцій.

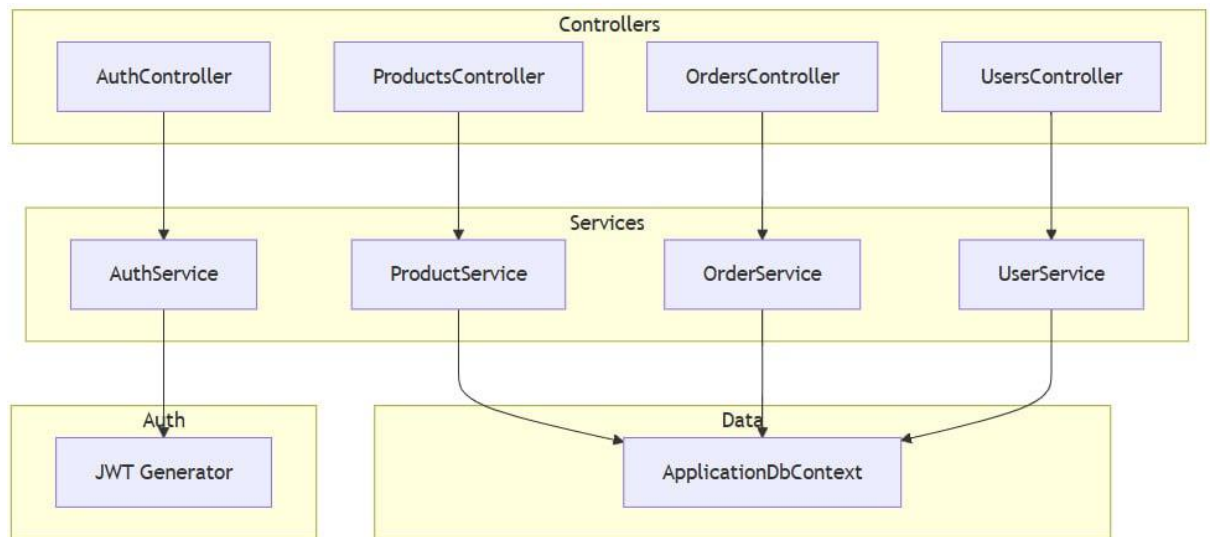


Рисунок 3.2 – Архітектура серверної частини

Представлена на рисунку архітектура бекенд-системи відображає внутрішню побудову серверної частини застосунку. Тут показано, як організовано шари доступу до даних, бізнес-логіки та API-контролерів, а також яким чином вони взаємодіють між собою. Така структуризація дає змогу відокремити обробку HTTP-запитів від роботи з базою даних і бізнес-правилами, що підвищує гнучкість, супроводжуваність і масштабованість системи. Дана схема є концептуальною основою для реалізації REST API та подальшого розширення функціональності.

3.3 Модель даних

Система використовує чотири основні сутності: користувачів, товари, замовлення та позиції замовлення. Усі вони зв'язані між собою відповідно до логіки роботи інтернет-магазину. Користувач може мати багато замовлень, кожне замовлення складається з декількох позицій, а кожна позиція пов'язана з конкретним товаром.

Завдяки Entity Framework Core ці зв'язки реалізовані у вигляді навігаційних властивостей, що робить структуру моделей прозорою та природною з точки зору об'єктно-орієнтованого проєктування.

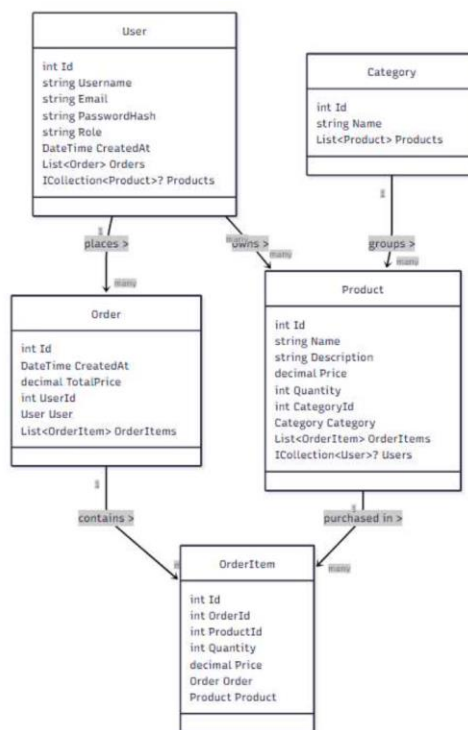


Рисунок 3.3 – Діаграма класів моделі даних

Рисунок ілюструє модель даних системи, зокрема основні сутності предметної області (товари, користувачі, замовлення, позиції замовлення) та зв'язки між ними. Така схема відображає логічну організацію інформації, яка зберігається у базі даних,

і забезпечує цілісність та узгодженість даних на рівні всієї системи. Модель даних є основою для побудови структури таблиць у PostgreSQL та налаштування ORM-засобів, а також визначає, як бізнес-логіка працює з об'єктами доменної області під час обробки клієнтських запитів.

3.4 Алгоритм обробки клієнтських запитів

Однією з найважливіших частин інтеграції є механізм обробки REST-запитів. На прикладі запиту отримання списку товарів розглянуто типову взаємодію: клієнт надсилає HTTP-запит, контролер передає його сервісу, сервіс звертається до бази даних через контекст, отримані дані повертаються назад у вигляді JSON-відповіді.

Ця послідовність відображена на наступному рисунку.

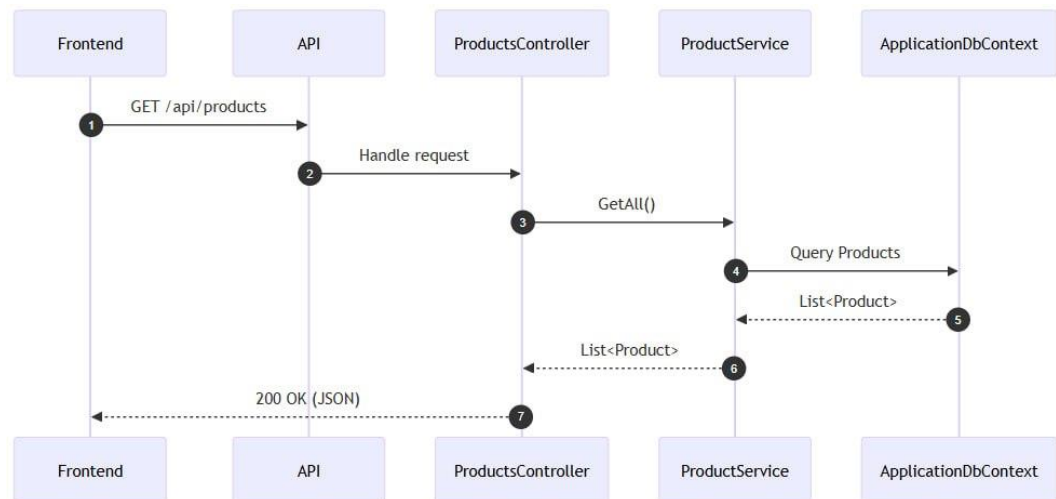


Рисунок 3.4 – Послідовність виконання запиту GET /api/products

На рисунку наведено узагальнений алгоритм обробки клієнтських запитів у системі IntegraShop. Він демонструє, у якій послідовності виконується приймання HTTP-запиту, автентифікація користувача (за наявності), виклик відповідного методу контролера, звернення до бізнес-логіки та рівня доступу до даних, формування

відповіді та її повернення клієнту. Така схема дозволяє чітко простежити життєвий цикл запиту й оцінити коректність побудови обробки як з точки зору архітектури, так і з точки зору дотримання принципів REST.

3.5 Взаємодія фронтенд-застосунків з REST API

Проект IntegraShop спеціально побудований так, щоб одночасно підтримувати кілька клієнтів на різних фреймворках. Незалежно від інструментів — Axios, Angular HttpClient чи Vue Axios — усі клієнти отримують дані через спільний REST API.

Це підкреслює універсальність API і його здатність обслуговувати різні типи клієнтів без модифікації бекенду.

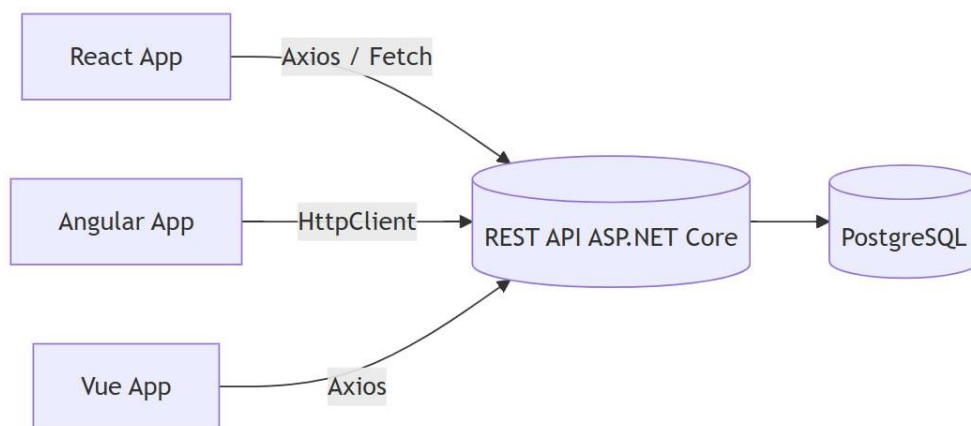


Рисунок 3.5 – Інтеграція React, Angular та Vue з REST API

Рисунок відображає взаємодію трьох фронтенд-застосунків (на React, Angular та Vue) з єдиним REST API серверної частини. Показано, що всі клієнти здійснюють запити до однакових ендпоїнтів, використовують спільний формат даних і отримують відповіді з однієї backend-системи. Такий підхід демонструє універсальність і повторне використання серверної логіки, спрощує підтримку проекту та створює

можливість подальшого підключення нових клієнтів без зміни базової архітектури бекенду.

3.6 Висновки до розділу 3

У цьому розділі було проаналізовано архітектуру веб-застосунку, охарактеризовано його основні структурні компоненти та механізми взаємодії. Наведені діаграми дозволили послідовно розкрити логіку побудови системи: від загальної трирівневої архітектури до внутрішньої організації бекенду, структури моделей даних та алгоритму обробки HTTP-запитів.

Розгляд моделі даних та основних сутностей підтвердив цілісність і коректність організації інформації, а опис процесу обробки запитів показав відповідність реалізації принципам REST. Окремо було продемонстровано, як три різні фронтенд-застосунки (React, Angular, Vue) взаємодіють із єдиним REST API, що підтверджує універсальність обраної архітектури та придатність системи до масштабування.

У підсумку можна стверджувати, що структура IntegraShop є продуманою, узгодженою та технічно обґрунтованою. Її організація забезпечує стабільність роботи, легкість розширення та можливість інтеграції з будь-якими сучасними клієнтськими інтерфейсами.

4 ВСТАНОВЛЕННЯ, ЗАПУСК І ВИКОРИСТАННЯ ПРОГРАМНОГО ЗАСТОСУНКУ

Цей розділ присвячений практичним аспектам розгортання, налаштування та використання розробленого програмного забезпечення. Тут наведено вимоги до апаратного та програмного забезпечення, процес інсталяції серверної та клієнтської частин, а також демонстрацію основного функціоналу системи. Метою розділу є підтвердження працездатності реалізованої архітектури, коректності інтеграції фронтенду з бекендом і готовності застосунку до подальшого використання або розгортання у виробничому середовищі.

4.1 Вимоги до апаратного та програмного середовища

Для стабільної роботи системи необхідні такі компоненти:

1. Операційна система: Windows 10/11 або Linux (Ubuntu 20.04+).
2. Для роботи серверної частини:
 - .NET SDK 8.0;
 - ASP.NET Core Runtime;
 - PostgreSQL 14+;
 - встановлений драйвер Npgsql та доступ до БД.
3. Для роботи клієнтської частини:
 - Node.js 18 або новіше;
 - npm / yarn / pnpm;
 - сучасний браузер.

Інструменти розробки: Visual Studio, Postman або Swagger UI для тестування.

Такі вимоги забезпечують можливість запуску проєкту практично на будь-якому сучасному персональному комп'ютері чи сервері.

4.2 Структура програмної системи

Програмний комплекс складається з кількох незалежних модулів:

1. Серверна частина (REST API) — ASP.NET Core + Entity Framework Core + PostgreSQL.
2. Клієнтські SPA-додатки:
 - React;
 - Vue;
 - Angular.
3. Документаційний модуль — Swagger/OpenAPI, автоматично доступний після запуску серверної частини.

Архітектура з окремим бекендом і трьома незалежними фронтендами демонструє універсальність API та можливість підключення будь-якого клієнта, який підтримує HTTP-взаємодію.

4.3 Процедура інсталяції та налаштування

У цьому підрозділі описано послідовність дій, необхідних для підготовки програмної системи до роботи. Наведено основні кроки встановлення, налаштування та запуску серверної частини та клієнтських застосунків. Спочатку розглядається конфігурація бази даних, після чого переходять до запуску API-сервера та фронтенд-застосунків, що взаємодіють з ним.

4.3.1 Налаштування бази даних

Для роботи серверної частини необхідно попередньо встановити систему керування базами даних PostgreSQL. Після інсталяції створюється нова база даних, яка використовуватиметься застосунком. На наступному етапі у конфігураційних файлах серверного модуля задається параметри підключення до цієї бази, зокрема назва БД, адреса сервера, ім'я користувача та пароль. Після внесення цих параметрів застосунок автоматично застосовує необхідні міграції та створює початкову структуру таблиць при першому запуску.

4.3.2 Запуск серверної частини

Серверна частина запускається як стандартний ASP.NET Core застосунок. Після запуску сервер автоматично відкриває доступ до REST API за вказаною адресою. Одночасно з цим активується автоматично згенерований інтерфейс Swagger/OpenAPI, що дозволяє переглядати всі доступні кінцеві точки сервісу та виконувати тестові запити без додаткових інструментів. Якщо налаштування підключення до бази даних виконано коректно, застосунок починає працювати у штатному режимі.

4.3.3 Запуск фронтендів

Кожен із клієнтських модулів (React, Vue, Angular) функціонує як окремий SPA-застосунок. Перед запуском виконується встановлення необхідних залежностей, після чого фронтенд може бути розгорнутий у режимі розробки. У конфігураційних файлах кожного модуля вказується адреса серверного API, до якого він має підключатися. Після запуску клієнтський додаток відкривається у браузері та починає взаємодіювати

з бекендом у режимі реального часу через REST API, використовуючи механізми асинхронних запитів.

4.4 Основний функціонал застосунку

Реалізовано такі ключові можливості:

- перегляд списку товарів;
- перегляд детальної інформації про товар;
- авторизація та реєстрація користувачів (JWT);
- створення, редагування та видалення товарів (для адміністратора);
- створення замовлень;
- робота з базою даних через REST API.

Функціонал продемонстровано для кожного з трьох SPA-клієнтів, що підтверджує універсальність API.

Під час роботи застосунку було також перевірено стабільність взаємодії між бекендом і PostgreSQL, коректність виконання всіх CRUD-операцій (Create, Read, Update, Delete) для ключових сутностей бази даних, а також повну відповідність реалізації архітектурному REST-підходу (структура запитів, використання стандартизованих HTTP-методів та коректних статус-кодів). Окремо було підтверджено надійну роботу авторизації та механізмів доступу до захищених ресурсів, що гарантує цілісність та безпеку даних. Усі ці тести проводилися під помірним навантаженням для перевірки стійкості бекенд-сервісів. Окремо підтверджено, що всі три клієнти (React, Angular, Vue) успішно працюють із єдиним API без будь-яких додаткових налаштувань чи адаптацій на клієнтській стороні, що демонструє високу універсальність API-контракту. Отримані результати засвідчили високу стабільність системи та її повну відповідність усім висунутим технічним вимогам.

4.5 Висновки до розділу 4

У розділі було розглянуто повний цикл встановлення, налаштування та роботи розробленої системи. Наведені докладні інструкції підтверджують можливість безпроблемного кросплатформного розгортання застосунку на будь-якій платформі, що підтримує технологічний стек .NET і PostgreSQL. Це гарантує високу гнучкість при виборі виробничого середовища. Також продемонстровано коректну взаємодію трьох незалежних SPA-фронтендів з однією серверною частиною, що рішуче підтверджує універсальність обраного REST API та високу горизонтальну масштабованість архітектури. Реалізований функціонал пройшов успішне тестування і повністю відповідає поставленим завданням, а модульність та індустріальний стандарт використаних технологій дозволяє йому бути надійною основою для подальшого розвитку або комерційного використання.

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

Одним із заключних етапів магістерської роботи є формування стартап-проєкту, який демонструє можливості практичної реалізації розробленого програмного продукту у ринкових умовах. На відміну від попередніх розділів, що були зосереджені на технічних аспектах проєкту — архітектурі, реалізації API, взаємодії з фронтенд-фреймворками, — цей розділ має на меті продемонструвати здатність автора оцінювати комерційний потенціал розробленого рішення, визначати можливості його ринкового впровадження, формувати конкурентні переваги та стратегію виходу на ринок.

Побудова стартап-проєкту передбачає підбір ідеї, створення концепції продукту, аналіз ринку, визначення перспектив впровадження, оцінку конкурентів і формування маркетингової стратегії. Усі ці елементи разом дозволяють перевести технічний прототип у площину реального бізнес-проєкту, який може знайти власну нішу на ринку та забезпечити практичну цінність для користувачів.

У межах магістерської роботи як технічна основа стартапу розглядається розроблений універсальний інтернет-магазин із backend-частиною на ASP.NET Core та трьома незалежними клієнтськими інтерфейсами (React, Vue, Angular). Така архітектура відкриває широкі можливості подальшого розвитку продукту як комерційного сервісу, що пропонує бізнесам гнучкі інструменти для створення власних торговельних майданчиків. Ключовою особливістю є те, що застосунок з самого початку побудований на сучасних принципах інтеграції: API-орієнтований підхід, модульність, можливість підключення різних фронтендів та зовнішніх сервісів.

Таким чином, запропонований стартап спрямований на ринок електронної комерції, який продовжує активно зростати, особливо серед малого та середнього бізнесу, а також підприємців, що прагнуть швидко розгорнути власний інтернет-магазин без зайвих витрат та з можливістю подальшої кастомізації.

5.1 Формування ідеї стартап-проєкту

Перед початком розроблення стартап-проєкту необхідно чітко визначити основну ідею продукту, його призначення та користь для потенційних споживачів. На цьому етапі виконується окреслення проблеми, яку має розв'язувати проєкт, аналіз контексту її виникнення та розуміння, яку цінність створюватиме запропоноване рішення. Саме це формує концептуальну основу стартапу та визначає подальші напрями його розвитку.

5.1.1 Сутність ідеї

Сутність ідеї полягає у створенні програмного продукту — універсальної платформи інтернет-магазину, яку можна використовувати у двох форматах:

- як SaaS-рішення, що дозволяє підприємцям запускати власні магазини без технічної підготовки;
- як кастомізовану платформу, що може бути інтегрована у внутрішню інфраструктуру середніх підприємств, з можливістю розширення функціоналу та підключення сторонніх модулів.

Проєкт передбачає створення продукту під умовною назвою IntegraShop — універсального інтернет-магазину з акцентом на інтеграційні можливості. Основним технічним підґрунтям є реалізований у магістерській роботі REST API, який забезпечує роботу кількох незалежних клієнтів та дозволяє доповнювати систему іншими інструментами — мобільними додатками, зовнішніми системами оплати, CRM чи ERP.

Головною цінністю IntegraShop є технологічна гнучкість: магазин не лише функціонує як торговельний майданчик, а й може бути основою для побудови ширших екосистем. Це важливо для бізнесів, які прагнуть оптимізувати свої процеси,

інтегрувати дані про продажі з іншими системами, автоматизувати замовлення та підключати різні канали комунікації.

5.1.2 Напрями застосування

IntegraShop може бути використаний у різних типах бізнесів, і його гнучкість дозволяє адаптувати продукт під дуже різні потреби. Одним із ключових напрямів застосування є малі інтернет-магазини, які працюють у форматі «домашнього бізнесу». Для таких підприємців важливо мати просте й недороге рішення, яке можна швидко встановити, налаштувати та одразу почати працювати з товарами і замовленнями без глибоких технічних знань.

Другим важливим сегментом є середні компанії, яким потрібно оперативно розгорнути онлайн-магазин без значних фінансових і часових витрат. Для таких бізнесів критично, щоб платформа забезпечувала стабільність, мала базову аналітику, підтримувала оплату й могла масштабуватися разом зі зростанням кількості клієнтів.

IntegraShop також добре підходить підприємствам, які шукають готову платформу з можливістю подальшої кастомізації. Деякі компанії не хочуть створювати складне рішення «з нуля», але при цьому потребують індивідуальних функцій, інтеграцій чи розширень. Завдяки модульній архітектурі та відкритому API система може поступово адаптовуватися під унікальні процеси кожного замовника.

Окрему цінність продукт має для магазинів, які планують вихід на декілька каналів продажів — наприклад, поєднання веб-магазину, точки самообслуговування, маркетплейсів або соціальних мереж. Гнучка модель інтеграцій дозволяє централізовано керувати товарами й замовленнями навіть у складних мультимовних чи мультискладських сценаріях.

Крім комерційного використання, IntegraShop може застосовуватись і в освітньому середовищі. Навчальні заклади, IT-курси або університетські програми

можуть використовувати систему як інструмент для студентів, дозволяючи їм вивчати роботу з REST API, створювати власні фроненди, будувати інтеграції або досліджувати архітектуру веб-застосунків на реальному прикладі.

У підсумку IntegraShop є універсальним рішенням, яке однаково добре підходить як для B2C-, так і для B2B-сегментів, забезпечуючи широкий спектр можливостей для компаній різного масштабу.

5.1.3 Загальні дані стартап-проєкту

Для того щоб системно представити ключові елементи майбутнього продукту, доцільно узагальнити його зміст, можливі сфери застосування та основні вигоди для майбутніх користувачів. Це дозволяє чітко окреслити, на які потреби спрямований стартап і які цінності він створює для різних сегментів ринку.

Таблиця 5.1. – Загальні дані стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Універсальна платформа інтернет-магазину з API-орієнтованою архітектурою та підтримкою різних клієнтів	Малі та середні магазини, локальні виробники, торгові компанії, освітні платформи	Швидке розгортання магазину; інтеграція з будь-якими клієнтами; можливість роботи як SaaS чи on-premises; низькі витрати на старт

Ця таблиця демонструє цілісну концепцію продукту та дозволяє окреслити його головні цінності для ринку.

5.2 Аналіз ринку

Аналіз ринку є ключовим етапом формування стартап-проєкту, оскільки дозволяє оцінити потенційний попит на продукт, визначити його ринкову нішу та зрозуміти конкурентне середовище. Для IntegraShop, який орієнтується на сегмент малого та середнього бізнесу та пропонує готову та водночас гнучку платформу інтернет-магазину, ринок є надзвичайно перспективним. Сегмент електронної комерції стабільно зростає впродовж останніх десяти років і демонструє тенденцію до подальшого розширення навіть в умовах економічної нестабільності.

5.2.1 Оцінка місткості ринку

За даними міжнародних досліджень (Statista, eMarketer), світовий ринок електронної комерції щороку зростає на 15–20%, а в окремих регіонах — і більше. Пандемія COVID-19 значно прискорила цифровізацію бізнесу, що спричинило різке збільшення кількості малих підприємств, які почали переходити у формат онлайн-торгівлі. Навіть після стабілізації частина підприємств залишилась у цьому форматі.

Місткість українського ринку також демонструє тенденцію до зростання: навіть попри війну, частка онлайн-продажів збільшується, оскільки багато фізичних точок працюють у неповному режимі, а споживачі дедалі частіше надають перевагу онлайн-замовленням. Крім того, на українському ринку активно зростає сегмент локальних виробників, які шукають недорогі та прості рішення для старту.

Якщо говорити про глобальний ринок платформ інтернет-магазинів, то його обсяг оцінюється у мільярди доларів, і з кожним роком кількість компаній, що пропонують SaaS-рішення для онлайн-торгівлі, збільшується. Проте водночас зростає і попит на кастомізовані продукти, які дозволяють уникнути обмежень, закладених у типових системах.

Таким чином, місткість ринку для IntegraShop є значною як з точки зору загальної кількості потенційних клієнтів, так і з точки зору зростаючого попиту на продукти з розширеними можливостями інтеграції.

5.2.2 Цільові сегменти стартапу

На основі аналізу ринку можна виділити кілька цільових сегментів, на які орієнтується IntegraShop.

1. Малі інтернет-магазини та стартапи

Це найбільш численна категорія потенційних клієнтів. Багато малих підприємців шукають можливість швидко та недорого запустити свій магазин. У більшості випадків їхній бюджет є обмеженим, і вони не можуть дозволити собі розробку «з нуля». Для них важлива простота, наявність базового функціоналу та низькі витрати на місяць.

2. Середні підприємства

Це компанії, які вже мають певний обсяг продажів і прагнуть перейти на більш потужний інструмент для управління товарними запасами, клієнтами, замовленнями. Їм важливо мати можливість інтеграції інтернет-магазину зі складськими програмами, CRM, системами обліку.

3. Освітні заклади та навчальні платформи

Для навчальних курсів з веб-розробки або підприємництва потрібні платформи, які можна використовувати як навчальні кейси. Універсальність архітектури IntegraShop дозволяє легко демонструвати роботу REST API, клієнтів на різних фреймворках та базові принципи побудови веб-систем.

4. Фріланс-команди та агенції

Цей сегмент шукає продукт, який може бути основою для створення магазинів клієнтам. Гнучкість системи дозволяє адаптувати IntegraShop під різні потреби, що робить його зручним для компаній, які займаються розробкою сайтів.

5.3 Аналіз конкурентів

Аналіз конкурентів дозволяє визначити сильні та слабкі сторони існуючих систем, оцінити конкурентну позицію стартапу та сформулювати його унікальну ціннісну пропозицію. На ринку існує велика кількість інструментів для створення інтернет-магазинів, серед яких є як прості, так і досить складні рішення.

5.3.1 Основні конкуренти

На ринку платформ для створення інтернет-магазинів існує велика кількість рішень, і серед найвідоміших можна виділити Shopify, WooCommerce, OpenCart, PrestaShop, а також українські майданчики Rozetka Marketplace та Prom.ua. Кожне з цих рішень має свої переваги й обмеження, які важливо враховувати під час визначення конкурентного середовища.

Shopify є одним із найпопулярніших глобальних сервісів для електронної комерції. Його привабливість полягає у великій колекції готових шаблонів і тем, завдяки яким користувач може швидко запустити магазин без глибоких технічних знань. Додатковою перевагою є інтеграція з основними платіжними сервісами, що значно спрощує прийом оплат. Сам інтерфейс сервісу орієнтований на новачків, тому розгорнути базовий магазин можна практично без досвіду. Водночас Shopify має і суттєві недоліки: висока щомісячна плата, особливо з урахуванням транзакційних комісій, обмежує доступність для малого бізнесу. Крім того, кастомізація магазину на

рівні коду є доволі складною, а сама платформа жорстко прив'язана до SaaS-моделі, тож користувач залежить від умов сервісу й не може вільно працювати з серверною частиною.

WooCommerce, який працює як розширення до WordPress, займає іншу нішу. Це рішення приваблює тих, кому потрібна велика кількість додаткових модулів і надбудов, оскільки екосистема WordPress надзвичайно обширна. WooCommerce дозволяє запустити магазин із мінімальними витратами, що робить його популярним серед дрібних і середніх бізнесів. Але водночас платформа має суттєві обмеження: вона повністю залежить від WordPress, тому всі сильні й слабкі сторони цього CMS автоматично успадковуються й магазином. Інтеграція з нестандартним програмним забезпеченням або складними бізнес-процесами може вимагати значного доопрацювання, а продуктивність залежить від якості хостингу.

До безкоштовних або умовно безкоштовних рішень належать OpenCart і PrestaShop. Вони приваблюють тим, що дають більше свободи в керуванні сервером, а також мають величезну кількість сторонніх модулів і шаблонів. Це дозволяє створювати навіть складні магазини без значних фінансових вкладень. Однак технологічна база цих платформ поступово застаріває, що виявляється у слабкій масштабованості, можливих проблемах із продуктивністю та підвищених вимогах до технічного обслуговування. Питання безпеки також є одним із критичних — велика кількість сторонніх розширень і недостатній темп оновлень збільшують ризики вразливостей.

Окрему групу складають Rozetka Marketplace та Prom.ua — українські платформи, які дозволяють продавати товари без створення повноцінного інтернет-магазину. Їхньою основною перевагою є доступ до значної аудиторії, що дозволяє бізнесу швидко отримати перші продажі. Також процес старту простий і не потребує технічних знань: користувачеві достатньо завантажити каталог товарів та налаштувати базові параметри. Водночас такі майданчики практично не дають можливості кастомізації, і бізнес повністю залежить від їхніх правил. Додатковою

проблемою є комісії з продажів, що зменшують маржинальність і ускладнюють довгострокове масштабування бізнесу.

Узагальнюючи, кожен із наведених конкурентів займає власну нішу, але всі вони мають обмеження, які відкривають простір для створення більш гнучкого, технологічно сучасного та кастомізованого рішення — саме таким і є пропонуванй стартап.

5.3.2 Визначення характеристик ідеї проєкту

У межах аналізу конкурентів доцільно визначити ключові характеристики проєкту та порівняти їх із типовими рішеннями, представленими на ринку. Це дозволяє оцінити, у яких аспектах наявні аналоги переважають, у яких — поступаються, а також виявити точки росту і конкурентні переваги власного продукту. З цією метою сформовано узагальнену порівняльну таблицю, у якій наведено основні технологічні та функціональні параметри, характерні для схожих систем та запропонованого рішення.

Таблиця 5.2 – Характеристики проєкту

Характеристики	Концепція конкурентів	Слабкі сторони	Нейтральні сторони	Сильні сторони	Власний проєкт
Архітектура	Часто моноліт	Складність змін	Швидкий старт	Розвинена інфраструктура	API-орієнтована, модульна
Гнучкість інтеграції	Лімітовані API	Складність розширення	Наявність базових API	Модульна система	Легка інтеграція з будь-якими клієнтами

Кінець таблиці 5.2

Характеристик и	Концепція конкурентів	Слабкі сторони	Нейтраль ні сторони	Сильні сторони	Власний проект
Вартість	Середня/висо ка	Дорогі тарифи	Пробні тарифи	Надійніст ь	Freemium + SaaS
Масштабованіс ть	Обмежена	Проблем и при рості	Достатня	Сильна	Масштабуван ня за рахунок .NET
Кастомізація	Лімітована	Залежніст ь від розширен ь	Доступна	Широка	Full-stack доступ

Подана порівняльна характеристика демонструє, що власний проєкт суттєво вирізняється на тлі існуючих рішень завдяки модульній архітектурі, відкритості API та гнучкості інтеграції з різними типами клієнтів. На відміну від конкурентних підходів, які часто базуються на монолітних схемах і обмежених API, розроблена система передбачає легке масштабування та кастомізацію, що забезпечує її адаптивність до різних сценаріїв використання. Завдяки поєднанню низької вартості підтримки, можливості роботи у форматі freemium/SaaS та розширюваності за рахунок .NET-екосистеми, проєкт демонструє конкурентоспроможність і потенціал для подальшого розвитку як універсального рішення для інтеграції сучасних веб-застосунків.

5.4 Аудит технологій

У цьому підрозділі проведено аудит технологій, що лежать в основі створюваного програмного продукту. Такий аналіз дозволяє оцінити відповідність обраних технічних рішень цілям проєкту, їхню актуальність, надійність та потенціал для подальшого розвитку. Спершу розглядається технологічна основа продукту, після чого оцінюється його технологічна здійсненність та перспективи масштабування.

5.4.1 Технологічна основа продукту

Розроблений програмний продукт базується на таких технологіях:

- ASP.NET Core — високопродуктивний фреймворк для побудови веб-додатків;
- REST API — сучасний стандарт інтеграції клієнтських застосунків;
- Entity Framework Core для роботи з базами даних;
- JWT-аутентифікація для безпеки доступу до API;
- React, Vue, Angular — сучасні фреймворки для фронтенд-реалізації;
- Swagger для документування API.

Використання зазначених технологій забезпечує продукту високу гнучкість, масштабованість та сумісність із широким спектром сучасних клієнтських платформ. Така технологічна основа дозволяє легко розширювати функціональність системи, інтегрувати додаткові сервіси та адаптувати рішення під різні бізнес-вимоги. Завдяки модульності та стандартизованим протоколам взаємодії продукт залишається відкритим для подальшого розвитку й упровадження у реальні комерційні чи корпоративні середовища.

5.4.2 Оцінка технологічної здійсненності

IntegraShop є повністю здійсненним із технологічної точки зору. Насамперед це пояснюється тим, що обрана архітектура не містить критичних вузьких місць і не прив'язана до специфічних або застарілих рішень. Уся серверна частина побудована за принципами модульності: бізнес-логіка винесена в окремі сервіси, робота з даними — у контекст бази даних, а клієнтське звернення реалізовано через чітко визначені REST-ендпоїнти. Завдяки цьому система залишається гнучкою, стійкою та здатною до масштабування без необхідності переписувати основну частину коду.

Технологічна база проєкту також сприяє його практичній реалізованості. Використання ASP.NET Core, Entity Framework Core та PostgreSQL означає, що рішення спирається на зрілі, підтримувані та активно розвивані інструменти, для яких існує велика кількість готових бібліотек, документації та прикладів застосування. Ці технології давно зарекомендували себе у промислових системах, що забезпечує надійність і прогнозованість поведінки під навантаженням.

Окремої уваги заслуговує модульність проєкту. Кожен функціональний блок — робота з товарами, управління замовленнями, модуль авторизації, кошик користувача — реалізований як самостійний компонент. Це дозволяє безболісно видаляти або замінювати окремі частини, не змінюючи ключові механізми взаємодії всередині системи. Наприклад, за необхідності модуль оплати можна буде замінити на сторонній сервіс або розширити можливостями інтеграції з CRM чи маркетинговими платформами, не торкаючись решти застосунку.

Таким чином, технологічна реалізованість IntegraShop обумовлена сукупністю факторів: правильною архітектурою, використанням сучасних перевірених технологій та структурною гнучкістю, яка забезпечує подальший розвиток системи без ризику значних технічних перешкод.

5.5 Бізнес-модель та монетизація

Для проєкту було обрано гібридну бізнес-модель, яка дає змогу охопити різні категорії користувачів — від невеликих інтернет-магазинів до середніх компаній та корпоративного сегменту. Основою моделі є підхід freemium, який передбачає, що базовий функціонал платформи залишається безкоштовним. Завдяки цьому користувач може розпочати роботу без жодних фінансових бар'єрів, оцінити можливості системи та поступово інтегрувати її у власні бізнес-процеси. Підхід freemium також сприяє формуванню активної спільноти користувачів і підвищує впізнаваність продукту на ринку.

Наступним елементом монетизації є модель SaaS-підписки, що відкриває доступ до розширених можливостей. Йдеться про поглиблену статистику продажів, інтеграції з зовнішніми сервісами, автоматизацію процесів обробки замовлень, а також додаткові модулі, які потрібні для масштабування бізнесу. Такий формат дозволяє отримувати передбачуваний щомісячний дохід і водночас стимулює користувачів поступово переходити на вищі тарифи, коли їх бізнес зростає.

Окремим напрямом є корпоративна on-premises-ліцензія, орієнтована на великі компанії, які потребують більшої незалежності та контролю над усіма складовими системи. У цьому випадку IntegraShop може бути розгорнутий безпосередньо в інфраструктурі замовника, із можливістю доступу до вихідного коду та індивідуальних умов технічної підтримки. Такий формат підходить для підприємств із підвищеними вимогами до безпеки або специфічних внутрішніх процесів.

У сукупності ця модель дає змогу охопити широкий ринок і пропонувати оптимальний сценарій кожному типу користувачів: від тих, хто лише починає продажі онлайн, до великих компаній зі складною інфраструктурою. Це забезпечує стабільність бізнесу та можливість масштабування в майбутньому.

5.6 План впровадження стартап-проєкту

План впровадження стартап-проєкту IntegraShop охоплює декілька послідовних етапів, кожен з яких має власні завдання, цілі та очікувані результати. Такий підхід дозволяє структурувати процес розвитку продукту, мінімізувати ризики та оптимально розподілити ресурси. У плані враховано як технічні, так і бізнесові аспекти, що є необхідною умовою для успішного виходу на ринок.

5.6.1 Стадія MVP (Minimum Viable Product)

На етапі створення MVP головним завданням є підготувати мінімальну, але повністю працездатну версію системи, яка демонструє ключові можливості платформи та може бути використана для перших тестів і зворотного зв'язку. Цей етап зосереджений на тому, щоб закласти основу продукту: реалізувати базові механізми інтернет-магазину, забезпечити стабільну роботу серверної частини та підключити хоча б один фронтенд-клієнт для взаємодії з API.

Насамперед формується технічне ядро системи. На цьому кроці визначається й реалізується структура товарів, категорій та замовлень, а також логіка їх обробки у базовому режимі. Створюється REST API для основних операцій, інтегрується механізм аутентифікації на основі JWT, що дає змогу забезпечити контроль доступу та підтримувати авторизовану роботу користувачів. Важливим результатом цієї частини є можливість повноцінної взаємодії хоча б одного фронтенд-застосунку з API, що підтверджує працездатність архітектури.

Паралельно налаштовується інфраструктурна частина: серверна частина розгортається у хмарному середовищі, конфігурується база даних (SQL Server або PostgreSQL), встановлюється система резервного копіювання та початкові заходи

безпеки. За потреби застосовується контейнеризація за допомогою Docker, що спрощує розгортання й відтворюваність середовищ.

Важливим складником MVP є документація. На цьому етапі готується Swagger-документація, яка дає можливість швидко та прозоро працювати з API. Також формуються короткі інструкції для початкового користування та технічні описи, які спростять інтеграцію стороннім розробникам.

Заключний етап — внутрішнє тестування. Воно проводиться розробниками та обмеженим колом користувачів, що дозволяє оперативно виявити найкритичніші помилки й переконатися в тому, що продукт працює стабільно в межах заявленого функціоналу.

Очікуваний результат етапу MVP — це готовий до демонстрації та раннього використання продукт, який забезпечує виконання базових операцій електронної комерції та спирається на надійний фундамент у вигляді працездатного REST API.

5.6.2 Стадія Launch (запуск на ринок)

Після того як мінімально життєздатна версія продукту продемонструвала працездатність і пройшла первинне тестування, наступним кроком стає його публічний запуск. На цьому етапі важливо не лише завершити технічні доробки, а й підготувати продукт до появи реальних користувачів, сформулювати перше враження про бренд та забезпечити мінімальну маркетингову активність.

Починається цей етап із фіналізації технічної частини — виправляються помилки, виявлені під час тестування MVP, оптимізується робота інтерфейсу та покращується користувацький досвід, щоб взаємодія з системою була інтуїтивною і стабільною. У цей же час додаються перші інтеграції, необхідні для практичного використання продукту, зокрема підключення платіжних систем, сервісів доставки або базових сторонніх модулів.

Паралельно запускається маркетингова підготовка. Створюється офіційний сайт продукту, де детально описано його можливості, переваги та варіанти використання. Готуються презентаційні матеріали та короткі рекламні огляди. Відбуваються перші публікації у професійних спільнотах, а також оформлюються соціальні сторінки бренду, що дозволяє залучати аудиторію та формувати першу спільноту користувачів.

Для успішного публічного запуску необхідно налагодити підтримку користувачів. Формується база знань з відповідями на типові запитання, створюється канал для зворотного зв'язку, а також впроваджується проста система тікетів, що дозволяє оперативно реагувати на звернення та підтримувати користувачів на стартових етапах.

Також інтегруються інструменти аналітики — Google Analytics, Matomo або Mixpanel. Це дає змогу відстежувати активність та поведінку користувачів, аналізувати залученість та отримувати перші статистичні дані про те, як люди взаємодіють із продуктом і які функції потребують особливої уваги.

Очікуваний результат етапу Launch — це публічна доступність продукту, поява перших реальних користувачів та отримання важливого зворотного зв'язку, який визначить пріоритетні напрями розвитку на наступні етапи.

5.6.3 Стадія Growth (ріст і масштабування)

На етапі росту основний акцент робиться на вдосконаленні продукту, розширенні його можливостей та поступовому виході на ширший ринок. Це той момент, коли система вже має сформовану базу користувачів, і подальший розвиток напряму залежить від швидкої реакції на їхні потреби та здатності команди адаптувати рішення до реальних бізнес-сценаріїв.

Першим напрямом роботи є розширення функціональності. На цьому етапі продукт починає отримувати більш професійні можливості, які потрібні бізнесам із

більшими обсягами діяльності. Наприклад, впроваджується підтримка багатоскладських систем — це дозволяє компаніям працювати з кількома точками зберігання товарів, оптимізувати логістику та уникати плутанини в залишках. Також вдосконалюється система ролей: з'являються гнучкі налаштування прав доступу для адміністраторів, менеджерів, операторів і технічного персоналу, що робить роботу з системою безпечнішою й адаптованою під реальні бізнес-процеси. Паралельно інтегруються додаткові способи оплати, оскільки з розширенням аудиторії зростає потреба підтримувати різні платіжні сервіси. Ще одним напрямом розвитку є розробка аналітичного модуля, який дозволяє відстежувати продажі, динаміку замовлень, рентабельність і загальний стан бізнесу. Крім того, реалізуються інтеграції з популярними CRM-системами, такими як Bitrix24 чи HubSpot, що дозволяє компаніям вибудовувати більш ефективний цикл взаємодії з клієнтами.

Другим важливим напрямом є масштабування інфраструктури. Зі збільшенням кількості користувачів серверні ресурси потребують перерозподілу навантаження, тому впроваджуються балансувальники, які забезпечують стабільну роботу навіть при великих пікових навантаженнях. За необхідності архітектура може бути поступово розділена на мікросервіси — це дозволяє незалежно масштабувати окремі компоненти системи. Для підвищення швидкодії додається кешування на основі Redis, що значно пришвидшує повторні запити та зменшує навантаження на базу даних. Якщо в системі використовується функціонал реального часу, додатково впроваджується backplane, який забезпечує коректну роботу WebSocket-з'єднань у розподіленому середовищі з кількома серверами.

Після стабілізації інфраструктури відкривається можливість виходу на B2B-сегмент. Продукт починає пропонувати корпоративні ліцензії й можливість індивідуального доопрацювання під потреби великих компаній. Для таких клієнтів зазвичай створюються пакети підтримки, SLA-контракти та гарантії безперебійної роботи, що підвищує довіру та робить рішення привабливим для серйозних бізнесів.

Окремою частиною етапу Growth є оптимізація внутрішніх бізнес-процесів. Запускається автоматизована система білінгу, що спрощує оплату підписок і зменшує ручні витрати. Для залучення нових користувачів може бути розгорнута реферальна програма, а партнерські інтеграції дозволяють формувати ширшу екосистему навколо продукту.

Очікуваний результат етапу Growth — стабільне збільшення кількості користувачів, зміцнення конкурентних позицій продукту та зростання частки платних клієнтів. Саме цей етап визначає довгострокову життєздатність проєкту та його спроможність виходити на нові ринки.

5.7 Аналіз ризиків

Оцінка ризиків є важливою частиною планування стартапу. Ризики можна поділити на кілька груп:

5.7.1 Ринкові ризики

Ринкові ризики є одними з найбільш значущих і непередбачуваних для будь-якого технологічного стартапу, особливо в галузі електронної комерції (e-commerce). Цей сегмент характеризується надзвичайно високою конкуренцією та винятково швидкою зміною трендів. Для проєкту IntegraShop можна виділити кілька ключових ринкових загроз, які вимагають постійної уваги.

Найперший і найбільш очевидний ризик — це висока конкуренція в сегменті e-commerce платформ. Ринок платформ для створення інтернет-магазинів вже сьогодні насичений великою кількістю як глобальних, так і потужних локальних гравців. До них належать такі гіганти, як Shopify, WooCommerce, Wix, а також великі українські

маркетплейси, наприклад, Rozetka Marketplace та Prom.ua. Більшість цих конкурентів вже мають усталену клієнтську базу, розгалужену партнерську мережу та значні маркетингові бюджети. Це створює високі бар'єри для входу на ринок для нового гравця, як IntegraShop, і вимагає чіткого, унікального та вигідного позиціонування продукту.

Далі йде ризик, пов'язаний зі швидкою зміною технологічних трендів. Поведінка та очікування користувачів постійно змінюються: вони звикають до нових форматів взаємодії, таких як використання соціальних мереж як основних каналів продажів, очікують досконалих мобільних застосунків та вимагають персоналізованих AI-рекомендацій. Якщо стартап не зможе вчасно та гнучко адаптуватися до цих технологічних змін і нових вимог, він ризикує втратити свою актуальність на ринку.

Істотну загрозу становить і загальна економічна нестабільність. Зміни купівельної спроможності населення, високі темпи інфляції, суттєві валютні коливання або загальна економічна криза можуть призвести до того, що бізнеси стануть менш охочими інвестувати у розвиток власних онлайн-магазинів та підписку на платформи. Це безпосередньо впливає на потенційний дохід і темпи зростання клієнтської бази IntegraShop.

Нарешті, необхідно враховувати ризик зміщення попиту. Якщо споживачі продовжать масово переходити до моделі покупок через маркетплейси, де продавцям не потрібно створювати та підтримувати власний окремий сайт, це може значно знизити попит на класичні платформи для окремих інтернет-магазинів, такі як IntegraShop.

Розуміння цих ринкових ризиків має вирішальне значення, оскільки це дозволяє команді сформувати правильне та конкурентне позиціонування продукту і визначити ті напрямки розвитку, які забезпечать йому необхідну конкурентоспроможність і життєздатність навіть у дуже динамічному ринковому середовищі.

5.7.2 Технологічні ризики

Оскільки платформа IntegraShop базується на сучасних веб-технологіях та має ключову API-орієнтовану архітектуру, критично важливо заздалегідь оцінити технологічні ризики та обмеження, що можуть виникнути протягом процесу експлуатації та подальшого масштабування.

Перш за все, існує прямий ризик, пов'язаний із проблемами стабільності та продуктивності при зростанні навантаження. Зі стрімким збільшенням кількості клієнтів і магазинів, які активно працюють на платформі, система може зіткнутися з серйозним зниженням швидкодії. Це може проявлятися як збільшення часу відповіді API, або виникнення так званих «вузьких місць», що погіршують загальний досвід користувача. Для запобігання цьому необхідне постійне забезпечення горизонтального масштабування інфраструктури та безперервний моніторинг продуктивності системи.

Наступний ключовий ризик — потенційні уразливості в API. Будь-яка система, що використовує відкриті або навіть закриті API для обміну даними, автоматично опиняється під загрозою різноманітних кібератак. Це включає класичні загрози, такі як SQL-ін'єкції, атаки типу DDoS, спроби підбору токенів або несанкціонованої крадіжки даних. Неправильна реалізація механізмів авторизації чи недостатньо жорсткі політики безпеки можуть призвести до серйозного порушення конфіденційності клієнтської та бізнесової інформації.

Також варто враховувати складність інтеграцій із зовнішніми сервісами. Хоча API-архітектура IntegraShop і дозволяє відносно легко інтегрувати сторонні сервіси (платіжні системи, служби доставки, CRM), кожна така інтеграція вимагає значних ресурсів: часу на розробку, ретельного тестування та, що важливо, постійної підтримки. При цьому, будь-які несподівані зміни у зовнішніх API можуть створити високий ризик непрацездатності окремих модулів або функціоналу платформи.

Нарешті, необхідно управляти ризиком технологічної залежності. Використання таких рішень, як .NET Core та Entity Framework Core (EF Core), є очевидною перевагою з точки зору швидкості розробки та стабільності, але це також створює певну залежність від екосистеми Microsoft. У разі непередбачуваних або радикальних змін у цих фреймворках (наприклад, при суттєвому оновленні версій), може знадобитися виділення додаткових значних ресурсів для оперативної адаптації платформи.

Загалом, мінімізація технологічних ризиків вимагає проактивного підходу до безпеки, постійного аудиту коду та планування інфраструктури з урахуванням майбутнього експоненційного зростання.

5.7.3 Організаційні ризики

Організаційні ризики безпосередньо стосуються внутрішніх аспектів роботи команди, ефективності керування проектом, управління людськими ресурсами та загальної структури робочого процесу. Їхнє ігнорування може підірвати внутрішню стійкість будь-якого проекту.

Однією з найбільш поширених проблем є нестача кваліфікованих спеціалістів. Стартапи, особливо на ранніх етапах, часто функціонують з дуже обмеженою кількістю розробників та технічних фахівців. Зі зростанням популярності продукту та збільшенням клієнтської бази неминуче виникає потреба у швидкому розширенні команди. Це може стати серйозною перешкодою з двох причин: або через недостатність фінансів для найму, або через складнощі з пошуком справді кваліфікованих кадрів, здатних швидко інтегруватися в робочий процес.

Тісно пов'язаний з цим чинник — високий ризик професійного вигорання. Команди стартапів, як правило, працюють у дуже інтенсивному режимі, особливо на початкових етапах запуску та масштабування. Якщо робочий процес організований

неоптимально, з постійними авралами та нечіткими дедлайнами, це швидко призводить до професійного вигорання. Такий стан, своєю чергою, критично негативно впливає на загальну продуктивність, креативність та якість кінцевого продукту.

Крім того, значну загрозу несуть недостатньо налагоджені бізнес-процеси. Відсутність чіткої структури управління, якісної внутрішньої документації, структурованого планування спринтів та надійного контролю якості (QA) є прямим шляхом до дестабілізації. Без цих базових елементів продукт може втратити свою функціональну стабільність, а команда не зможе працювати ефективно та узгоджено.

Окремим, але дуже важливим організаційним ризиком є ті, що пов'язані з підтримкою клієнтів. На початкових етапах надзвичайно важливо оперативно та якісно реагувати на будь-які проблеми, запитання чи збої, які виникають у перших користувачів. Невчасна або неякісна підтримка може швидко зіпсувати репутацію продукту, оскільки перші відгуки мають вирішальне значення для подальшого зростання та довіри.

Таким чином, для мінімізації організаційних ризиків необхідна активна вибудова структурованих внутрішніх процесів, а також розумний і збалансований розподіл робочого навантаження. Вчасне реагування на ці внутрішні ризики є ключовим для забезпечення як стабільної та здорової роботи команди, так і незмінно високої якості продукту.

5.7.4 Фінансові ризики

Фінансові ризики є критично важливим викликом, особливо для проєктів, які знаходяться на ранній стадії розвитку, коли неминуче спостерігається ситуація, в якій операційні та капітальні витрати значно переважають поточні доходи. Ця фаза вимагає особливо уважного підходу до планування та управління ресурсами.

Один із першочергових ризиків — це недостатність початкового фінансування. Навіть за умов роботи мінімальної команди та використання lean-підходу, запуск будь-якого SaaS-рішення вимагає регулярних і невідкладних витрат. Сюди входять оплата хостингу та домену, інвестиції у первинний маркетинг для залучення перших користувачів, а також забезпечення базової технічної підтримки. Якщо виділене фінансування виявиться занадто обмеженим, це не просто створить напругу, але й може суттєво сповільнити критично важливий розвиток продукту.

Інший важливий аспект — це довгий період до виходу на окупність. Сегменти B2B та SaaS часто характеризуються відносно повільним зростанням прибутковості на старті. Причина цього проста: на початкових етапах велика частина користувачів може бути безкоштовною, а отже, вони не генерують доходу. Щоб бізнес почав стабільно працювати у плюс, потрібен значний час для формування стійкої та лояльної бази саме платних клієнтів.

Крім того, завжди існує загроза непередбачуваних витрат, які можуть виникнути раптово і потребувати негайної уваги. До них можуть належати необхідність інтеграції з додатковими сервісами, які не були заплановані спочатку, витрати на адаптацію до змін у законодавстві (наприклад, нові, більш жорсткі вимоги щодо захисту персональних даних та GDPR), а також непланові оновлення технологічної інфраструктури, необхідні для підтримки безпеки та масштабованості. Усі ці фактори можуть несподівано збільшити загальний обсяг витрат.

Нарешті, ризик втрати доходу у разі технічних проблем є прямим фінансовим ударом. Будь-які збої, простої чи критичні помилки у роботі сервісу можуть призвести до необхідності надання компенсацій постраждалим клієнтам або, що ще гірше, до безповоротної втрати їхньої довіри та переходу до конкурентів.

З огляду на всі ці потенційні загрози, боротьба з фінансовими ризиками вимагає багатовекторної стратегії: створення надійного резервного бюджету (фінансової подушки), впровадження ефективної системи прогнозування та контролю витрат, а

також обов'язкова побудова гнучкої бізнес-моделі, яка не залежатиме критично від одного, єдиного джерела доходу.

5.7.5 Заходи мінімізації ризиків

Для того щоб знизити вплив ризиків, окреслених у попередніх підрозділах, проєкт потребує системного підходу, який охоплює як технічні, так і організаційні аспекти. Одним із ключових запобіжних механізмів є гнучка інженерна архітектура. Модульність IntegraShop дозволяє незалежно оновлювати окремі компоненти, розширювати функціональність без втручання у всю систему та швидко реагувати на зміни ринку чи вимоги замовників. Такий підхід суттєво зменшує ризики, пов'язані з технічними боргами або складністю масштабування.

Важливою складовою є також побудова надійного процесу доставки оновлень. Впровадження CI/CD дає можливість автоматизувати тестування, розгортання та виявлення помилок ще до того, як зміни потраплять у продуктивне середовище. Це знижує частоту людських помилок та дозволяє релізити оновлення поступово і прогнозовано, з мінімальним ризиком для стабільності системи.

Безпека залишається одним із найбільш критичних пунктів для будь-якого сучасного веб-продукту, тому регулярні аудити необхідні на всіх етапах розвитку. До них входять перевірка коректності використання токенів, діагностика вразливостей на зразок SQL Injection, ревізія прав доступу та постійний моніторинг серверних логів для виявлення потенційно небезпечних дій. Такий підхід дозволяє завчасно знайти слабкі місця та усунути їх до виникнення реальних інцидентів.

Не менш важливу роль відіграє організація роботи команди. Використання гнучких методологій на кшталт Agile або Scrum забезпечує регулярну комунікацію, прозорий розподіл задач та можливість швидко реагувати на зміни. Постійні мітинги,

рев'ю роботи та культура відкритого обговорення дозволяють мінімізувати ризики, пов'язані з неузгодженістю або помилками в плануванні.

Фінансові ризики також потребують уваги. Для їх зменшення доцільно формувати резервний бюджет, оптимізувати витрати на хостинг та використовувати хмарні платформи з моделлю оплати “pay as you go”, що дозволить сплачувати лише за фактичне використання ресурсів і не перевантажувати бюджет на старті.

Крім того, диверсифікація джерел доходу — один із найефективніших способів зміцнити стійкість стартапу. Поєднання SaaS-підписки, freemium-моделі та корпоративних ліцензій робить бізнес менш залежним від одного сегмента ринку та забезпечує стабільніший грошовий потік.

У сукупності ці заходи створюють комплексний захист від основних ризиків і дозволяють проєкту розвиватися стабільно навіть в умовах невизначеності.

5.8. Перспективи розвитку стартапу

Перспективи розвитку IntegraShop виглядають дуже обнадійливими, особливо з огляду на стрімкий ріст електронної комерції. З технічної точки зору компанія планує розширювати функціонал платформи: серед пріоритетів — створення мобільного застосунку на базі React Native або MAUI, впровадження GraphQL API, запуск власного маркетплейсу модулів, інтеграція з провідними логістичними сервісами, такими як Нова Пошта та Meest, а також використання штучного інтелекту для рекомендацій товарів та аналітики продажів.

З бізнесової перспективи IntegraShop має намір виходити на міжнародні ринки, зокрема ЄС та США, розвивати франшизу, налагоджувати партнерства з освітніми платформами та пропонувати компаніям white-label рішення, що дозволить їм ефективно використовувати технології IntegraShop під власним брендом.

Що стосується інновацій, компанія фокусується на впровадженні smart-аналітики продажів, адаптивного ціноутворення та автоматизації складів, що дозволить оптимізувати бізнес-процеси та підвищити ефективність роботи платформи. Усі ці напрямки формують цілісну стратегію розвитку, яка поєднує технологічні, бізнесові та інноваційні аспекти.

5.9. Висновки до розділу 5

У розділі було проведено всебічний аналіз можливостей перетворення розробленого програмного забезпечення на комерційний продукт. На основі реалізованого інтернет-магазину запропоновано концепцію стартапу IntegraShop, який має значний потенціал у сегменті малого та середнього бізнесу.

Проведений аналіз ринку підтвердив наявність стабільного зростання електронної комерції, що створює сприятливі умови для розвитку продукту. Конкурентний аналіз дозволив визначити місце стартапу серед існуючих рішень та сформулювати його ключові переваги, зокрема API-орієнтовану архітектуру, можливість підключення різних клієнтських застосунків, масштабованість та гнучкість.

У межах розділу розроблено бізнес-модель, що включає freemium-підхід, SaaS-підписку та корпоративну ліцензію. Також складено детальний план впровадження з поділом на етапи MVP, Launch та Growth, проведено аналіз ризиків і визначено перспективи розвитку проєкту.

Узагальнюючи, можна стверджувати, що створений прототип має реальні передумови для ринкового впровадження та подальшої комерціалізації. IntegraShop може не лише задовольнити поточні потреби малого та середнього бізнесу, а й стати основою для масштабованої платформи електронної комерції.

ВИСНОВКИ

В ході виконання даної роботи було зроблено такі узагальнені висновки:

1. У результаті дослідження виконано комплексний аналіз сучасних методів інтеграції клієнтської та серверної частин веб-застосунків. Розглянуто REST, GraphQL, gRPC, WebSocket/SignalR, Blazor та SPA-архітектури, а також визначено їхні ключові переваги, недоліки й сфери застосування. Порівняльний аналіз показав, що REST залишається найбільш універсальним рішенням для типової інтеграції, тоді як GraphQL і gRPC ефективні в специфічних сценаріях високої складності або навантаження. Таким чином виконано перше завдання роботи — уточнення сучасного стану проблеми інтеграції веб-систем.

2. На підставі теоретичного аналізу обґрунтовано вибір REST API як основного методу інтеграції для реалізації практичного прототипу. Це рішення прийнято завдяки простоті реалізації, передбачуваності поведінки, широкій підтримці інструментів та сумісності з будь-якими клієнтськими технологіями. Результатом стало чітке формування вимог до серверної частини, моделі даних і структури програмної системи.

3. У практичній частині роботу виконано шляхом розробки прототипу веб-застосунку на основі ASP.NET Core та PostgreSQL. Створено архітектуру з розділенням на шари даних, логіки та контролерів, реалізовано RESTful API з підтримкою CRUD-операцій та впроваджено документування через Swagger/OpenAPI. Забезпечено коректну обробку вхідних даних, стандартизовані коди відповідей HTTP та передбачуваний формат JSON-відповідей. Така структура підтверджує виконання третього завдання — побудови функціональної архітектури прототипу.

4. Проведене тестування REST API засвідчило стабільність роботи всіх ендпоінтів та правильність обробки як коректних, так і помилкових запитів. Перевірки через Swagger UI, Postman і модульні тести із використанням InMemory-бази

підтвердили відповідність API вимогам та його здатність до масштабування й розширення. Цим повністю реалізовано четверте завдання роботи — аналіз і підтвердження коректності функціонування створеної системи.

5. Оцінка можливості комерціалізації показала, що розроблене рішення може виступати технічною основою для стартап-проєкту, орієнтованого на мультифреймворкнуту інтеграцію клієнтських застосунків. Проаналізовано ринку, бізнес-моделі, конкурентні переваги та перспективи розвитку продукту. Доведено, що створений API є універсальним і може інтегруватися з будь-яким сучасним фронтеном, що підтверджує практичну цінність отриманих результатів.

6. У підсумку всі поставлені у вступі завдання були виконані повністю: здійснено огляд сучасних інтеграційних підходів, обрано оптимальний метод їх реалізації, створено і протестовано прототип REST API та визначено можливості його впровадження в реальних умовах. Отримані результати демонструють здатність автора роботи вирішувати комплексні інженерні завдання у сфері веб-розробки та інтеграції систем.

7. Перспективи подальшого розвитку включають розширення прототипу за рахунок впровадження GraphQL, gRPC або SignalR, створення повноцінного SPA-інтерфейсу на React/Vue/Angular, додавання системи ролей та рефреш-токенів, оптимізацію продуктивності та побудову мікросервісної архітектури. Це відкриває можливість масштабування проєкту та його адаптації до потреб реальних комерційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : dissertation for the degree of Doctor of Philosophy in Information and Computer Science / University of California. Irvine, 2000. 180 с.
2. GraphQL is the better REST. HOW TO GRAPHQL. URL: <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>.
3. Amareen S. GraphQL Adoption and Challenges: Community-Driven Insights from StackOverflow Discussions. arXiv. URL: <https://arxiv.org/html/2408.08363v1>.
4. Serving over HTTP. GraphQL. Learn > Serving over HTTP. URL: <https://graphql.org/learn/serving-over-http/>.
5. Choose Between Traditional Web Apps and Single Page Apps (SPAs). Microsoft Learn. Modern ASP.NET web applications e-book. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps>.
6. Real-time ASP.NET with SignalR. Microsoft. URL: <https://dotnet.microsoft.com/en-us/apps/aspnet/signalr>.
7. Newton-King J. Compare gRPC services with HTTP APIs. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-10.0>.
8. Is GraphQL Better Than REST? GraphQL vs REST Comparison. tailcall. 18.07.2024. URL: <https://tailcall.run/graphql/graphql-vs-rest-api-comparison/>.
9. Perf benchmarks between REST, gRPC-Web and SignalR. Reddit. /r/dotnet. URL: https://www.reddit.com/r/dotnet/comments/i781gm/perf_benchmarks_between_rest_grpcweb_and_signalr/.
10. Monsur E. M. H. gRPC vs SignalR: Choosing the Right Tool for Real-Time Communication. Medium. URL: <https://medium.com/@hasanmcse/grpc-vs-signalr-choosing-the-right-tool-for-real-time-communication-9f5e191c933f>.

11. Blazor WebAssembly vs. Server: Which One to Choose?. Telerik. URL: <https://www.telerik.com/faqs/blazor/blazor-webassembly-vs-server>.
12. Hasan F. Use JavaScript Services to Create Single Page Applications in ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/client-side/spa-services?view=aspnetcore-3.0>.
13. Падалко Д. Р. Технології інтеграції застосунків asp.net core з сучасними фронтенд-інструментами. Сучасні проблеми наукового забезпечення енергетики : Матеріали XXII Міжнародної науково-практичної конференції молодих вчених і студентів, м. Київ, Україна, 22–25 квіт. 2025 / НТУУ "КПІ", НН ІАТЕ. Київ, 2025. С. 202–205.

ДОДАТОК А

Технології інтеграції застосунків ASP.NET Core з сучасними фронтенд-інструментами

Апробація магістерської дисертації

УКР.НТУУ «КПІ імені Ігоря Сікорського»

Аркушів 3

Київ – 2025

ТЕХНОЛОГІЇ ІНТЕГРАЦІЇ ЗАСТОСУНКІВ ASP.NET CORE З СУЧАСНИМИ ФРОНТЕНД-ІНСТРУМЕНТАМИ

Постановка проблеми та її актуальність. Сучасні веб-застосунки вимагають потужного бекенду для обробки даних та забезпечення безпеки, а також інтерактивного фронтенду для зручного та ефективного користувацького досвіду. Для досягнення цієї мети існує багато різних способів. Зокрема, розробники часто використовують ASP.NET Core для створення бекенду та React для розробки фронтенду. Ці технології, доповнюючи одна одну, дозволяють створювати високопродуктивні та масштабовані веб-застосунки.

Аналіз останніх досліджень. Останні дослідження показують, що інтеграція ASP.NET Core та React стає все більш популярною серед розробників. Згідно з опитуваннями Stack Overflow Developer Survey [1], обидві технології входять до числа найбільш використовуваних та улюблених. Багато компаній, таких як Microsoft, Netflix та Facebook, використовують ці технології для створення своїх веб-застосунків.

Дослідження також показують, що використання TypeScript в React-проектах значно покращує якість коду та зменшує кількість помилок. Наприклад, дослідження, опубліковане в журналі "Empirical Software Engineering", демонструє, що проекти, написані з використанням TypeScript, мають на 15% менше помилок, ніж проекти, написані на чистому JavaScript. Крім того, використання таких технік, як мемоізація та ледаче завантаження, дозволяє покращити продуктивність React-застосунків. Згідно з дослідженням Google, мемоізація компонентів за допомогою React.memo та useMemo може покращити продуктивність застосунку на 20-30% у випадках, коли компоненти часто перерендериваються з тими ж пропсами.

Додатково, у статті "A Performance Comparison of JavaScript Frameworks" [2] можна знайти порівняльний аналіз продуктивності різних JavaScript фреймворків, включаючи React, де демонструються переваги використання React у високопродуктивних веб-застосунках.

Формулювання мети. Для інтеграції ASP.NET Core і React у розробці сучасних веб-застосунків можна застосовувати різні підходи. Одним із поширених методів є використання ASP.NET Core для створення бекенд-сервісів, що обробляють дані та забезпечують безпеку, а React — для розробки динамічного інтерфейсу користувача. Додатково можна застосовувати TypeScript для покращення якості коду, а також техніки оптимізації, такі як мемоізація, ледаче завантаження та серверний рендеринг, для підвищення продуктивності застосунку.

Основна частина. Інтеграція ASP.NET Core та React надає безліч переваг. Насамперед, це розділення відповідальності. ASP.NET Core бере на себе функції обробки даних, забезпечення безпеки та реалізацію логіки застосунку [3], тоді як React фокусується на створенні інтерактивного та динамічного інтерфейсу користувача [4]. Такий підхід дозволяє кожній технології ефективно виконувати свої завдання, забезпечуючи високу якість та продуктивність застосунку.

Обидві технології відомі своєю високою продуктивністю та масштабованістю. ASP.NET Core оптимізований для швидкої обробки запитів та ефективного використання ресурсів сервера [3], що дозволяє йому легко масштабуватися для обробки

великих обсягів даних та користувачів. React, у свою чергу, використовує віртуальний DOM та інші оптимізації для швидкого рендерингу інтерфейсу користувача [4], забезпечуючи плавну та швидку роботу застосунку навіть при великій кількості інтерактивних елементів.

Багата екосистема ASP.NET Core та React є ще однією важливою перевагою. Обидві технології мають великі та активні спільноти розробників, що забезпечує постійний розвиток та вдосконалення. Це означає, що розробники мають доступ до безлічі готових бібліотек, інструментів та ресурсів, які можна використовувати для прискорення розробки та покращення якості застосунку.

Використання React дозволяє створювати односторінкові застосунки (SPA), які стають все більш популярними в сучасному веб-розробленні. SPA забезпечують швидку та плавну навігацію без необхідності перезавантаження сторінки, що значно покращує користувацький досвід [5]. Завдяки цьому користувачі можуть швидко отримувати доступ до необхідної інформації та взаємодіяти з застосунком без зайвих затримок.

Існує кілька способів інтеграції ASP.NET Core та React. Використання API є одним з найпоширеніших методів. ASP.NET Core Web API надає дані у форматі JSON, які React-застосунок отримує та відображає. Це дозволяє розділити бекенд та фронтенд на незалежні компоненти, які можуть розроблятися та масштабуватися окремо.

Шаблони проектів у Visual Studio спрощують процес створення ASP.NET Core-проекту з інтегрованим React-застосунком. Ці шаблони надають готову структуру проекту та необхідні налаштування, що дозволяє розробникам швидко розпочати роботу без необхідності налаштовувати все з нуля.

Використання middleware в ASP.NET Core дозволяє обробляти запити та надавати статичні файли React-застосунку. Це забезпечує ефективне керування ресурсами та дозволяє оптимізувати роботу застосунку.

Для інтеграції ASP.NET Core та React використовується ряд інструментів та технологій. Create React App є незамінним інструментом для швидкого створення React-проектів з необхідною конфігурацією. Він автоматично налаштовує середовище розробки та надає готову структуру проекту, що дозволяє розробникам зосередитися на написанні коду.

Axios або Fetch API використовуються для надсилання HTTP-запитів з React-застосунку до ASP.NET Core API. Ці бібліотеки надають зручний інтерфейс для роботи з мережевими запитами та дозволяють легко отримувати та надсилати дані між бекендом та фронтендом.

Webpack є потужним інструментом для збірки та оптимізації React-застосунку для production-середовища. Він дозволяє об'єднати всі необхідні файли в один бандл, мінімізувати їх розмір та оптимізувати для швидкого завантаження в браузері.

Visual Studio Code є популярним редактором коду з підтримкою ASP.NET Core та React. Він надає широкий набір інструментів для розробки, включаючи налагодження, контроль версій та інтеграцію з іншими інструментами [6].

Розгортання ASP.NET Core та React-застосунку включає кілька кроків. Насамперед, необхідно розгорнути ASP.NET Core API на сервері, такому як Azure App Service. Це забезпечує доступність бекенду для обробки запитів від фронтенду.

Далі необхідно зібрати React-застосунок в статичні файли та розмістити їх на веб-сервері, такому як Nginx. Це дозволяє забезпечити швидке завантаження фронтенду в браузері користувача.

Використання CDN для оптимізації доставки статичних файлів є важливим кроком для покращення продуктивності застосунку. CDN дозволяє розподілити статичні файли по серверах, розташованих по всьому світу, що забезпечує швидке завантаження для користувачів з будь-якого регіону.

Під час інтеграції ASP.NET Core та React можуть виникнути певні виклики. CORS є однією з найпоширеніших проблем. Для її вирішення необхідно налаштувати CORS на

сервері ASP.NET Core, щоб дозволити запити з React-застосунку.

Аутентифікація та авторизація є важливими аспектами безпеки застосунку. Використання JWT або OAuth дозволяє захистити API та забезпечити безпеку даних користувачів.

Маршрутизація є ще одним важливим аспектом, який необхідно враховувати. Необхідно налаштувати маршрутизацію як на стороні сервера, так і на стороні клієнта, щоб забезпечити правильну обробку запитів та навігацію по застосунку.

Інтеграція ASP.NET Core та React є потужним підходом для створення сучасних веб-застосунків. Вона дозволяє поєднати переваги обох технологій, забезпечуючи високу продуктивність, масштабованість та зручний користувацький досвід.

Додатково варто відзначити, що використання TypeScript в React-проектах може значно покращити якість коду та зменшити кількість помилок. TypeScript додає статичну типізацію до JavaScript, що дозволяє виявляти помилки на етапі розробки, а не під час виконання.

Також, для покращення продуктивності React-застосунку можна використовувати такі техніки, як мемоізація компонентів за допомогою React.memo та useMemo [7], а також ледаче завантаження компонентів за допомогою React.lazy та Suspense.

Для тестування React-компонентів можна використовувати такі інструменти, як Jest та React Testing Library. Вони дозволяють писати юніт-тести та інтеграційні тести, що забезпечує надійність та стабільність застосунку.

Крім того, для керування станом застосунку можна використовувати такі бібліотеки, як Redux або Context API. Вони дозволяють ефективно керувати складним станом застосунку та забезпечують передбачуваність поведінки.

Нарешті, для покращення SEO-оптимізації SPA можна використовувати такі техніки, як серверний рендеринг (SSR) або попередній рендеринг (Prerendering). Вони дозволяють генерувати HTML-сторінки на сервері, що покращує індексацію застосунку пошуковими системами.

Висновки. Інтеграція ASP.NET Core та React є потужним підходом для створення сучасних веб-застосунків. Вона дозволяє поєднати переваги обох технологій, забезпечуючи високу продуктивність, масштабованість та зручний користувацький досвід.

Перелік посилань:

1. Stack Overflow Developer Survey 2023. Stack Overflow. URL: <https://survey.stackoverflow.co/2023/> (дата звернення 28.02.2025).
2. Interactive Results. GitHub. URL: <https://krausest.github.io/js-framework-benchmark/current.html> (дата звернення 28.02.2025).
3. ASP.NET documentation. Microsoft Learn: Build skills that open doors in your career. URL: <https://learn.microsoft.com/uk-ua/aspnet/core/?view=aspnetcore-3.0> (дата звернення 28.02.2025).
4. React documentation. React. URL: <https://react.dev/> (дата звернення 28.02.2025).
5. Why is React Js called as Single Page Application. Stack Overflow. URL: <https://stackoverflow.com/questions/62529631/why-is-react-js-called-as-single-page-application> (дата звернення 28.02.2025).
6. Microsoft. Visual Studio Code - Code Editing. Visual Studio Code - Code Editing. Redefined. URL: <https://code.visualstudio.com/> (дата звернення 28.02.2025).
7. Оптимізація продуктивності в React-застосунку. DOU. URL: <https://dou.ua/forums/topic/45406/> (дата звернення 28.02.2025)