МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМ. С.З.ГЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: **“Розробка та дослідження інтерфейсів користувача у масштабованих бізнес-аплікаціях на основі мікрофронтенд-архітектури”**

Виконав: студент гр. Іт-61

Спеціальності 126 «Інформаційні системи та технології»
(шифр і назва)

Люліч Дмитро Іванович
(Прізвище та ініціали)

Керівник: к.т.н., доц. Лиса О.В.
(Прізвище та ініціали)

Рецензенти: д.т.н., проф. Власовець В.М.
(Прізвище та ініціали)

ЛЬВІВ-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМ. С.З.ГЖИЦЬКОГО
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Другий (магістерський) рівень вищої освіти
Спеціальність 126 «Інформаційні системи та технології»

"ЗАТВЕРДЖУЮ"
Завідувач кафедри_____
д.т.н., проф. А.М. Тригуба
"____"_____2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу студенту

Люліч Дмитро Іванович

1. Тема роботи: «Розробка та дослідження інтерфейсів користувача у масштабованих бізнес-аплікаціях на основі мікрофронтенд-архітектури»

Керівник роботи Лиса Ольга Володимирівна, к.т.н., доцент.
Затверджені наказом по університету від 28.02 2025 року № 140 /к-с.

2. Строк подання студентом роботи 05.12.2025 р.

3. Вихідні дані до роботи: *фронтенд-фреймворки та бібліотеки React, Vue.js та Angular, для управління станом бібліотеки Redux Toolkit (для React) та Pinia (для Vue.js), для візуалізації складних даних бібліотека D3.js.*

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

Вступ.
1. Теоретичні основи архітектури мікрофронтендів.
2. Проєктування та реалізація модульного інтерфейсу користувача з використанням мікрофронтендів.
3. Практична реалізація та експериментальна оцінка.
4. Охорона праці та безпека у надзвичайних ситуаціях.
5. Визначення ефективності від впровадження архітектури мікрофронтендів для бізнес-аплікацій.
Висновки та пропозиції.
Список використаної літератури

5. Перелік ілюстраційного матеріалу (з точним зазначенням обов'язкових слайдів): *Порівняння архітектур фронтенду; Схема технологічного стеку мікрофронтендів; Вимоги до інтерфейсів масштабованих бізнес-аплікацій; Схема взаємодії мікрофронтендів у бізнес-додатку; Схематичне відображення архітектури рішення; Вибір технологій і інструментів для реалізації мікрофронтендів; Технологічний стек для реалізації архітектури мікрофронтендів; Продуктивність при навантаженні (Users vs Response Time); Розроблений прототип для побудови складних бізнес-аплікацій; Розрахунок економічної ефективності.*

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1, 2, 3, 5	*Лиса О.В., доцент кафедри інформаційних технологій*		
4	*Городецький І.М., доцент кафедри інженерної механіки*		

7. Дата видачі завдання 1 березня 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Приміт-ка
1	*Написання першого розділу*	01.03.25-04.05.25	
2	*Виконання другого розділу та аркушів ілюстраційного матеріалу до нього*	05.05.25-14.07.25	
3.	*Виконання третього розділу та аркушів ілюстраційного матеріалу до нього*	15.07.25-24.09.25	
4.	*Написання розділу «Охорона праці та безпека у надзвичайних ситуаціях»*	25.09.25-10.10.25	
5.	*Оцінення ефективності запропонованої системи*	11.10.25-31.10.25	
6.	*Завершення оформлення розрахунково-пояснювальної записки та презентації*	01.11.25-30.11.25	
7.	*Завершення роботи в цілому*	01.12.25-05.12.25	

Студент _____ Люліч Д.І.
(підпис)

Керівник роботи _____ Лиса О.В.
(підпис)

УДК 004.42:004.738.5:004.415.2

Розробка та дослідження інтерфейсів користувача у масштабованих бізнес-аплікаціях на основі мікрофронтенд-архітектури. Люліч Д.І. Кафедра інформаційних технологій – Львів, ЛНУВМБ ім. С.З.Ґжицького, 2025. Кваліфікаційна робота: 78 с. текст. част., 6 рис., 27 табл., 10 арк. ілюстраційного матеріалу, 30 джерел.

У роботі розглянуто методологію та практичний підхід до створення масштабованих бізнес-аплікацій із модульним інтерфейсом користувача на основі архітектури мікрофронтендів. У вступі обґрунтовано актуальність теми, визначено мету, завдання, об’єкт і предмет дослідження.

У першому розділі досліджено сучасні підходи до розробки фронтенд-додатків, проаналізовано принципи та особливості мікрофронтенд-архітектури, розглянуто технологічні стеки для її реалізації та визначено вимоги до інтерфейсів користувача в масштабованих бізнес-аплікаціях. У другому розділі спроєктовано загальну архітектуру рішення на основі мікрофронтендів, обґрунтовано вибір інструментів і технологій, розроблено структуру модулів, механізми їхньої взаємодії та інтеграції в єдиний користувацький інтерфейс. У третьому розділі розроблено прототип бізнес-аплікації з використанням мікрофронтендів, проведено тестування масштабованості, продуктивності та зручності супроводу, а також здійснено порівняльний аналіз з традиційними фронтенд-архітектурами.

У четвертому розділі розглянуто питання охорони праці та безпеки у надзвичайних ситуаціях. У п’ятому розділі оцінено економічну та технічну ефективність впровадження мікрофронтенд-архітектури, проаналізовано вплив на витрати розробки, масштабованість і підтримуваність бізнес-аплікацій.

Ключові слова: мікрофронтенди, масштабовані бізнес-аплікації, архітектура програмного забезпечення, інтерфейс користувача, модульність, React, Module Federation, Webpack 5, TypeScript, Redux Toolkit, REST API, EventBus, продуктивність, масштабованість, DevOps, Docker, Kubernetes.

ЗМІСТ

ВСТУП

Сучасний ринок інформаційних технологій стрімко розвивається, і бізнес-аплікації все частіше стають масштабними та комплексними. Зростає кількість користувачів, функціональних модулів і інтеграцій з різними системами, що призводить до ускладнення архітектури фронтенду. Традиційні підходи до розробки інтерфейсів користувача, такі як монолітні SPA-додатки, стають менш ефективними через труднощі масштабування, підтримки коду та впровадження нових функцій без порушення стабільності системи. Бізнес-аплікації стають багатомодульними, інтегрованими з різними сервісами та платформами, що створює додаткові виклики для розробників у забезпеченні масштабованості, надійності та продуктивності.

Традиційні фронтенд-архітектури, такі як монолітні SPA-додатки або класичні веб-додатки, часто виявляються недостатньо ефективними для великих і динамічних проектів. Основні проблеми включають: складність масштабування при збільшенні кількості функціональних модулів; ускладнення підтримки коду через залежності між компонентами; труднощі одночасної роботи декількох команд розробників; високий ризик порушення стабільності системи при внесенні змін у код.

Архітектура мікрофронтендів дозволяє розбивати великий фронтенд на незалежні, автономні модулі, які можуть розроблятися та підтримуватися окремими командами, при цьому забезпечуючи єдиний користувацький інтерфейс. Такий підхід підвищує гнучкість, прискорює впровадження нових функцій і зменшує ризики при масштабуванні системи. Особливо актуальним цей підхід є для корпоративних систем, SaaS-платформ та електронної комерції, де кожен модуль має специфічну функціональність, а масштабованість і швидка інтеграція нових компонентів є критичними для успіху. Таким чином, дослідження застосування архітектури мікрофронтенду для розробки бізнес-аплікацій є актуальним і має значний практичний та науковий інтерес.

Актуальність дослідження підтверджується практикою провідних IT-компаній та стартапів, де мікрофронтенди застосовуються для корпоративних систем, SaaS-платформ, онлайн-магазинів і фінтех-додатків, що мають високі вимоги до масштабованості та продуктивності.

Мета роботи – розробити і обґрунтувати підхід до створення масштабованих бізнес-аплікацій на основі архітектури мікрофронтендів, який забезпечує модульність, автономність розробки та ефективну інтеграцію компонентів інтерфейсу користувача.

Об’єкт дослідження – процес розробки масштабованих бізнес-аплікацій із складними інтерфейсами користувача.

Предмет дослідження – архітектура мікрофронтендів як інструмент модульної організації фронтенд-компонентів та її застосування для підвищення масштабованості, гнучкості і підтримуваності бізнес-аплікацій.

Наукова новизна дослідження полягає у комплексному аналізі і застосуванні архітектури мікрофронтендів для бізнес-аплікацій, а також у формуванні методології модульного проектування інтерфейсів користувача з урахуванням вимог масштабованості та автономності команд розробки.

Практична значимість роботи полягає у створенні рекомендацій для розробників щодо впровадження мікрофронтенд-архітектури у корпоративних і SaaS-додатках; підвищенні швидкості розробки і гнучкості інтеграції нових модулів; зниженні ризиків при масштабуванні фронтенд-додатків та забезпеченні стабільності роботи системи; наданні прикладу реалізації прототипу, що може бути використаний як шаблон для практичних проектів.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ АРХІТЕКТУРИ МІКРОФРОНТЕНДІВ

1.1. Основні підходи до розробки фронтенд-додатків

Фронтенд-архітектура визначає структуру та організацію користувацького інтерфейсу веб-додатків або бізнес-аплікацій, а також принципи взаємодії компонентів і модулів, що його формують. Ефективна архітектура фронтенду забезпечує стабільність, масштабованість, продуктивність і гнучкість системи, полегшує підтримку та розвиток додатку у довгостроковій перспективі.

Історично розвиток фронтенд-архітектур проходив кілька етапів:

1. Класичні веб-сторінки (Static Web Pages, 1990–2005 рр.). На ранніх етапах розвитку Інтернету користувацькі інтерфейси формувалися статичними HTML-сторінками. Усі зміни на сайті вимагали перезавантаження сторінки, а інтерактивність обмежувалася простими формами та скриптами на JavaScript. Масштабування таких систем було складним через відсутність модульності та централізованого контролю логіки.

2. Монолітні динамічні додатки (Monolithic Web Applications, 2005–2010 рр.). З появою серверних мов програмування (PHP, Java, .NET) з’явилися динамічні веб-додатки, де фронтенд та бізнес-логіка тісно інтегровані на сервері. Такі рішення дозволяли автоматизувати генерацію контенту і частково інтерактивність, проте масштабування залишалося проблемним: будь-які зміни у фронтенді потребували редизайну всієї системи.

3. Односторінкові додатки (Single Page Applications, SPA, 2010–2015 рр.). Поява SPA на базі JavaScript-фреймворків (AngularJS, Backbone.js, Ember.js) дозволила переносити значну частину логіки на клієнтську сторону. Додаток завантажувався один раз, а подальші зміни контенту відбувалися без перезавантаження сторінки. SPA підвищили інтерактивність, але із збільшенням складності проекту зросли проблеми підтримки коду та масштабування, особливо при роботі великих команд розробників.

4. Модульні фронтенди та компоненти (Component-Based Architecture, 2015–2020 рр.). У відповідь на проблеми SPA з’явився підхід компонентного програмування. Фреймворки React, Vue.js та Angular почали підтримувати створення ізольованих, повторно використовуваних компонентів. Компонентний підхід полегшував розробку і тестування окремих частин інтерфейсу, але при великих проєктах все ще виникали труднощі з інтеграцією та масштабуванням через центральну залежність від основного додатку.

5. Архітектура мікрофронтендів (Micro-Frontends, 2020–тепер) Мікрофронтенди є еволюцією компонентного підходу. Основна ідея полягає у розбитті фронтенд-додатку на автономні модулі (мікрофронтенди), кожен із яких відповідає за окремий бізнес-функціонал і може розроблятися, тестуватися та деплоїтися незалежно. Ключові принципи архітектури:

- Автономність модулів – кожен мікрофронтенд має власну логіку, стилі та маршрутизацію.
- Повторне використання компонентів – окремі модулі можуть бути інтегровані в різні частини додатка або в різні проекти.
- Незалежна розробка команд – дозволяє паралельно працювати над різними модулями без конфліктів.
- Інтеграція у єдиний інтерфейс – через контейнер або shell, який забезпечує маршрутизацію та об’єднує мікрофронтенди в цілісний додаток.

Поява мікрофронтендів була зумовлена потребою масштабування великих бізнес-додатків, де швидкість розробки, незалежність команд і гнучкість інтеграції нових функцій є критичними для успіху проекту. Сьогодні цей підхід активно застосовується у корпоративних системах, SaaS-платформах та великих комерційних веб-додатках.

Сучасні веб-додатки і бізнес-аплікації розробляються із застосуванням різних архітектурних підходів, що відрізняються організацією коду, масштабованістю та принципами інтеграції компонентів. Основні підходи можна класифікувати наступним чином:

1. Монолітний фронтенд (Monolithic Frontend). У монолітному підході весь фронтенд-додаток є єдиною системою, де всі компоненти та модулі інтегровані без поділу на автономні частини. Основні характеристики: всі функціональні блоки тісно пов’язані між собою; підтримка і внесення змін потребують координації всіх команд розробників; масштабування системи обмежене через складність інтеграції нових модулів. Монолітні фронтенди ефективні для невеликих проєктів або систем із стабільною функціональністю, проте вони виявляються неефективними для великих бізнес-додатків через високі витрати на підтримку та низьку гнучкість.

2. Односторінкові додатки (SPA, Single Page Applications). SPA-додатки завантажуються один раз і динамічно оновлюють контент без перезавантаження сторінки. Основні особливості: використання сучасних JavaScript-фреймворків (React, Angular, Vue.js); підвищена інтерактивність та покращений користувацький досвід; частина логіки переміщена на клієнтську сторону, що знижує навантаження на сервер; складність масштабування при розширенні додатку та збільшенні команд розробки. SPA-додатки дозволяють покращити швидкість взаємодії з користувачем, однак великі проєкти зростають у складності та стають проблемними для підтримки.

3. Компонентний підхід (Component-Based Architecture) Компонентний підхід базується на розбитті інтерфейсу на окремі, незалежні компоненти, що можуть повторно використовуватися у різних частинах додатку. Кожен компонент ізольований і має власні стилі, логіку та події. Знижується взаємозалежність між командами. Підвищується тестованість і повторне використання коду.Незважаючи на переваги, компонентна архітектура потребує централізованого контролю інтеграції та не вирішує повністю проблеми масштабування великих бізнес-додатків.

4. Мікрофронтенди (Micro-Frontends) - це сучасний підхід, який дозволяє поєднати переваги компонентного програмування з модульною автономністю команд розробки. Основні принципи та переваги: модульність та автономність: кожен мікрофронтенд відповідає за окремий бізнес-функціонал; незалежне

деплоювання: модулі можна оновлювати без впливу на весь додаток; паралельна розробка: кілька команд можуть одночасно працювати над різними частинами фронтенду; масштабованість та гнучкість: нові функції інтегруються швидко без великих змін у кодовій базі; підвищена підтримуваність: легше тестувати та підтримувати окремі модулі. Мікрофронтенди є логічним продовженням еволюції фронтенд-архітектур, особливо актуальним для великих бізнес-додатків та корпоративних SaaS-рішень.

1.2. Архітектура мікрофронтендів

Архітектура мікрофронтендів – це підхід до організації фронтенд-додатку, при якому велика система розбивається на низку автономних модулів, що об’єднуються у єдиний користувацький інтерфейс. Кожен модуль (мікрофронтенд) може: розроблятися окремою командою; використовувати власний стек технологій; самостійно проходити цикл тестування і деплою; бути інтегрованим у спільний інтерфейс через контейнер або shell.

Архітектура мікрофронтендів є логічним продовженням еволюції веб-додатків від монолітних структур до розподілених систем, які орієнтовані на масштабування, незалежність і ефективну підтримку. Її головна ідея полягає у поділі фронтенд-частини на окремі самостійні модулі, кожен із яких відповідає за певну функціональну зону, має власну логіку, дизайн і цикл життєдіяльності. Це дає змогу створювати складні інтерфейси користувача, які легко розвивати, тестувати і розгортати без ризику порушення роботи всієї системи.

Основним принципом мікрофронтенд-архітектури є автономність модулів. Кожен мікрофронтенд управляє власним станом, маршрутами, стилями, уникаючи надмірних залежностей від інших частин системи. Це створює умови для незалежного розгортання й оновлення, дозволяє командам працювати у власному темпі, не блокуючи загальний процес розробки. Незалежний життєвий цикл модулів дає можливість вносити зміни, тестувати і випускати нові версії без

зупинки або ризику для інших елементів системи, що особливо важливо для бізнес-додатків з високими вимогами до безперервності роботи.

Розділення відповідальності у мікрофронтендах реалізується через чітке визначення меж між модулями: кожен компонент виконує власну бізнес-функцію — управління користувачами, обробка замовлень, аналітика, налаштування тощо. Це забезпечує логічну структурованість системи та сприяє її зрозумілості й керованості. При цьому модулі можуть повторно використовуватися у різних частинах проєкту або навіть у зовнішніх системах, що підвищує ефективність розробки і знижує витрати на створення нових рішень.

Незважаючи на децентралізований характер, усі мікрофронтенди інтегруються в єдиний користувацький інтерфейс, який зберігає послідовність UX-дизайну. Це досягається завдяки використанню shell-додатку (контейнера), який координує завантаження модулів, маршрутизацію сторінок і комунікацію між ними. Shell забезпечує стабільну навігацію, спільні стилі, автентифікацію та управління станом на рівні всієї аплікації.

Переваги використання мікрофронтендів проявляються на всіх етапах життєвого циклу додатка. Масштабованість системи дозволяє швидко додавати нові модулі без ризику порушення стабільності існуючих компонентів. Гнучкість технологій відкриває можливість застосовувати різні фреймворки та інструменти у межах одного проєкту (наприклад, React для одних модулів і Vue або Angular для інших), що робить архітектуру технологічно нейтральною. Завдяки цьому команди можуть використовувати найефективніші рішення під конкретні задачі.

Паралельна робота кількох команд над різними мікрофронтендами суттєво прискорює розробку, адже зникає потреба у централізованій координації кожного кроку. Підвищується стабільність, оскільки збій в одному модулі не впливає на функціонування всієї системи. Крім того, тестування і підтримка стають значно простішими — модулі можна перевіряти окремо, автоматизувати деплой і CI/CD процеси, мінімізуючи людський фактор.

Схема інтеграції мікрофронтендів передбачає наявність трьох ключових елементів: контейнера (Shell Application), який відповідає за маршрутизацію і

рендеринг; мікрофронтенд-модулів, що реалізують функціональні частини інтерфейсу; та комунікаційного шару, який забезпечує взаємодію між модулями через API, події або спільний стан. Ці компоненти взаємодіють за допомогою централізованого шини подій (EventBus), REST або GraphQL API, а також інструментів обміну станом (наприклад, Redux або RxJS).

Таким чином, архітектура мікрофронтендів поєднує модульність компонентного підходу з гнучкістю масштабованих систем. Вона створює ефективну основу для розробки великих бізнес-аплікацій, що потребують швидких оновлень, високої доступності та незалежності окремих частин. Завдяки цьому підходу компанії отримують не лише технічну стабільність і продуктивність, а й стратегічну перевагу — можливість швидко адаптуватися до змін ринку, зберігаючи безперервність розвитку своїх цифрових продуктів.

Для кращого розуміння переваг і обмежень архітектури мікрофронтендів доцільно порівняти її з традиційними підходами: монолітними та SPA-додатками. Основні характеристики наведено у таблиці 1.1.

Таблиця 1.1

Порівняння архітектур фронтенду

Параметр	Монолітний фронтенд	SPA (Single Page Application)	Мікрофронтенди
Структура коду	Єдина кодова база	Код розділений на компоненти, але централізований	Автономні модулі з власною логікою і стилями
Масштабованість	Обмежена	Середня, складно при великій кількості команд	Висока, нові модулі додаються незалежно
Паралельна розробка	Складна	Можлива, але конфлікти часті	Легко, команди працюють над окремими модулями
Деплой та оновлення	Центральний деплой	Частковий деплой через SPA-фреймворки	Незалежний деплой кожного модуля
Технологічний стек	Один стек для всього додатку	Один або обмежений стек	Кожен модуль може використовувати власний стек
Стабільність при оновленнях	Вразлива, помилки впливають на весь додаток	Вища, але все ще залежить від централізованого коду	Висока, помилки в одному модулі не впливають на інші
Підтримка і тестування	Складна	Полегшена, тестування компонентів	Легка, модульне тестування кожного модуля окремо

Як видно з таблиці, мікрофронтенди поєднують переваги SPA та компонентного підходу, вирішуючи проблеми масштабування і підтримки великих бізнес-додатків.

1.3. Технологічний стек для реалізації мікрофронтендів

Реалізація архітектури мікрофронтендів вимагає використання сучасного технологічного стеку, що забезпечує створення автономних модулів, їхню ізольовану розробку, незалежне оновлення та гнучку інтеграцію в єдиний інтерфейс користувача. На відміну від монолітних фронтендів, де всі частини системи щільно пов’язані між собою, мікрофронтенди дозволяють розподілити відповідальність між командами, що розробляють окремі модулі, та підвищують масштабованість і гнучкість проєкту.

Основу технологічного стеку складають фреймворки, інструменти маршрутизації, комунікації між модулями та засоби автоматизації розгортання. Серед основних фреймворків, що підтримують підхід мікрофронтендів, вирізняються React.js, Angular, Vue.js та Svelte. React.js став найпопулярнішим завдяки підтримці Module Federation, який дозволяє підключати незалежні модулі під час виконання. Angular, у свою чергу, пропонує цілісну архітектуру з інтегрованими механізмами модульності, що полегшує керування великими застосунками. Vue.js використовується для легких, високопродуктивних компонентів, а Svelte відрізняється підходом до компіляції компонентів на етапі збірки, що робить його ідеальним для швидких і компактних рішень.

Інтеграція модулів у спільний інтерфейс реалізується завдяки інструментам, таким як Webpack Module Federation, Single SPA або Web Components. Module Federation дає змогу динамічно підключати мікрофронтенди з різних репозиторіїв і навіть оновлювати їх без перезбірки всієї системи. Single SPA забезпечує узгоджену маршрутизацію між модулями, створеними на різних фреймворках (React, Angular, Vue), а Web Components гарантують технологічну сумісність і

стандартизований спосіб інкапсуляції стилів та логіки через Shadow DOM і Custom Elements.

Взаємодія між мікрофронтендами — один із найскладніших аспектів такої архітектури. Для цього застосовуються різні підходи: Event Bus або Pub/Sub моделі для асинхронної передачі подій, Redux, RxJS або Vuex для централізованого управління станом, а також API Gateway для синхронізації даних із бекендом і забезпечення єдиної точки доступу до сервісів. Це дозволяє підтримувати узгодженість стану між різними частинами застосунку без втрати незалежності модулів.

Надійність і якість коду забезпечуються засобами тестування, такими як Jest, Cypress або Testing Library, які дозволяють перевіряти як окремі компоненти, так і інтеграційні сценарії взаємодії між ними. Для забезпечення ефективного процесу доставки змін використовуються Docker і Kubernetes, що дозволяють контейнеризувати кожен мікрофронтенд і масштабувати його окремо, а також системи GitLab CI або GitHub Actions, які автоматизують процес збірки, тестування та розгортання.

Таким чином, мікрофронтенд-архітектура базується на поєднанні інноваційних фреймворків, гнучких механізмів інтеграції та сучасних інструментів DevOps. Вона створює основу для масштабованих, стійких і легко підтримуваних бізнес-аплікацій, у яких кожен модуль є самодостатнім, але водночас інтегрованим елементом єдиної системи.

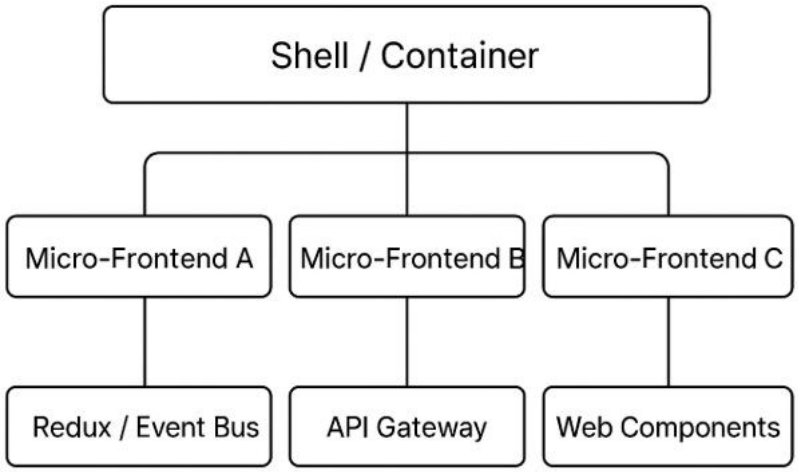


Рис.1.1. Схема технологічного стеку мікрофронтендів (приклад інтеграції)

На рисунку 1.1 подано узагальнену структуру взаємодії між основними складовими архітектури мікрофронтендів у бізнес-додатку.

У центрі схеми розташовано Shell (або Container) — це головний контейнер, який відповідає за завантаження, ініціалізацію та координацію роботи окремих мікрофронтендів. Він виступає своєрідною «рамкою» або оболонкою, у межах якої об’єднуються незалежні модулі інтерфейсу користувача. Shell керує маршрутизацією, загальними стилями, автентифікацією та передачею контексту між компонентами.

Під контейнером розміщено три автономні модулі — Micro-Frontend A, Micro-Frontend B та Micro-Frontend C. Кожен із них є самостійним застосунком, який може бути розроблений на різних фреймворках (наприклад, React, Angular або Vue), мати власну логіку, систему керування станом та інтерфейс. Це забезпечує незалежність розробки та спрощує масштабування: команди можуть розвивати й оновлювати свої модулі без ризику порушення роботи інших частин системи.

У нижній частині схеми зображено основні технологічні механізми, що забезпечують взаємодію між модулями:

- Redux / Event Bus — використовується для спільного керування станом додатку та обміну подіями між мікрофронтендами. Завдяки цьому забезпечується синхронізація даних, наприклад, між панеллю управління та аналітичним модулем.
- API Gateway — виступає проміжною ланкою між фронтендами та бекендом. Він відповідає за маршрутизацію запитів, безпеку, кешування та централізований доступ до серверних ресурсів.
- Web Components — забезпечують можливість повторного використання UI-елементів у різних мікрофронтендах, незалежно від технології, на якій вони реалізовані. Це спрощує інтеграцію різнорідних модулів у єдиний інтерфейс.

Рисунок ілюструє принцип модульної інтеграції у мікрофронтендній архітектурі, де Shell виступає координатором, а окремі мікрофронтенди — незалежними функціональними частинами системи. Завдяки таким механізмам, як

Redux, Event Bus, API Gateway і Web Components, досягається масштабованість, гнучкість та незалежність оновлення окремих частин бізнес-додатку.

Отже, ефективне впровадження мікрофронтендів вимагає комплексного підходу - від модульного проєктування компонентів до використання сучасних інструментів інтеграції, комунікації та деплою. Цей стек забезпечує автономність розробки, гнучкість, масштабованість та високу стабільність бізнес-аплікацій.

Реалізація архітектури мікрофронтендів потребує системного підходу до проєктування, впровадження та підтримки, тому важливо дотримуватися низки рекомендацій, які забезпечують стабільність, масштабованість і гнучкість системи. Передусім варто проводити декомпозицію інтерфейсу за бізнес-функціями, де кожен мікрофронтенд відповідає за конкретний напрям, наприклад, каталог товарів, модуль аналітики чи управління замовленнями. Такий поділ дозволяє розробникам ізолювати зони відповідальності та мінімізувати залежності між модулями, що спрощує оновлення та підтримку кожного з них без ризику порушення роботи інших частин системи.

Не менш важливим є правильний вибір технологічного стеку для кожного модуля. Команди можуть працювати з різними фреймворками — React, Angular, Vue чи Svelte, — але при цьому необхідно дотримуватися стандартів інтеграції, таких як Web Components або Module Federation, щоб забезпечити сумісність та стабільну взаємодію модулів у межах спільної системи. Кожен мікрофронтенд має бути максимально автономним: мати власну логіку, маршрути та стан, а також використовувати ізоляцію через Shadow DOM для уникнення конфліктів стилів і забезпечення незалежності від контейнера.

Інтеграція модулів здійснюється за допомогою комунікаційного шару, який може бути реалізований через Event Bus, централізовані сховища стану (Redux, Vuex, RxJS) або подієву взаємодію між компонентами. Контейнер (Shell Application) відповідає за маршрутизацію, керування життєвим циклом і динамічне завантаження мікрофронтендів, забезпечуючи при цьому цілісність користувацького досвіду (UX).

З погляду DevOps-практик, кожен мікрофронтенд повинен мати окремий цикл CI/CD — власну збірку, тестування, деплой і моніторинг. Це підвищує стабільність системи й дозволяє швидко вносити зміни без ризику глобальних збоїв. Рекомендується застосовувати різнорівневе тестування: юніт-тести для перевірки компонентів, інтеграційні — для взаємодії модулів, та end-to-end — для оцінки роботи всієї системи.

Ключовою перевагою такої архітектури є її масштабованість: можна легко додавати нові модулі, оновлювати технології або змінювати команди розробників без необхідності перебудови всієї системи. Також доцільно впроваджувати механізми повторного використання бібліотек, стилів і спільних компонентів, що оптимізує час розробки і підтримує узгоджений стиль користувацького інтерфейсу. Таким чином, наведені принципи та рекомендації формують основу стабільної, гнучкої і технологічно сучасної архітектури мікрофронтендів, яка відповідає вимогам масштабованих бізнес-аплікацій.

Мікрофронтенди дозволяють розбивати додаток на автономні модулі з незалежним життєвим циклом, забезпечують гнучкість, швидку інтеграцію та повторне використання компонентів. Впровадження мікрофронтендів потребує використання сучасних технологій, таких як React, Angular, Vue, Svelte, Module Federation, Web Components, а також автоматизації CI/CD та тестування.

Рекомендована стратегія побудови мікрофронтендів включає декомпозицію додатку за бізнес-функціями, автономність модулів, ефективну комунікацію та інтеграцію через контейнер, що забезпечує високу масштабованість і стабільність бізнес-аплікацій.

1.4. Характеристика масштабованих бізнес-аплікацій

Масштабовані бізнес-аплікації є комплексними програмними системами, що підтримують різноманітні бізнес-процеси організації, інтегрують великі обсяги даних, працюють із багатьма користувачами та системами одночасно. Вони розробляються для забезпечення ефективності управління, автоматизації робочих

процесів і прийняття рішень на основі аналітики. Масштабовані бізнес-аплікації — це складні багаторівневі системи, які поєднують десятки або навіть сотні функціональних модулів, кожен з яких реалізує окремі бізнес-процеси: управління фінансами, продажами, логістикою, персоналом, клієнтськими відносинами чи аналітикою. Кожен з цих модулів має власну логіку, інтерфейс та вимоги до продуктивності, тому для побудови таких систем потрібен гнучкий модульний підхід, який дозволяє ефективно розподіляти функціональні зони та підтримувати масштабування в майбутньому.

Однією з ключових особливостей таких аплікацій є здатність підтримувати велику кількість користувачів, які можуть одночасно взаємодіяти із системою. Це вимагає стабільної роботи серверної частини, швидкого відгуку інтерфейсу та продуманої системи ролей і доступів, що визначає, які саме функції або модулі будуть доступні для конкретного користувача. Рівень навантаження в таких додатках може сягати мільйонів активних сесій, тому кожен елемент системи має бути оптимізований під масштабне використання.

Масштабованість тут розглядається не лише як здатність системи обслуговувати більше користувачів, а й як можливість розширення її функціональності. Нові модулі або сервіси повинні легко інтегруватися в існуючу архітектуру без збоїв у роботі інших частин. Це досягається завдяки використанню архітектурних патернів, які підтримують незалежний деплой, розподілену розробку та паралельне оновлення компонентів.

Важливим аспектом є інтеграція з зовнішніми системами — ERP, CRM, бухгалтерськими сервісами, платіжними шлюзами чи сторонніми API. Масштабована бізнес-аплікація має вміти об’єднувати дані з різних джерел у єдиний інтерфейс, забезпечуючи узгоджене представлення інформації без втрати контексту чи порушення логіки бізнес-процесів. Крім того, такі системи зазвичай включають розвинені засоби аналітики — інтерактивні панелі керування, графіки, дашборди та звіти. Користувацький інтерфейс має забезпечувати швидкий доступ до даних, адаптивне відображення та можливість взаємодії з візуальними елементами для прийняття управлінських рішень у режимі реального часу.

Адаптивність і кросплатформність також є невід’ємними характеристиками сучасних бізнес-додатків. Вони повинні коректно працювати на різних пристроях — від настільних комп’ютерів до мобільних телефонів, автоматично підлаштовуючись під розміри екрана, тип пристрою та сценарій використання.

Висока надійність і безпека — ще один обов’язковий критерій. Навіть при відмові окремих модулів система має залишатися працездатною, забезпечуючи безперервний доступ користувачів до критичних функцій. Для цього реалізуються механізми резервування, автентифікації, авторизації та шифрування даних, а також інтелектуальна обробка помилок, що гарантує стабільність і захист інформації.

Зручність та інтуїтивність інтерфейсу визначають успішність використання системи. Навіть найскладніші бізнес-процеси повинні бути подані просто й логічно, щоб користувач швидко орієнтувався в інтерфейсі, мав доступ до необхідних функцій і міг персоналізувати робоче середовище відповідно до своїх ролей і потреб. Масштабовані бізнес-аплікації поєднують технічну складність, високу надійність і орієнтацію на зручність користувача, що робить їх основою сучасної цифрової трансформації бізнесу.

Таблиця 1.2

Вимоги до інтерфейсів масштабованих бізнес-аплікацій

Параметр	Вимога	Пояснення
Функціональна складність	Модульність	Розбиття інтерфейсу на автономні модулі, що відображають бізнес-функції
Кількість користувачів	Масштабованість	Забезпечення стабільної роботи при тисячах одночасних користувачів
Інтеграція	Сумісність з API та зовнішніми системами	Узгоджене відображення даних з різних джерел
Аналітика	Інтерактивні дашборди та графіки	Динамічне та зрозуміле представлення великих обсягів даних
Адаптивність	Кросплатформність	Підтримка різних пристроїв та розмірів екранів
Надійність	Відновлення після збоїв	Помилки одного модуля не порушують роботу всього додатку
Безпека	Автентифікація та авторизація	Захист доступу до конфіденційних даних
Зручність UX	Інтуїтивність	Проста навігація, мінімізація навчання користувачів

Вимоги до інтерфейсів масштабованих бізнес-аплікацій можна узагальнити у таблиці 1.2. Масштабовані бізнес-аплікації відрізняються високими вимогами до інтерфейсів користувача, оскільки саме вони визначають ефективність взаємодії користувача з системою та швидкість виконання бізнес-процесів. Традиційні монолітні та SPA-архітектури часто не справляються із забезпеченням цих вимог на великих проектах, що обумовлює актуальність використання архітектури мікрофронтендів.

Масштабування інтерфейсів користувача у великих бізнес-додатках є однією з найбільш складних задач фронтенд-розробки. Традиційні архітектури, такі як монолітні фронтенди та SPA-додатки, часто не здатні забезпечити достатню гнучкість, автономність і швидкість розробки при рості системи.

Основні проблеми масштабування інтерфейсів у сучасних веб-додатках пов’язані з централізацією коду, відсутністю модульності та складністю управління великими проєктами. У традиційних монолітних фронтендах усі компоненти зібрані в одну кодову базу, де будь-яка зміна або оновлення в окремій частині може вплинути на інші модулі. Це створює серйозні труднощі при додаванні нових функцій, адже навіть незначне оновлення інтерфейсу може призвести до неочікуваних помилок у інших частинах програми. У таких умовах підтримка великого проєкту вимагає значних ресурсів, а координація роботи декількох команд стає складним завданням.

Ще однією поширеною проблемою є обмежена масштабованість SPA-додатків (Single Page Application). Хоча вони забезпечують високу інтерактивність і швидке оновлення контенту, зі зростанням кількості компонентів виникають труднощі з управлінням станом, маршрутизацією та взаємодією між частинами інтерфейсу. У великих системах це призводить до збільшення залежностей між компонентами, що ускладнює інтеграцію нових функцій, уповільнює розробку та унеможливлює незалежний деплой. Коли над одним додатком працюють кілька команд, це часто створює конфлікти в коді та проблеми зі стабільністю.

Відсутність автономності модулів також є критичною перешкодою. У традиційних підходах зміна логіки або структури одного функціонального блоку

часто вимагає оновлення інших частин системи, що тягне за собою додаткові витрати часу на тестування, узгодження API та розгортання. У результаті, навіть незначні зміни в бізнес-процесах можуть призвести до повного циклу оновлення всього додатку, що сповільнює інновації.

Серйозною проблемою лишається і складність інтеграції з бекендом. Великі бізнес-додатки зазвичай взаємодіють із численними зовнішніми системами — ERP, CRM, платіжними шлюзами або API сервісів третіх сторін. Якщо фронтенд не має чітко визначеної модульної структури, керування цими даними стає хаотичним. Виникають труднощі з узгодженістю інформації, синхронізацією даних і забезпеченням стабільного обміну між інтерфейсом і сервером.

Проблема підтримки та тестування у великих SPA ще більш загострюється. Оскільки всі частини коду тісно пов’язані, тестування окремих компонентів або сценаріїв вимагає значних зусиль. Помилка в одному модулі може спричинити збій у всьому додатку, знижуючи стабільність і довговічність системи. Розробники витрачають багато часу на регресійне тестування, тоді як у модульних архітектурах ці процеси могли б бути ізольованими.

Окремо варто згадати про продуктивність. Великі кодові бази, що завантажуються як єдина сторінка, створюють суттєве навантаження на браузер, особливо при великій кількості користувачів. Довгі часи рендерингу, затримки в оновленні інтерфейсу, зниження FPS під час взаємодії з елементами — усе це негативно впливає на досвід користувача. Зі зростанням кількості запитів та одночасних користувачів система втрачає стабільність, а сервери потребують додаткових ресурсів для підтримки роботи.

Таким чином, основні проблеми масштабування інтерфейсів полягають у відсутності модульності, централізованій архітектурі, складності тестування та обмеженій продуктивності при зростанні навантаження. Саме тому сучасні компанії переходять до мікрофронтенд-архітектури, що дозволяє подолати ці бар’єри, забезпечивши автономність, незалежність життєвого циклу модулів і стабільність при розвитку системи.

Обмеження традиційних підходів подано у таблиці 1.3.

Таблиця 1.3

Обмеження традиційних підходів

Архітектура	Основні обмеження
Монолітний фронтенд	Важко масштабувати, складна інтеграція нових функцій, низька автономність модулів, висока залежність команд розробки
SPA	Збільшення складності з ростом системи, труднощі в управлінні станом, конфлікти при паралельній розробці, обмежена масштабованість команд
Компонентний підхід	Поліпшує модульність, але не забезпечує повної автономності та незалежного деплою, інтеграція модулів все ще централізована

Отже, основною проблемою масштабування інтерфейсів у великих бізнес-додатках є недостатня автономність модулів та централізація коду. Традиційні монолітні та SPA-архітектури не завжди дозволяють паралельну роботу численних команд розробників, що призводить до уповільнення процесу розвитку та оновлення системи. Висока складність інтеграції, управління станом та комунікації між компонентами обмежує швидкість внесення змін і зменшує стабільність інтерфейсу. Тому є необхідно застосувати модульний підхід, який забезпечує автономність, масштабованість і гнучкість – тобто архітектуру мікрофронтендів. Отже, застосування мікрофронтендів дозволяє вирішити проблеми масштабування, підвищити продуктивність команд розробки та забезпечити стабільність великих бізнес-аплікацій із складною логікою інтерфейсу користувача.

1.5. Вимоги до інтерфейсів користувача та обґрунтування застосування мікрофронтендів

У сучасних масштабованих бізнес-аплікаціях інтерфейс користувача відіграє ключову роль, оскільки він безпосередньо впливає на ефективність виконання бізнес-процесів, швидкість навчання користувачів та продуктивність роботи.

Інтерфейс користувача в сучасних програмних системах має бути побудований так, щоб окремі частини працювали автономно, але при цьому залишалися частинами єдиного цілого. Тому інтерфейс зазвичай складають із модулів, де кожен модуль відповідає за певну функціональність — наприклад, за перегляд аналітики, керування товарами чи оформлення замовлень. Така модульність дозволяє командам працювати паралельно, ізолює помилки й дає

змогу розвивати продукт без «розвалу» цілого застосунку. Важливо, щоб інтерфейс легко адаптувався до нових бізнес-вимог, адже процеси постійно змінюються, а компанії часто додають нові функції. Гнучкість дає можливість розширювати систему, додаючи нові екрани, компоненти чи інтеграції так, щоб вони не порушували вже існуючу логіку.

Не менш значущою є надійність: збої в одному компоненті не повинні призводити до падіння всієї системи, тому інтерфейс будують таким чином, щоб кожен елемент мав власний стан, чіткі межі відповідальності та мінімум залежностей. Важливу роль відіграє також інтеграція з зовнішніми сервісами — ERP, CRM, платіжними системами. Інтерфейс повинен коректно відображати отримані дані, швидко оновлювати інформацію й формувати єдиний інформаційний простір, щоб користувач бачив узгоджені та достовірні дані.

Усе це поєднується з потребою забезпечити інтерфейс, який однаково зручний на будь-якому пристрої — від великого монітора до смартфона. Адаптивність та інтерактивність гарантують, що користувач отримає швидкий доступ до потрібних інструментів, дашбордів чи звітів без зайвих дій. Нарешті, велике значення має простота подальшої підтримки: інтерфейс має бути побудований так, щоб його можна було легко тестувати, оновлювати, виправляти та розгортати частинами, без ризику порушити роботу системи. Це зменшує навантаження на команди, скорочує час на інтеграційні тести та підвищує стабільність програмного продукту під час активного розвитку.

На рисунку 1.2 зображено схему взаємодії мікрофронтендів у бізнес-додатку, яка ілюструє модульну архітектуру з централізованим контейнером (Shell / Container) та кількома незалежними мікрофронтендами (MFE).

У верхній частині схеми розташований Shell / Container — це головна оболонка системи, яка відповідає за завантаження та координацію роботи окремих мікрофронтендів. Вона забезпечує спільну навігацію, аутентифікацію, управління станом і маршрутизацію між модулями.

Нижче контейнера знаходяться три незалежні мікрофронтенди:

- MFE A – автономний модуль, який взаємодіє з контейнером через API або Event Bus, забезпечуючи обмін подіями та даними;
- MFE B – модуль, що працює із загальним станом (Shared State) для синхронізації даних між частинами інтерфейсу;
- MFE C – компонент, який реалізує повторне використання інтерфейсних елементів через Web Components або інші стандартизовані механізми інтеграції.

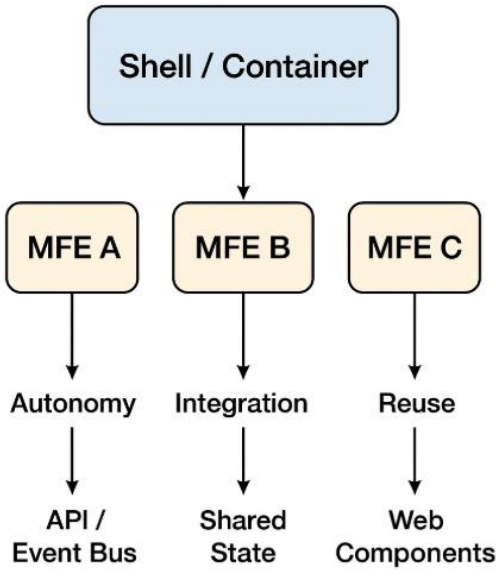


Рис.1.2. Схема взаємодії мікрофронтендів у бізнес-додатку

На схемі відображені ключові властивості цієї архітектури. Автономність — кожен мікрофронтенд може розроблятися, тестуватися і деплоїтися незалежно. Інтеграція — забезпечується через API, Event Bus або спільний стан. Повторне використання — можливість створення універсальних компонентів, які можна застосовувати у різних частинах бізнес-додатку. Схема демонструє, як мікрофронтендна архітектура дозволяє розділити велику систему на модулі, що полегшує масштабування, підвищує гнучкість розробки та зменшує ризики при оновленні або розширенні функціоналу.

Масштабовані бізнес-аплікації характеризуються високою складністю функціоналу, великою кількістю користувачів, потребою інтеграції з різними системами та необхідністю швидкого оновлення і розвитку. Традиційні фронтенд-архітектури – моноліти та SPA-додатки – не завжди можуть ефективно впоратися з

цими вимогами. Саме тому сучасні проєкти дедалі частіше обирають архітектуру мікрофронтендів.

Мікрофронтендна архітектура має низку переваг, які роблять її ефективним підходом для створення масштабованих бізнес-додатків. Однією з головних переваг є модульність і автономність. Великі інтерфейси розділяються на незалежні модулі, кожен з яких виконує свою конкретну бізнес-функцію, наприклад, управління користувачами, аналітику чи обробку замовлень. Це дає змогу командам розробляти, тестувати та впроваджувати зміни автономно, без ризику вплинути на роботу інших частин системи, що особливо важливо в умовах багатокомандної розробки.

Ще однією сильною стороною є масштабованість командної роботи. Коли над одним бізнес-додатком працюють десятки чи сотні розробників, мікрофронтенди дозволяють розділити зони відповідальності між командами. Кожна команда може працювати над власним модулем, не створюючи конфліктів у коді, що значно прискорює процес розробки і спрощує координацію.

Гнучкість технологічного стеку також є вагомим аргументом. Мікрофронтенди дають можливість використовувати різні фреймворки чи бібліотеки в межах одного проєкту — React, Angular, Vue чи Svelte — залежно від функціональних потреб модуля або компетенцій команди. Це знімає обмеження, властиві монолітним підходам, і дозволяє впроваджувати сучасні технології без ризику переробки всього додатку.

Не менш важливою перевагою є незалежність деплою та оновлень. Кожен мікрофронтенд може оновлюватися окремо, без впливу на інші модулі системи. Це значно знижує ризик виникнення збоїв і простоїв під час оновлення, що критично для корпоративних систем, які повинні працювати безперервно.

Важливим аспектом є покращена інтеграція та узгодженість даних. Завдяки централізованим API, подієвій шині (Event Bus) або спільному стану (Shared State) досягається синхронізація між модулями та забезпечується єдиний підхід до обміну даними. Це дозволяє отримувати точні аналітичні дані та підтримувати узгодженість у роботі всіх компонентів.

Крім того, мікрофронтенди забезпечують високу надійність та стійкість системи. Якщо один модуль виходить із ладу або містить помилку, інші частини додатку продовжують працювати стабільно. Такий підхід мінімізує ризик глобальних відмов і дозволяє швидше знаходити та виправляти проблеми. Автоматизоване тестування окремих модулів підвищує якість коду, а розподілена архітектура сприяє гнучкому розвитку системи у довгостроковій перспективі. Таким чином, мікрофронтенди поєднують у собі модульність, масштабованість, технологічну гнучкість і високу надійність, що робить їх оптимальним вибором для побудови сучасних масштабованих бізнес-аплікацій.

Таблиця 1.4

Порівняння мікрофронендy з традиційними архітектурами

Параметр	Монолітний фронтенд	SPA	Мікрофронтенди
Модульність	Низька	Середня	Висока, автономні модулі
Масштабованість	Обмежена	Середня	Висока, легке додавання нових модулів
Командна робота	Важка, конфлікти	Можлива, але складна	Паралельна, незалежна
Оновлення та деплой	Центральний деплой	Частковий деплой	Незалежний деплой модулів
Використання технологій	Один стек	Один/обмежений стек	Можливість різних стеків для кожного модуля
Надійність	Помилки впливають на весь додаток	Помилки можуть впливати на SPA	Помилки одного модуля ізольовані

Враховуючи високі вимоги сучасних масштабованих бізнес-аплікацій до автономності модулів, масштабованості командної роботи, гнучкості технологічного стеку, незалежного деплою та надійності системи, архітектура мікрофронтендів є оптимальним вибором для побудови інтерфейсів користувача. Вона дозволяє ефективно вирішувати проблеми, пов’язані з масштабуванням, інтеграцією та управлінням складними бізнес-процесами, забезпечуючи швидке оновлення і високу стабільність системи.

Таким чином, застосування мікрофронтенд-архітектури у великих бізнес-додатках обґрунтоване як з точки зору технічної ефективності, так і з позицій підвищення продуктивності команд розробників та стабільності інтерфейсу користувача.

Тому слід вирішити у роботі такі завдання:

1. Проаналізувати існуючі підходи до розробки фронтенд-архітектур та виділити їх сильні і слабкі сторони.
2. Вивчити принципи архітектури мікрофронтендів та визначити ключові технології, які застосовуються для її реалізації.
3. Дослідити вимоги масштабованих бізнес-аплікацій до інтерфейсу користувача та проблеми традиційних монолітних рішень.
4. Розробити модульну структуру інтерфейсу користувача на основі мікрофронтендів.
5. Реалізувати прототип бізнес-аплікації з використанням обраних технологій і оцінити її продуктивність, масштабованість та зручність підтримки.
6. Порівняти результати реалізації з традиційними фронтенд-архітектурами і сформулювати рекомендації щодо застосування мікрофронтендів у бізнес-розробці.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ МОДУЛЬНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА З ВИКОРИСТАННЯМ МІКРОФРОНТЕНДІВ

2.1. Постановка задачі та загальна архітектура рішення

Метою даного розділу є розробка концептуальної та практичної моделі модульного інтерфейсу користувача для масштабованої бізнес-аплікації з використанням архітектури мікрофронтендів. У сучасних бізнес-системах, де одночасно працюють сотні користувачів і інтегруються численні бізнес-процеси, традиційні монолітні фронтенд-архітектури часто не забезпечують необхідної гнучкості, автономності модулів та стабільності.

Для прикладу розглянемо бізнес-аплікацію, яка включає такі функціональні модулі:

1. Управління замовленнями – створення, редагування, відстеження стану замовлень;
2. Аналітика продажів – формування дашбордів, звітів і графіків на основі даних продажів;
3. Облік товарів на складі – ведення бази товарів, контроль залишків, повідомлення про дефіцит;
4. Клієнтський портал – особистий кабінет користувача, історія замовлень, підтримка;
5. Адміністративна панель – управління ролями користувачів, налаштування системи, моніторинг активності.

Задача полягає у створенні інтегрованого інтерфейсу користувача, де кожен із цих модулів буде автономним мікрофронтендом, здатним незалежно завантажуватися, оновлюватися та взаємодіяти з іншими частинами системи. Основні вимоги до рішення:

- модульність і автономність - кожен бізнес-модуль реалізується окремим мікрофронтендом;

- масштабованість - можливість додавання нових модулів без впливу на існуючі;
- інтеграція - єдина точка доступу для користувача через контейнер (Shell Application);
- надійність - помилки одного модуля не повинні впливати на роботу інших;
- гнучкість технологій - кожен модуль може використовувати власний технологічний стек (React, Angular, Vue, Svelte тощо);
- автоматизація деплою - можливість незалежного CI/CD для кожного модуля;
- узгодженість даних - централізоване управління станом через API Gateway або Event Bus.

Для реалізації описаних вимог та забезпечення високої масштабованості, надійності й гнучкості у розробці була обрана архітектура мікрофронтендів, побудована на основі контейнера (Shell / Container), який виконує роль центрального елемента керування всією системою. Такий підхід дозволяє об’єднати окремі незалежні модулі в єдиний узгоджений інтерфейс користувача, забезпечити стабільну маршрутизацію між ними, а також підтримувати єдину логіку автентифікації, авторизації та управління станом. Архітектура включає кілька взаємопов’язаних рівнів, які забезпечують модульність і гнучкість розгортання додатку.

1. Shell / Container (контейнер додатку).

На верхньому рівні розташований Shell / Container, що виступає контейнером усього додатку. Це центральна точка входу для користувачів, яка відповідає за маршрутизацію між модулями, керування навігацією, а також відображення загальних елементів інтерфейсу — таких як шапка сайту (header), панель навігації (menu), підвал (footer) та панель авторизації. Контейнер також відповідає за динамічне підключення мікрофронтендів під час роботи користувача, використовуючи технології Module Federation або Single SPA, що забезпечує безперервну інтеграцію нових модулів без перезапуску всього додатку. Крім того,

Shell виконує роль координаційного центру для комунікацій між модулями, завдяки чому всі частини системи залишаються узгодженими навіть при незалежному оновленні.

2. Мікрофронтенди (MFE – Micro Frontends)

Другий рівень складається з окремих мікрофронтендів (Micro Frontends, MFE), кожен з яких є самостійним веб-додатком із власною логікою, стилями, маршрутами, компонентами та іноді навіть технологічним стеком. У розробленій архітектурі передбачено декілька ключових модулів:

- MFE Orders — управління замовленнями, формування, редагування та відстеження статусу;
- MFE Analytics — модуль аналітики продажів і побудови інтерактивних звітів;
- MFE Inventory — управління складськими залишками, оновлення товарів і синхронізація з ERP-системами;
- MFE Client Portal — персональний кабінет користувача, що забезпечує доступ до історії замовлень, профілю та налаштувань;
- MFE Admin — адміністративна панель для управління правами доступу, контентом і загальними налаштуваннями системи.

Кожен із цих мікрофронтендів розробляється незалежно різними командами, може мати власний життєвий цикл та деплоїтися окремо. Це дозволяє уникати конфліктів у коді, пришвидшувати розробку і спрощувати оновлення окремих частин без зупинки всієї системи.

3. Сервісний рівень (API Gateway / Backend).

Сервісний рівень (API Gateway / Backend) забезпечує централізовану взаємодію з базами даних, зовнішніми сервісами та внутрішніми API. Через API Gateway усі мікрофронтенди отримують узгоджені дані, що запобігає дублюванню запитів і підтримує єдину бізнес-логіку. Цей рівень також виконує функції автентифікації, авторизації, кешування, обробки запитів і контролю доступу, що гарантує безпеку та стабільність роботи системи.

4. Комунікаційний рівень.

Комунікаційний рівень відповідає за обмін даними між модулями та синхронізацію їх стану. Для цього використовуються Event Bus**,** Redux**,** Vuex або RxJS, які забезпечують передачу подій і зміни глобального стану у реальному часі. Наприклад, якщо користувач змінює налаштування теми або фільтри аналітики в одному модулі, ці зміни автоматично синхронізуються з іншими мікрофронтендами, що забезпечує узгодженість усього інтерфейсу.

Таким чином, обрана архітектура мікрофронтендів із Shell-контейнером і чітко визначеними рівнями — це гнучке, масштабоване і технологічно стійке рішення, яке поєднує переваги модульності, незалежності розробки, централізованої інтеграції та надійної комунікації. Вона дозволяє легко додавати нові функціональні блоки, впроваджувати сучасні технології і забезпечувати безперебійну роботу бізнес-додатку навіть при активному розвитку системи.

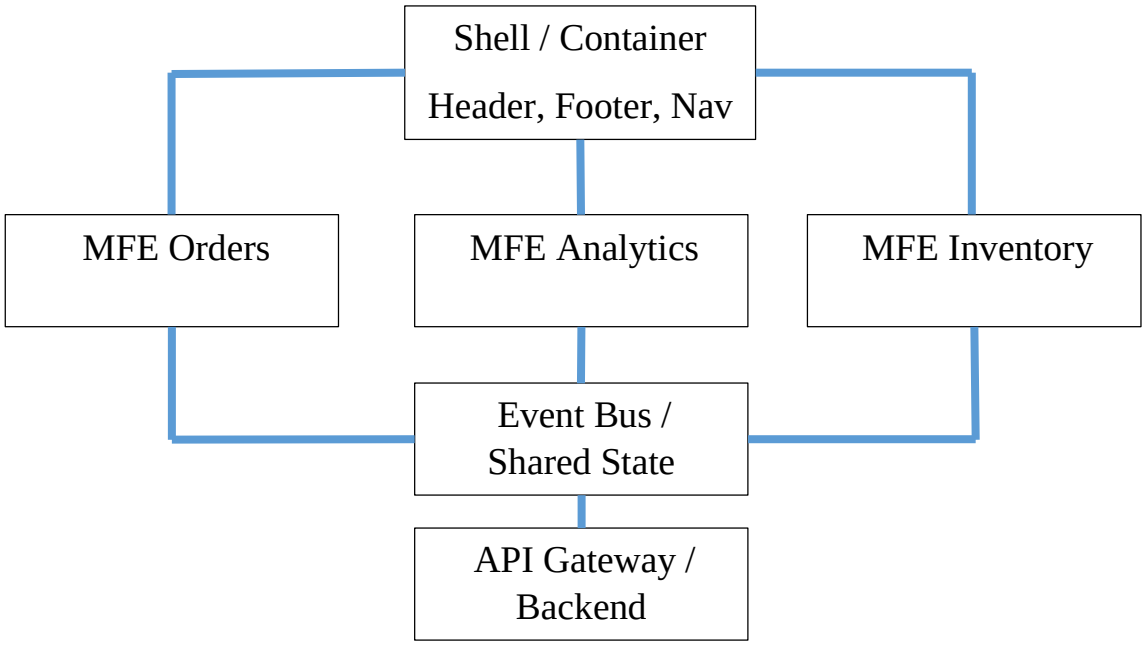


Рис.2.1. Схематичне відображення архітектури рішення

Архітектура мікрофронтендів має низку переваг, що робить її привабливою для розробки сучасних вебзастосунків. Основна перевага полягає в автономності модулів — кожен мікрофронтенд функціонує як окрема самостійна частина системи, яка може оновлюватися, тестуватися чи розгортатися незалежно від інших компонентів. Це забезпечує можливість паралельної розробки, коли різні команди працюють над власними модулями, не створюючи конфліктів і не залежачи одна

від одної. Це, у свою чергу, значно прискорює цикл розробки та спрощує координацію в межах великого проєкту.

Архітектура мікрофронтендів сприяє високій масштабованості системи — додавання нових функціональних модулів або перехід на інші технології здійснюється без серйозного втручання в існуючу інфраструктуру. Це також підвищує надійність і стабільність роботи вебзастосунку, оскільки збої чи помилки в одному модулі не впливають на роботу решти компонентів. Крім того, централізоване управління станом і подіями через такі механізми, як Redux чи Event Bus, забезпечує узгодженість даних між модулями та спрощує інтеграцію різних частин системи.

Ще однією важливою перевагою є простота тестування та розгортання — кожен модуль можна перевіряти окремо, не зачіпаючи інші частини застосунку. Це зменшує ризик виникнення помилок під час оновлення та дозволяє швидше впроваджувати зміни, що особливо важливо в умовах динамічного розвитку продукту. Таким чином, мікрофронтендна архітектура забезпечує гнучкість, стабільність і ефективність у побудові масштабованих вебсистем.

Постановка задачі та розроблена загальна архітектура демонструють, що використання мікрофронтендів є оптимальним рішенням для побудови модульного інтерфейсу користувача у масштабованих бізнес-аплікаціях. Архітектура забезпечує автономність модулів, паралельну розробку, масштабованість, стабільність та інтеграцію з різними системами, що відповідає сучасним вимогам бізнесу та користувачів.

2.2. Вибір технологій і інструментів для реалізації мікрофронтендів

Реалізація архітектури мікрофронтенду у сучасних бізнес-аплікаціях потребує ґрунтовного підходу до вибору технологій, які забезпечують модульність, масштабованість, ефективну комунікацію між компонентами та простоту розгортання. Розглянемо ключові інструменти, технологічні стекі та практики, які

дозволяють побудувати надійну систему на основі мікрофронтендів, орієнтовану на сучасні вимоги бізнесу.

Вибір технологій для мікрофронтенду визначається низкою критеріїв:

1. Незалежність модулів — кожен модуль має мати власний життєвий цикл розробки, тестування та деплою.

2. Сумісність з різними фреймворками — архітектура повинна підтримувати можливість використання різних технологій у різних частинах інтерфейсу (наприклад, React у одному модулі, Vue або Angular — в іншому).

3. Мінімальна взаємозалежність — модулі не повинні мати щільних зв’язків, що ускладнюють оновлення або масштабування.

4. Підтримка динамічного завантаження — можливість асинхронного підвантаження компонентів без перезавантаження сторінки.

5. Простота інтеграції з бекендом — сумісність з REST API, GraphQL або gRPC.

6. Масштабованість і DevOps-сумісність — підтримка CI/CD, контейнеризації (Docker, Kubernetes), розподіленого моніторингу та логування.

Фреймворки для реалізації мікрофронтендів

Single-SPA — один із найпопулярніших фреймворків для побудови мікрофронтендів. Він дозволяє поєднувати кілька незалежних SPA (Single Page Application) в одному середовищі. Основні переваги: підтримка різних технологій одночасно (React, Vue, Angular, Svelte); динамічне завантаження модулів під час навігації; централізоване керування маршрутизацією; спрощена інтеграція зі сторонніми бібліотеками; активна підтримка спільноти.

Типова архітектура проєкту на Single-SPA складається з root-config — головного контейнера, який відповідає за маршрутизацію та ініціалізацію піддодатків; applications — незалежних мікрофронтендів (наприклад, “Orders”, “Users”, “Reports”); shared libraries — спільних модулів (напр., автентифікація, стилі, компоненти UI).

Module Federation (Webpack 5) — нова концепція, представлена в Webpack 5, яка дозволяє динамічно підвантажувати код з інших додатків під час виконання. Це

сучасний і гнучкий механізм обміну компонентами між різними проєктами. Переваги: мінімальні залежності між модулями; зменшення дублювання коду (shared dependencies); динамічна інтеграція без повторної збірки всього застосунку; сумісність з React, Vue, Angular, Svelte тощо. Module Federation часто використовується в поєднанні з React і Next.js, де кожен мікрофронтенд — це окремий застосунок, який може бути розгорнутий незалежно.

Qiankun — китайський фреймворк, створений на базі Single-SPA з покращенням у галузі безпеки, кешування та управління життєвим циклом піддодатків. Його переваги: вбудована підтримка sandbox ізоляції DOM; краща оптимізація при роботі з різними фреймворками; кешування піддодатків для швидкого перемикання між ними. Qiankun часто використовується у великих корпоративних платформах, де важлива стабільність та узгодженість інтерфейсу.

Web Components — це стандартна технологія браузера, що дозволяє створювати повторно використовувані, ізольовані компоненти. До її складу входять: Custom Elements — користувацькі HTML-теги; Shadow DOM — ізольований DOM для інкапсуляції стилів; HTML Templates — шаблони для створення структури компонентів. Web Components чудово підходять для мікрофронтендів, оскільки не прив’язують розробника до конкретного фреймворку й забезпечують справжню технологічну незалежність. Їх можна інтегрувати у будь-яке середовище: React, Angular або навіть звичайний HTML.

Технології для маршрутизації та комунікації між модулями

Мікрофронтенди повинні ефективно взаємодіяти між собою. Для цього використовують такі механізми: Custom Event API — обмін повідомленнями між модулями через події браузера, RxJS / EventBus — реалізація реактивного потоку даних, Redux Toolkit або Zustand (для React) — централізоване управління станом, якщо декілька модулів працюють з одними даними, Shared Context API — поширений підхід у React для розподілу глобального стану.

Інструменти збірки, деплою та інтеграції

Для розгортання і підтримки архітектури мікрофронтенду важливо забезпечити автоматизацію процесів CI/CD. Найпоширеніші рішення:

- Webpack + Babel — оптимізація, компіляція та мінімізація коду.
- Vite — швидка збірка для модульних систем із підтримкою HMR (Hot Module Replacement).
- Docker — контейнеризація кожного мікрофронтенду для незалежного розгортання.
- Kubernetes — оркестрація контейнерів, масштабування, моніторинг.
- Nginx або Traefik — реверс-проксі сервери для маршрутизації запитів між модулями.
- GitLab CI/CD або GitHub Actions — автоматизація тестування, збірки й деплою.

Бібліотеки для уніфікації інтерфейсу

Щоб забезпечити єдину візуальну мову всіх мікрофронтендів, використовуються UI-бібліотеки:

- Material UI / Ant Design / Chakra UI — стандартизований набір компонентів;
- Storybook — ізольоване середовище для розробки та тестування UI-компонентів;
- Tailwind CSS — утилітарний CSS-фреймворк для швидкої побудови інтерфейсів;
- CSS Modules / Emotion / Styled Components — ізоляція стилів для уникнення конфліктів між модулями.

Вибір стеку для реалізації. У контексті даного дослідження, оптимальним вибором для побудови мікрофронтенд-системи є комбінація таких технологій:

- React + Module Federation (Webpack 5) — основа для реалізації незалежних модулів;
- TypeScript — типізація для підвищення стабільності коду;
- Tailwind CSS — модульний підхід до стилізації;
- Redux Toolkit / Zustand — централізоване управління станом;
- Docker + Kubernetes — масштабоване середовище розгортання;
- GitHub Actions — CI/CD пайплайн.

Вибір технологій для побудови архітектури мікрофронтендів є ключовим етапом, який визначає гнучкість, продуктивність та довгострокову життєздатність системи. Використання таких підходів, як Module Federation, Single-SPA і Web Components, забезпечує технологічну незалежність, швидке оновлення інтерфейсу, спрощену інтеграцію нових функцій та легку масштабованість системи. Таким чином, правильно сформований технологічний стек дозволяє створити сучасну, модульну, надійну й ефективну бізнес-аплікацію, готову до розширення й адаптації під потреби користувачів.

Структурована таблиця 2.1 вибору технологій і інструментів для реалізації архітектури мікрофронтендів у масштабованих бізнес-аплікаціях:

Таблиця 2.1

Вибір технологій і інструментів для реалізації мікрофронтендів

№	Категорія	Технологія / Інструмент	Призначення	Основні переваги
1	Фреймворки для мікрофронтендів	Single-SPA	Інтеграція кількох SPA в одну систему	Підтримка React, Vue, Angular; централізована маршрутизація; динамічне завантаження
		Module Federation (Webpack 5)	Динамічний обмін кодом між незалежними застосунками	Незалежне розгортання модулів, зменшення дублювання бібліотек
		Qiankun	Розширена платформа на базі Single-SPA	Ізоляція DOM, кешування піддодатків, підвищена безпека
		Web Components	Створення універсальних UI-компонентів незалежно від фреймворку	Технологічна незалежність, інкапсуляція стилів, сумісність з будь-яким середовищем
2	Мови програмування та типізація	JavaScript / TypeScript	Основна мова фронтенду	TypeScript забезпечує статичну типізацію та кращу підтримку великих проєктів
3	Фреймворки для UI	React	Основна бібліотека для побудови компонентів інтерфейсу	Висока продуктивність, компонентна структура, велика екосистема
		Vue.js / Angular	Альтернативні фрейм ворки для інших модулів	Можливість гібридної систе ми з різними технологіями
4	Управління станом	Redux Toolkit / Zustand / RxJS	Централізоване зберігання стану додатку	Прозора синхронізація між модулями, спрощене тестування, реактивність
5	Маршрутизація та комунікація	Custom Event API / EventBus /Shared Context	Взаємодія між незалежними модулями	Мінімальна залежність, спрощений обмін повідомленнями

№	Категорія	Технологія / Інструмент	Призначення	Основні переваги
6	Збірка і оптимізація	Webpack / Vite / Babel	Збірка, мінімізація, транспіляція та оптимізація коду	Швидка компіляція, підтримка HMR, сумісність з Module Federation
7	Стилізація та UI-уніфікація	Tailwind CSS / Material UI / Ant Design / Chakra UI	Уніфікований вигляд інтерфейсу користувача	Адаптивний дизайн, багато готових компонентів
		Styled Components / CSS Modules / Emotion	Ізоляція стилів між модулями	Уникає конфліктів CSS, забезпечує повторне використання стилів
8	Тестування	Jest / Cypress / Playwright	Юніт- і енд-ту-енд тестування	Автоматизація тестів, висока надійність модулів
9	Контейнеризація і розгортання	Docker / Kubernetes	Ізоляція та масштабо ване розгортання мікрофронтендів	Незалежні середовища, гнучке масштабування, оркестрація контейнерів
10	Сервери маршрутизації	Nginx / Traefik	Реверс-проксі та маршрутизація запитів між модулями	Балансування навантаження, безпечна комунікація
11	CI/CD інструменти	GitHub Actions / GitLab CI/CD / Jenkins	Автоматизація процесів збірки, тестування та деплою	Безперервна інтеграція, швидке оновлення системи
12	Управління бібліотеками компонентів	Storybook	Ізольована розробка і тестування UI-компонентів	Документування, перевико ристання компонентів, візуальна перевірка
13	Комунікація з бекендом	REST API / GraphQL/gRPC	Передача даних між фронтендом і сервером	Гнучкий обмін даними, оптимізація запитів

На основі порівняння представлених технологій для реалізації архітектури мікрофронтендів рекомендовано використовувати технологічний стек представлений у таблиці 2.2.

Таблиця 2.2

Технологічний стек для реалізації архітектури мікрофронтендів

Компонент	Вибрана технологія
Основний фреймворк	React
Механізм мікрофронтендів	Module Federation (Webpack 5)
Типізація	TypeScript
Стилізація	Tailwind CSS
Управління станом	Redux Toolkit
Контейнеризація	Docker
Оркестрація	Kubernetes
CI/CD	GitHub Actions
UI-компоненти	Material UI + Storybook

2.3. Розробка структури модулів: принципи декомпозиції

Побудова масштабованої бізнес-аплікації на основі архітектури мікрофронтендів вимагає чіткого визначення структури модулів, їхніх функцій, взаємозв'язків і рівнів відповідальності. Декомпозиція — це процес поділу системи на незалежні логічні частини (модулі), які можна розробляти, тестувати та розгортати окремо. Саме грамотна декомпозиція є основою для забезпечення гнучкості, повторного використання коду, масштабованості та простоти супроводу. Модульність передбачає створення системи з незалежних компонентів, де кожен модуль реалізує певну бізнес-функцію або частину інтерфейсу; має власні залежності, ресурси та життєвий цикл; може розгортатися окремо без впливу на інші модулі.

У класичних монолітних SPA (Single Page Applications) всі компоненти з'єднані в єдину структуру. У мікрофронтенд-архітектурі, навпаки, кожен модуль — це окремий застосунок, який може мати власну технологію, систему збирання та навіть окремий репозиторій коду. Така структура відповідає принципу незалежного розгортання (Independent Deployability), що дозволяє командам працювати автономно над різними частинами проєкту.

Основна мета декомпозиції — поділ великої системи на логічно завершені модулі. При цьому важливо зберегти єдність користувацького досвіду (UX) та узгодженість дизайну.

Функціональна декомпозиція – це поділ системи на модулі за бізнес-логікою. Кожен модуль відповідає певному бізнес-процесу або сутності:

- модуль "Користувачі" — управління обліковими записами;
- модуль "Замовлення" — оформлення та обробка замовлень;
- модуль "Аналітика" — візуалізація статистики;
- модуль "Налаштування" — персоналізація профілю.

Функціональна декомпозиція забезпечує зрозумілу логіку розподілу відповідальності, що полегшує подальше масштабування.

Технічна декомпозиція – це розподіл за технічними аспектами роботи інтерфейсу:

- Core (ядро) — маршрутизація, навігація, загальні стилі, автентифікація;
- Shared (спільні модулі) — бібліотеки, UI-компоненти, фільтри, константи;
- Feature Modules (функціональні модулі) — незалежні частини, які реалізують бізнес-логіку;
- Micro Apps (мікрозастосунки) — самостійні SPA, інтегровані через Single-SPA або Module Federation.

Така структура дозволяє уникати дублювання коду, зберігаючи при цьому незалежність модулів.

Декомпозиція за рівнями взаємодії. Для великих систем доцільно виділяти модулі за рівнями:

- Presentation Layer (UI) — відображення інтерфейсу, компоненти, стилі;
- Application Layer — керування станом, маршрутизація, інтеграція модулів;
- Data Layer — робота з API, кешування, обробка запитів;
- Infrastructure Layer — спільні сервіси (логування, безпека, аналітика).

Це відповідає принципам чистої архітектури (Clean Architecture) та SOLID, зокрема принципу єдиної відповідальності (Single Responsibility Principle).

Таблиця 2.3

Критерії ефективної декомпозиції

№	Критерій	Опис
1	Ізольованість	Модуль не залежить від внутрішньої реалізації інших модулів.
2	Повторне використання	Компоненти або сервіси можуть бути застосовані в інших модулях.
3	Простота інтеграції	Модуль легко підключається до центрального контейнера (root application).
4	Незалежне розгортання	Можливість оновлення або заміни модуля без впливу на інші частини системи.
5	Мінімальні залежності	Використання лише публічних інтерфейсів (API), мінімум глобальних змінних.
6	Відповідність бізнес-логіці	Модулі відображають реальні бізнес-процеси компанії.
7	Легкість тестування	Модуль можна ізольовано тестувати з допомогою Jest, Cypress тощо.

Під час розробки структури мікрофронтенд-модулів важливо враховувати критерії ефективної декомпозиції, які представлені у таблиці 2.3.

Для кращого розуміння розглянемо приклад структури бізнес-аплікації, побудованої за мікрофронтенд-підходом:

```
/root-config/
│    ├── index.html
│    ├── main.js
│    └── router-config.js
│
├── /auth-app/
│    ├── src/
│    │    ├── components/
│    │    ├── pages/
│    │    ├── services/
│    │    └── store/
│    └── webpack.config.js
│
├── /orders-app/
│    ├── src/
│    │    ├── components/
│    │    ├── pages/
│    │    └── api/
│    └── webpack.config.js
│
├── /analytics-app/
│    ├── src/
│    │    ├── charts/
│    │    ├── dashboards/
│    │    └── utils/
│    └── webpack.config.js
│
└── /shared/
     ├── components/
     ├── utils/
     └── styles/
```

Рис.2.2. Структура бізнес-аплікації, побудованої за мікрофронтенд-підходом.

Наведена структура бізнес-додатку демонструє типову організацію проєкту, побудованого за мікрофронтенд-архітектурою, де кожен піддодаток (модуль) функціонує автономно, але інтегрується у спільне середовище через головний контейнер (root-config).

У верхньому рівні розміщено каталог /root-config/ — це центральний контейнер, який відповідає за загальну маршрутизацію, початкове завантаження піддодатків та інтеграцію між ними. Файл index.html є точкою входу до всієї системи, main.js ініціалізує мікрофронтенди, а router-config.js визначає правила маршрутизації між окремими модулями (наприклад, /auth, /orders, /analytics). Саме

контейнер виступає «оболонкою» для підключення та взаємодії незалежних SPA-застосунків.

Далі розміщуються каталоги /auth-app/, /orders-app/, /analytics-app/ — це окремі мікрофронтенди, кожен з яких має власну внутрішню структуру.

- /auth-app/ реалізує функціональність автентифікації та авторизації користувачів. Усередині директорії src/ містяться підпапки components/ (UI-компоненти, наприклад форми входу), pages/ (окремі сторінки, такі як Login чи Register), services/ (логіка запитів до API) та store/ (механізми зберігання стану, наприклад Redux або Zustand). Файл webpack.config.js визначає параметри збірки та експорту компонентів.
- /orders-app/ відповідає за управління замовленнями — відображення списків, деталей замовлень, фільтрацію, CRUD-операції. У каталозі src/ зберігаються components/ (таблиці, картки, форми), pages/ (сторінки списку замовлень та деталей), а також api/ — інтерфейси для обміну даними з бекендом.
- /analytics-app/ містить модуль аналітики. Підпапки charts/ та dashboards/ використовуються для побудови візуальних звітів, графіків і KPI-панелей, а utils/ містить допоміжні функції для форматування даних, обчислень та роботи з API.

Окремо виділений спільний каталог /shared/, який містить повторно використовувані ресурси — components/ (наприклад, кнопки, модальні вікна, панелі навігації), utils/ (загальні утиліти, функції валідації, форматування дат тощо) та styles/ (глобальні стилі або змінні CSS/SCSS). Це дозволяє дотримуватись єдиного стилю в усіх піддодатках і мінімізує дублювання коду.

Завдяки такій структурі команди можуть незалежно розробляти, тестувати й розгортати свої мікрофронтенди, не порушуючи стабільність усієї системи. Кожен модуль має власну збірку, залежності, CI/CD-процеси та може оновлюватися без потреби повного релізу всієї аплікації. Це забезпечує масштабованість, гнучкість інтеграції та простоту підтримки бізнес-додатку.

Оскільки кожен мікрофронтенд є автономним, між ними необхідно забезпечити ефективну комунікацію. Основні способи взаємодії - через спільні сервіси (shared state, наприклад Redux store); через події браузера (CustomEvent,

EventBus); через API-виклики до спільного бекенду; через контракти (інтерфейси), описані у shared-бібліотеках.

Важливо, щоб модулі не мали прямих імпортів один одного, а взаємодіяли через чітко визначені канали. Це зменшує ризик залежностей і забезпечує гнучкість системи.

Узгодженість дизайну та UX. Окрема проблема мікрофронтендів — різноманіття фреймворків і бібліотек. Щоб уникнути візуальної фрагментації інтерфейсу, використовують єдину дизайн-систему (наприклад, Material Design, Ant Design або власну); спільну бібліотеку UI-компонентів (shared UI library); уніфіковані правила стилів (Tailwind CSS, CSS Variables); інтеграційне тестування для перевірки сумісності компонентів. Це дозволяє користувачу сприймати систему як єдине ціле, навіть якщо окремі модулі створені різними командами.

Етапи розробки структури модулів

1. Аналіз бізнес-процесів – визначення ключових функцій системи.
2. Формування доменних областей – групування функцій за бізнес-контекстом.
3. Виділення модулів – створення автономних SPA або бібліотек.
4. Визначення контрактів взаємодії – встановлення інтерфейсів для обміну даними.
5. Розроблення shared-бібліотек – створення спільних компонентів, стилів і утиліт.
6. Інтеграція модулів у root-додаток – через Single-SPA, Module Federation або Qiankun.
7. Налаштування CI/CD – забезпечення автономного деплою кожного модуля.

Розробка структури модулів у мікрофронтенд-архітектурі — це стратегічний етап проєктування, що визначає життєздатність усієї системи. Грамотна декомпозиція дозволяє: зменшити складність проєкту; підвищити автономність команд; спростити масштабування та оновлення; забезпечити стабільність і повторне використання коду.

Таким чином, принципи декомпозиції, засновані на функціональній, технічній та рівневій структуризації, створюють фундамент для побудови модульних, масштабованих і гнучких бізнес-аплікацій, готових до довгострокового розвитку.

2.4. Комунікація між мікрофронтенд-модулями: підходи та реалізація

Ефективна взаємодія між мікрофронтенд-модулями є критичною умовою для побудови масштабованої та стабільної бізнес-аплікації. Кожен мікрофронтенд є автономним SPA (Single Page Application), що реалізує певний набір функцій. Однак у реальних умовах модулі потребують обміну даними, синхронізації станів та узгодження дій користувача.

Під час розробки мікрофронтенд-архітектури особливу увагу приділяють правильній організації комунікації між модулями, адже від цього залежить стабільність і масштабованість усієї системи. Основним принципом є автономність, коли кожен модуль працює незалежно й не покладається на внутрішню логіку інших. Взаємодія між ними відбувається виключно через визначені інтерфейси або подієві механізми, що гарантує слабку зв’язність і спрощує оновлення окремих частин системи.

Не менш важливим є принцип ізоляції стану, за якого кожен мікрофронтенд самостійно керує власними даними, а узгодження глобального стану реалізується через спільне сховище (shared state) або API-запити до бекенду. Це дає змогу уникати конфліктів при зміні станів у різних модулях.

Комунікація між компонентами базується на подієво-орієнтованій моделі, де інформація передається не шляхом прямого виклику методів, а через події. Це дозволяє системі бути гнучкою, легко масштабуватися й підтримувати незалежну розробку кожного піддодатку.

Ще одним важливим елементом є чітко визначені контракти взаємодії — стандартизовані API або подієві канали, які гарантують сумісність і

передбачуваність поведінки при оновленні або додаванні нових модулів. Завдяки цьому система зберігає стабільність навіть під час активного розширення.

Принцип масштабованості забезпечує можливість підключення нових мікрофронтендів без потреби змінювати існуючу структуру обміну даними. Це створює основу для гнучкого зростання бізнес-додатку, полегшує інтеграцію інноваційних рішень і гарантує безперервну еволюцію системи без ризику порушення її цілісності.

Схема побудована навколо центрального Shell / Container (контейнер, root-аплікація), що розміщено в центрі візуалізації. В контейнері — кілька кольорових блоків, які позначають окремі MFE (micro frontends): Orders, Analytics, Inventory (на схемі також показані Client Portal та Auth як зовнішні або додаткові модулі). Під контейнером розташований блок API / Event Bus — спільний комунікаційний шар. З боків позначені зовнішні Backend сервіси. Стрілки та лінії різного типу показують шляхи взаємодії: події, API-запити, спільний стан.

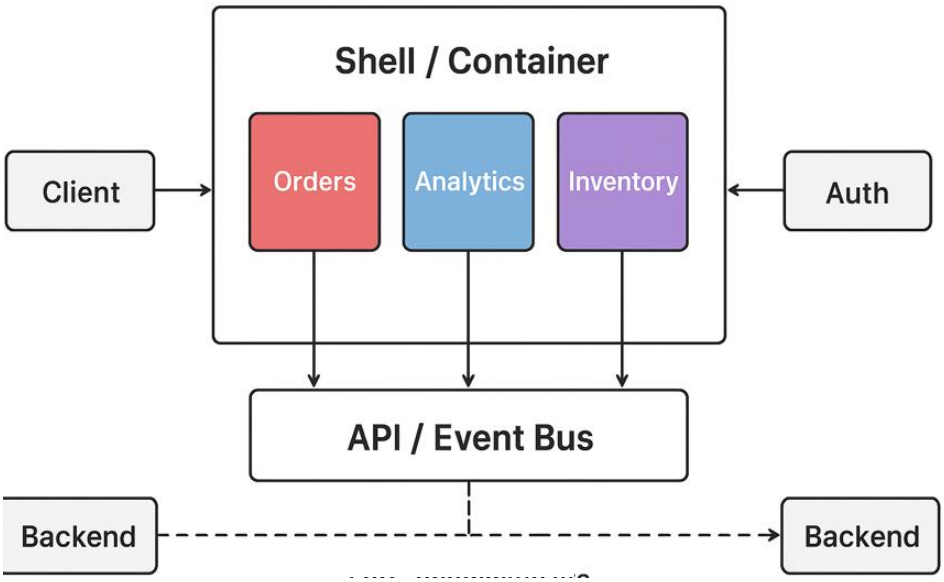


Рис.2.3. Схема архітектури мікрофронтендів

Shell / Container - головна точка входу користувача; забезпечує глобальні елементи UI (header, навігацію, footer), авторизацію/сесію, центральну маршрутизацію, завантаження і вмонтування мікрофронтендів. Функції: **і**ніціалізація процесу авторизації / перевірки прав доступу; динамічне завантаження remote-модулів (під час навігації) — через Module Federation чи Single-SPA root-config; управління глобальними ресурсами: теми, локалізація, загальні стилі;

моніторинг і fallback: якщо один MFE не доступний, shell показує запасний UI або повідомлення. Shell не повинен містити бізнес-логіки модулів — лише координує інтеграцію.

MFE Orders, MFE Analytics, MFE Inventory (і інші) - автономні піддодатки, кожен реалізує окрему бізнес-функцію. Характеристики - власна маршрутизація, збірка та життєвий цикл; може використовувати свій стек (React/Angular/Vue/Svelte) та власну збірку; має власні UI-компоненти і стилі; стилі можна ізолювати через Shadow DOM або CSS Modules; кожен MFE може деплоїтися окремо; при збої Orders інші модулі продовжують роботу.

API / Event Bus (комунікаційний шар) - основний канал для синхронізації й передачі даних між модулями і бекендом. Складові: API Gateway / Backend (REST/GraphQL) — для персистентності, логіки з боку сервера, автентифікації; Event Bus / Message Bus (локальний або сервісний, наприклад, через in-memory EventBus, WebSocket, або Pub/Sub) — для передачі подій у реальному часі; Shared State (опціонально) — централізований store (Redux, Zustand), якщо потрібно швидко синхронізувати стан між модулями. Принцип роботи - модулі можуть надсилати подію (orderCreated) у Event Bus; підписані модулі (Analytics) отримують її і оновлюють UI.

Backend (зовнішні сервіси) зберігання даних, бізнес-логіка, автентифікація, інтеграції (ERP, платіжні шлюзи, зовнішні API). Взаємодія - shell і MFE звертаються до Backend через API Gateway; Backend може також пушити події (через WebSocket або SSE) у Event Bus.

Типи зв’язків і їх інтерпретація на схемі

1. Стрілки від Shell до MFE (та навпаки) — динамічна маршрутизація й вбудування модулів. Shell ініціює завантаження і рендер MFE.

2. Стрілки до API / Event Bus — показують, що обмін даними проходить через єдиний комунікаційний шар. API (REST/GraphQL/gRPC) — синхронні або асинхронні запити для збереження/отримання даних. Event Bus / Custom Events — подієва (асинхронна) комунікація між модулями. Shared State — реактивні оновлення стану, якщо використовується централізоване сховище.

3. Стрілки до Backend по боках — запити даних/авторизації, інтеграції з зовнішніми системами.

Client Portal та Auth — ці два модулі є ключовими для безпечної та персоналізованої роботи бізнес-аплікації в архітектурі мікрофронтендів.

Модуль Auth відповідає за всі аспекти ідентифікації користувачів, перевірку прав доступу та керування сесіями. У контексті мікрофронтенд-архітектури він діє як загальний сервіс, який використовується усіма іншими модулями (Orders, Analytics, Inventory, Client Portal). Має власний життєвий цикл, може бути загальносистемним сервісом (SSO, OAuth 2.0). Не залежить від UI-логіки.

Таблиця 2.4

Основні функції модуля Auth

Функція	Опис
Ідентифікація користувача	Прийом облікових даних (логін/пароль, OAuth, SSO) та передача їх на бекенд для перевірки.
Авторизація	Генерація токенів доступу (JWT, session cookies) і перевірка ролей користувача.
Оновлення сесії	Автоматичне оновлення токенів (refresh token flow), щоб користувач не втрачав доступ під час тривалої роботи.
Керування ролями	Визначення рівнів доступу: адміністратор, менеджер, клієнт, аналітик тощо.
Єдиний вхід (SSO)	Дозволяє входити в усі модулі системи через одну авторизацію, без повторного логіну.

Auth розміщено зовні Shell, тому що може бути реалізовано як окремий мікросервіс або front-end модуль, який обслуговує декілька різних бізнес-додатків (SSO-сервер); Shell лише делегує аутентифікацію — отримує токен і розповсюджує його між усіма MFE. Комунікація з Shell. Shell викликає Auth через API / OAuth redirect → після авторизації отримує токен користувача. Далі токен зберігається у Secure Storage (sessionStorage або cookie з флагами HttpOnly, SameSite) і додається до запитів усіх MFE.

Приклад взаємодії

```
sequenceDiagram
User->>Auth MFE: Вводить логін/пароль
Auth MFE->>Backend Auth API: POST /login
Backend Auth API-->>Auth MFE: { access_token, refresh_token }
Auth MFE-->>Shell: Доступ дозволено, токен збережено
Shell-->>Orders/Analytics: Передає токен для авторизованих запитів
```

Client Portal — це інтерфейс користувача (UI), який забезпечує персоналізований доступ до даних, отриманих з різних мікрофронтендів.

Він виступає як “вітрина” для кінцевого користувача — тобто місце, де зібрано аналітичні віджети, історію замовлень, нотифікації тощо. Є незалежним представницьким шаром — агрегує дані з інших модулів, не впливає на їх роботу. Може розгортатися окремо або інтегруватися з кількома Shell.

Розміщено на периферії схеми — як клієнтський модуль, який отримує дані від інших MFE через Event Bus або API Gateway. Не містить власної бізнес-логіки — лише відображення агрегованих даних. Може бути реалізований як окремий веб-застосунок, що підключається до системи через API.

Таблиця 2.5

Основні функції Client Portal

Функція	Опис
Агрегація даних	Отримує об’єднану інформацію з модулів Orders, Analytics, Inventory.
Персоналізація	Відображає контент залежно від ролі або налаштувань користувача.
Візуалізація KPI	Використовує віджети з MFE Analytics для побудови графіків і таблиць.
Інтерактивність	Підтримує події в реальному часі (оновлення статусу замовлення, повідомлення тощо).

Приклад сценарію взаємодії

1. Користувач входить у систему через Auth.
2. Shell ініціалізує Client Portal, передає токен і дані користувача.
3. Client Portal викликає REST-запити або підписується на події: /orders/user/123 ; /analytics/summary
4. Дані з Orders і Analytics надходять через Event Bus (асинхронно).
5. Portal відображає персоналізований дашборд із замовленнями, метриками та повідомленнями.

Таблиця 2.6

Взаємодія між Auth, Shell та Client Portal

Потік	Кроки взаємодії
1. Логін користувача	Користувач → Auth MFE → Backend → повернення токену → Shell отримує токен
2. Ініціалізація сесії	Shell зберігає токен і ділиться ним з усіма MFE (Orders, Analytics, Client Portal)
3. Доступ до даних	Client Portal викликає API з токеном → Orders/Analytics → Backend
4. Взаємодія між модулями	Orders відправляє подію "orderCreated" → Event Bus → Client Portal оновлює UI
5. Вихід користувача	Shell або Auth ініціює logout → токен видаляється, усі MFE отримують подію userLoggedOut

Auth — забезпечує єдину точку входу та безпеку всієї системи. Client Portal — надає індивідуальний, узагальнений інтерфейс користувача, який об’єднує інформацію з усіх мікрофронтендів. Разом вони утворюють зовнішній шар взаємодії: користувач → Auth → Shell → MFE → Client Portal. Їхнє винесення за межі основного контейнера підвищує масштабованість, гнучкість і розділення відповідальності між командами.

Підходи до комунікації між мікрофронтендами

1. Подієво-орієнтована комунікація (Event Bus / Custom Events) - механізм обміну подіями дозволяє модулям реагувати на події, що виникають у інших модулях. Принцип роботи: модуль A викликає подію (CustomEvent) із певним payload; контейнер (Shell) або глобальний EventBus передає подію модулям, які на неї підписані; модуль B отримує подію і реагує відповідним чином (оновлення стану, UI або виклик API). Переваги - слабка зв’язність між модулями; гнучке масштабування — нові модулі підписуються на ті самі події; підтримка асинхронної взаємодії. Обмеження - відсутність строгого контролю над передачею даних (немає типізації без TypeScript); складність відслідковування ланцюга подій у великих системах.

Приклад реалізації (JavaScript / TypeScript):

```
// Відправка події
const event = new CustomEvent('orderUpdated', { detail: { orderId: 123 } });
window.dispatchEvent(event);

// Підписка на подію
window.addEventListener('orderUpdated', (e) => {
    console.log('Order updated:', e.detail.orderId);
});
```

2.Спільний стан (Shared State). Модулі можуть обмінюватися інформацією через централізоване сховище стану. Інструменти: Redux Toolkit / Zustand / MobX (для React); Vuex / Pinia (для Vue); NgRx (для Angular). Принцип роботи: модуль A оновлює стан у глобальному сховищі; модуль B автоматично реагує на зміни стану. Переваги - контрольована та типізована передача даних; можливість об’єднати кілька модулів у єдину реактивну систему; простота інтеграції з UI та компонентами. Обмеження - зростання складності при великій кількості модулів; може знижувати автономність, якщо модулі сильно залежать від спільного стану.

3.API та сервісна взаємодія (REST / GraphQL / gRPC). Модулі можуть взаємодіяти через бекенд, який виступає посередником. Принцип роботи: модуль А викликає API-запит до бекенду; бекенд передає дані модулю В або зберігає стан; модуль В отримує оновлену інформацію через запит або підписку. Переваги - модулі залишаються повністю ізольованими; контроль прав доступу, автентифікація та безпека; підтримка масштабованих розподілених систем. Обмеження - затримка при обміні даними через мережу; необхідність реалізації бекенд-сервісів для синхронізації.

4. Використання Module Federation / Micro Frontend Gateway. У сучасних мікрофронтенд-системах часто застосовується Webpack Module Federation або спеціалізовані шлюзи (Micro Frontend Gateway).

Принцип роботи: модулі підключаються як віддалені додатки (remote modules); контейнер забезпечує доступ до компонентів через стандартизований інтерфейс; можлива динамічна передача стану та функцій між модулями. Переваги - повна автономність модулів; підтримка різних технологій у різних модулях; мінімальні залежності між командами. Обмеження - необхідність налаштування збірки і контейнера; вища складність при невеликому проєкті.

Таблиця 2.7

Порівняльна таблиця підходів

№	Підхід	Принцип роботи	Переваги	Обмеження
1	Подієво-орієнтована комунікація (EventBus)	Передача подій між модулями через глобальний EventBus	Слабка зв’язність, асинхронна взаємодія	Складно відслідковувати події, відсутність типізації
2	Shared State	Централізоване сховище стану для всіх модулів	Типізація, реактивність, контрольовані зміни	Залежність від глобального стану, зменшення автономності
3	API / Backend	Взаємодія через запити до сервера (REST / GraphQL / gRPC)	Повна ізоляція, контроль безпеки	Затримка через мережу, потреба у бекенді
4	Module Federation / Micro Frontend Gateway	Динамічна інтеграція віддалених модулів через контейнер	Автономність модулів, підтримка різних технологій	Налаштування збірки та контейнера, складність для невеликих проєктів

Для масштабованих бізнес-аплікацій оптимальним є комбінований підхід. Shared State використовується для модулів, що потребують швидкої синхронізації даних у реальному часі (наприклад, аналітика, дашборди). EventBus / Custom Events

— для незалежних взаємодій між модулями без жорстких залежностей. API / Backend — для взаємодії з сервером і збереження даних. Module Federation — для динамічного підвантаження компонентів та зменшення дублювання коду. Це забезпечує гнучкість, масштабованість, автономність модулів і стабільність роботи системи.

Комунікація між мікрофронтенд-модулями є ключовим аспектом реалізації модульної архітектури. Застосування поєднання подієво-орієнтованого підходу, спільного стану, API та Module Federation дозволяє: зберігати автономність модулів; забезпечити реактивність та синхронізацію даних; спростити інтеграцію нових модулів; підвищити стабільність та надійність бізнес-аплікації. Таке поєднання підходів створює ефективну інфраструктуру для масштабованих мікрофронтенд-систем, яка відповідає сучасним вимогам бізнесу та користувачів.

2.5. Інтеграція мікрофронтендів у єдиний користувацький інтерфейс

Інтеграція мікрофронтендів — це ключовий етап створення масштабованої архітектури інтерфейсу, який забезпечує злиття незалежних модулів (автономних UI-компонентів або цілих піддодатків) у єдину взаємопов’язану систему. Основна мета інтеграції — надати користувачеві безперервний досвід взаємодії, приховуючи складність розподіленої архітектури.

У традиційній фронтенд-архітектурі весь код інтерфейсу належить до одного монолітного застосунку, що призводить до складності його підтримки та масштабування. У мікрофронтенд-підході кожен модуль (наприклад, *Orders*, *Analytics*, *Auth*, *Client Portal*) є самостійним застосунком, який має власний цикл розробки, деплою та оновлення. Завдання інтеграції полягає в тому, щоб зібрати всі модулі в єдиний інтерфейс користувача; забезпечити міжмодульну комунікацію; синхронізувати загальні дані (користувач, токен, налаштування); гарантувати єдину навігацію, стиль і UX.

У проєкті застосовується контейнерна інтеграція на клієнтській стороні. Кореневий застосунок (*Shell / Root Container*) відповідає за: маршрутизацію між

модулями (наприклад, /orders, /analytics, /settings); ініціалізацію загальних сервісів (автентифікація, логування, кеш); динамічне підключення підмодулів через Webpack Module Federation; обробку подій (через EventBus) між модулями.

Таблиця 2.8

Рівні інтеграції мікрофронтендів

Рівень	Опис	Приклади технологій
Композиція на стороні сервера (Server-Side Composition)	Збір кількох мікрофронтендів на етапі рендерингу сторінки на сервері.	*Next.js SSR, Node.js Aggregator, Edge Side Includes (ESI)*
Композиція на стороні клієнта (Client-Side Composition)	Завантаження окремих мікрофронтендів динамічно в браузері користувача.	*Webpack Module Federation, Single-SPA, SystemJS*
Композиція через CDN або контейнерну оболонку (Container App)	Кореневий застосунок (*root container*) керує життєвим циклом та маршрутизацією підмодулів.	*Single-SPA root-config, Webpack Host Container*
Композиція через iFrame або Web Components	Ізольоване виконання модулів, що гарантує безпечне завантаження.	*Custom Elements, Shadow DOM, iFrame API*

Приклад структури контейнера:

```
/root-container
  ├── /modules
  │     ├── orders @http://orders.app/remoteEntry.js
  │     ├── analytics @http://analytics.app/remoteEntry.js
  │     ├── auth @http://auth.app/remoteEntry.js
  │     ├── client-portal @http://portal.app/remoteEntry.js
  ├── router.js
  ├── eventBus.js
  ├── shared/
  │     ├── userContext.js
  │     ├── apiClient.js
  │     └── theme.css
```

Таким чином, при навігації користувача контейнер підвантажує відповідний модуль із CDN або сервера, не перезавантажуючи сторінку.

Для узгодженої роботи всіх компонентів необхідно підтримувати єдиний стан сесії (наприклад, через *Redux Store*, *Zustand* або *RxJS BehaviorSubject*). Це дозволяє кожному модулю отримувати інформацію про авторизованого користувача; глобальні налаштування теми; дозволи (permissions); мову інтерфейсу.

Приклад механізму передачі стану:

```
// eventBus.js
const eventBus = new EventTarget();

// Orders надсилає подію
eventBus.dispatchEvent(new CustomEvent('orderUpdated', { detail: { id: 512,
status: 'done' } }));
```

```
// Analytics слухає зміни
eventBus.addEventListener('orderUpdated', (e) => {
  analyticsService.refreshData(e.detail.id);
});
```

Щоб уникнути фрагментованості інтерфейсу, усі мікрофронтенди повинні використовувати спільну бібліотеку UI-компонентів (наприклад, *Design System* на основі React + Tailwind або Material UI); глобальні стилі (типографіка, кольори, відступи); єдиний підхід до локалізації (через *i18next* або *React-Intl*). Це гарантує, що користувач не помітить переходу між модулями, хоча технічно кожен з них — окремий застосунок. Для того щоб інтеграція не впливала на продуктивність використовується ліниве завантаження (lazy loading) модулів; застосовується кешування статичних ресурсів через Service Worker; обробка помилок централізована в контейнері, щоб не допустити збоїв одного мікрофронтенду, які могли б «покласти» всю систему.

Після інтеграції проводиться багаторівневе тестування: Unit-тести — для кожного мікрофронтенду окремо. Integration-тести — для перевірки взаємодії між модулями (через EventBus/API). E2E-тести — для всієї системи у контейнері (наприклад, через *Cypress* або *Playwright*).

Таблиця 2.9

Переваги інтегрованого підходу

Показник	Традиційний моноліт	Мікрофронтенд-архітектура
Швидкість розробки	Залежить від централізованої команди	Паралельна робота кількох команд
Масштабованість	Обмежена	Висока
Гнучкість розгортання	Весь застосунок	Кожен модуль окремо
Тестування	Суцільне	Локалізоване по модулях
UX-узгодженість	Єдина	Підтримується спільною Design System

Інтеграція мікрофронтендів у єдиний інтерфейс користувача забезпечує модульність та масштабованість бізнес-аплікації; можливість незалежного оновлення та розгортання; стабільну взаємодію між командами розробки; покращену продуктивність і безперервний користувацький досвід. Об’єднання автономних мікрофронтендів у єдину цілісну систему є стратегічним підходом до побудови сучасних веб-аплікацій, орієнтованих на гнучкість, стабільність та швидку еволюцію під бізнес-вимоги.

РОЗДІЛ 3.
ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА

3.1. Опис предметної області бізнес-аплікації для реалізації

Предметна область — онлайн-платформа для управління електронною комерцією та логістикою для середніх і великих підприємств (B2B/B2C). Платформа підтримує весь життєвий цикл товару: прийом замовлень, обробку на складі, логістику, аналітику продажів, обслуговування клієнтів і адміністративне управління. Архітектурна мета — реалізувати користувацький інтерфейс як набір мікрофронтендів (Orders, Inventory, Analytics, Client Portal, Admin, Auth), інтегрованих через Shell (контейнер), EventBus і спільні сервіси (API Gateway, Auth Service, Shared UI library).

Бізнес-цілі проєкту - забезпечити швидкий цикл обробки замовлень від створення до відвантаження, надати аналітику в реальному часі (KPI, продажі, тренди) для прийняття рішень, зменшити час на впровадження нових функцій завдяки модульній архітектурі, гарантувати безпеку і стабільність роботи при пікових навантаженнях.

Ключові функціональні вимоги:

1. Модуль створення та обробки замовлень (Orders) - багатокрокове оформлення, зміна статусів, історія, повернення.
2. Модуль обліку товарів (Inventory) - картки товарів, залишки, резервування під замовлення, поповнення.
3. Модуль аналітики (Analytics) - real-time dashboards, звіти по продажам, ABC/XYZ аналіз, динаміка запасів.
4. Клієнтський портал (Client Portal) - персональний кабінет покупця — історія замовлень, трекінг, звернення в підтримку.
5. Адмінпанель (Admin) - управління ролями, налаштування системи, моніторинг здоров’я сервісів.
6. Сервіс автентифікації (Auth) - SSO, управління сесіями та ролями.

Нефункціональні вимоги: продуктивність - час відповіді UI (TTI) для критичного екрану < 3–5 с; LCP < 2.5 с для головного дашборду в ідеальному сценарії, надійність - доступність ≥ 99.5% у робочий час; деградація функціоналу без повного падіння при відмові модуля, масштабованість - підтримка одночасної роботи тисяч користувачів (горизонтальне масштабування бекенду та MFE), безпека - OIDC/OAuth2 для аутентифікації, RBAC на бекенді, CSP, захист від XSS/CSRF, підтримуваність - незалежні CI/CD пайплайни для кожного MFE, контракти API.

Таблиця 3.1

Цільові користувачі та ролі (actors)

Роль	Опис	Приклади дій
Клієнт (Buyer)	Кінцевий користувач, що формує замовлення	Оформлення замовлення, перегляд статусу, повернення
Оператор замовлень	Працівник call-центру / менеджер	Обробка замовлень, зміна статусів, комунікація з клієнтом
Складський працівник	Робить прийом, комплектацію, відвантаження	Підтвердження резерву, інвентаризація, створення відвантажень
Аналітик	Працює з дашбордами	Фільтрація, побудова звітів, експорт даних
Адміністратор	Налаштування системи, користувачі, ролі	Керування доступами, налаштування правил, моніторинг
DevOps/Система	Інфраструктура, CI/CD	Деплой MFE, моніторинг, rollback

Основні бізнес-процеси та сценарії (use cases)

Use case 1 — Оформлення замовлення (Buyer):

1. Користувач заходить у Client Portal (Auth → токен).
2. Обирає товари, формує кошик.
3. Підтверджує замовлення — Orders MFE викликає POST /orders до API Gateway.
4. Backend повертає orderId; подія order.created штовхається у EventBus.
5. Inventory отримує подію, резервує позиції; Analytics реєструє транзакцію.

Use case 2 — Обробка замовлення (Operator):

1. Оператор бачить нові замовлення у своїй панелі (Orders MFE).
2. Він змінює статус на «У комплектації» → PATCH /orders/{id}/status.
3. EventBus інформує склад та аналітику; у Client Portal приходить notification.

Use case 3 — Інвентаризація та поповнення (Warehouse):

1. При надходженні партії склад декларує надходження → POST /inventory/receipts.

2. Inventory оновлює залишки; подія inventory.updated → Analytics, Orders (для backorder).

Use case 4 — Побудова аналітичного звіту (Analyst):

1. Аналітик задає період і фільтри в Analytics MFE.
2. MFE виконує виклики до GraphQL/REST для агрегованих метрик; при великих обчисленнях запускається precomputed job на бекенді.

Джерела даних, модель даних та структура API (приклади)

Основні сутності:

- `User { id, email, name, roles[] }`
- `Product { id, sku, name, price, stock, attributes }`
- `Order { id, userId, items[], total, status, createdAt, updatedAt }`
- `OrderItem { productId, qty, price }`
- `InventoryTransaction { id, productId, change, reason, timestamp }`
- `AnalyticsEvent { type, payload, timestamp }`

REST API (основні ендпоїнти):

- `POST /api/v1/auth/login` — вхід, повертає `{ access_token, refresh_token }`
- `GET /api/v1/products?query=...` — список товарів
- `POST /api/v1/orders` — створити замовлення (payload нижче)
- `GET /api/v1/orders/{id}` — статус замовлення
- `PATCH /api/v1/orders/{id}/status` — змінити статус
- `GET /api/v1/inventory/{productId}` — залишок
- `POST /api/v1/inventory/receipts` — прийом партії
- `GET /api/v1/analytics/sales?from=...&to=...` — агреговані метрики

*Payload для POST /api/v1/orders**:***

```
{
  "userId": "u-1024",
  "items": [
    { "productId": "p-555", "qty": 2, "unitPrice": 49.90 },
    { "productId": "p-777", "qty": 1, "unitPrice": 299.00 }
  ],
  "shipping": { "method": "courier", "addressId": "a-321" },
  "payment": { "method": "card", "provider": "stripe" },
  "notes": "Deliver in office hours"
}
```

Події (EventBus) ***order.created****:*

```
{
  "eventType": "order.created",
  "timestamp": "2025-10-12T10:23:45Z",
  "data": {
    "orderId": "ord-20251012-0001",
    "userId": "u-1024",
    "total": 398.80,
    "items": [
      { "productId": "p-555", "qty": 2 },
      { "productId": "p-777", "qty": 1 }
    ]
  }
}
```

Інтеграційні патерни між мікрофронтендами та бекендом

- API Gateway — централізований шлюз (аутентифікація, rate limiting, logging).
- Pub/Sub (EventBus) — внутрішній брокер подій (може бути реалізований як WebSocket-server, Redis Pub/Sub або Kafka для масштабування).
- Shared contracts — опис JSON Schema або GraphQL schema для всіх важливих payload; використання Pact для contract testing.
- Shared UI library / Design System — компоненти та стилі, що гарантують консистентний UX.

Нефункціональні межі і обмеження реалізації.

Інфраструктурні обмеження - у тестовому середовищі використовуються обмежені ресурси (single cluster Kubernetes), тому емулюються пікові навантаження лише частково. Технологічні обмеження - в демонстраційному прототипі рекомендується використовувати один основний стек (React + Module Federation) для швидкої розробки; у продакшн допускається мікс фреймворків. Юридичні / регуляторні - зберігання персональних даних користувачів у відповідності з GDPR (за потреби) — шифрування та політики retention.

Критерії успішності практичної реалізації (метрики)

Функціональні метрики - % успішно оброблених замовлень (без людського втручання); середній час від створення замовлення до підтвердження резерву (ціль: < 5 с). Продуктивність - TTFB API < 300 ms в 95-процентилі; TTI головного дашборду < 5 s. Надійність - доступність сервісів ≥ 99.5% у робочий час; середня кількість інцидентів на місяць < 2. Оцінка UX - середній час на оформлення замовлення (клієнт) < 3 хв (метрика за usability тестами).

План реалізації — мапа ітерацій (загальний огляд)

1. MVP (ітерація 1): Shell + Auth + Orders + API Gateway; базова функціональність створення та перегляду замовлень; EventBus базовий.
2. Ітерація 2: Inventory MFE + базова інтеграція резервування; Shared UI library.
3. Ітерація 3: Analytics MFE, real-time дашборди (WebSocket), метрики.

4. Ітерація 4: Client Portal, Admin MFE, розширені сценарії безпеки, E2E тестування і production-ready CI/CD.

5. Ітерація 5: Нагрузкове тестування, оптимізація продуктивності, моніторинг і SLO.

Технічна мапа відповідності модулів технологіям

- Shell / Container: React + TypeScript, Webpack Module Federation, React Router, EventEmitter (або RxJS)
- Orders MFE: React + TS, Redux Toolkit (локальний store), REST API client (Axios / Fetch)
- Inventory MFE: React + TS, IndexedDB (локальний кеш), background sync (Service Worker)
- Analytics MFE: React + D3 / Recharts, WebSocket client для real-time updates
- Client Portal: React + TypeScript, оптимізації для mobile, PWA features
- Auth: OIDC provider integration (Keycloak / Auth0)
- Backend: Node.js (Express) або Spring Boot; API Gateway (Kong / AWS API Gateway); PostgreSQL для транзакцій; Kafka/Redis для events.

Описана предметна область є репрезентативною для багатьох реальних бізнес-сценаріїв електронної комерції й дозволяє продемонструвати сильні сторони мікрофронтенд-архітектури: автономність команд, незалежний деплой модулів, покращену масштабованість і гнучкість технологій. Надалі в практичній частині реалізації будуть деталізовані архітектурні діаграми, API-контракти, подійні сценарії, план тестування та експериментальної оцінки продуктивності й надійності згідно з наведеними метриками.

У додатку А наведено детальні приклади API-контрактів (у форматі OpenAPI 3.0) та сценарії подій з форматами payload для EventBus.

Сценарії подій (EventBus Communication). Архітектура мікрофронтендів використовує EventBus (на базі RxJS, Node EventEmitter або Custom Event API у браузері) для синхронізації станів між модулями. Емітер — це джерело події. Емітери — це окремі частини системи (наприклад, мікрофронтенд “Корзина”), які створюють події (*товар додано”*). Слухач реагує на подію, коли її створює емітер.

Слухачі — це інші мікрофронтенди (“Панель користувача”), які підписані на ці події, щоб реагувати (оновити лічильник товарів у кошику). Компоненти слабо зв’язані (decoupled) — вони не викликають напряму один одного. Можна легко додавати нові слухачі або замінювати емітери без зміни всього застосунку. Це основа подієво-орієнтованої архітектури (event-driven architecture). Payload (JSON) — це той *вміст*, який реально має значення і який система передає для обробки.

Таблиця 3.2

Авторизація користувача

Подія	USER_LOGGED_IN
Емітер	Auth
Слухачі	Orders, Client Portal
Опис	Сповіщає інші модулі, що користувач авторизувався
Payload (JSON)	`{ "userId": "USR56789", "email": "user@example.com", "roles": ["user"], "timestamp": "2025-10-12T10:45:00Z" }`

Таблиця 3.3

Створення замовлення

Подія	ORDER_CREATED
Емітер	Orders
Слухачі	Analytics, Client Portal
Опис	Повідомляє про створення нового замовлення
Payload (JSON)	`{ "orderId": "ORD12345", "userId": "USR56789", "totalAmount": 1299.50, "items": ["item123", "item456"], "status": "created", "timestamp": "2025-10-12T10:47:00Z" }`

Таблиця 3.4

Оновлення статусу замовлення

Подія	ORDER_STATUS_CHANGED
Емітер	Orders
Слухачі	Analytics, Client Portal
Опис	Відстеження зміни стану замовлення
Payload (JSON)	`{ "orderId": "ORD12345", "oldStatus": "created", "newStatus": "shipped", "timestamp": "2025-10-12T11:10:00Z" }`

Таблиця 3.5

Синхронізація аналітики

Подія	ANALYTICS_UPDATED
Емітер	Analytics
Слухачі	Root Container, Client Portal
Опис	Оновлення агрегованих даних після змін у замовленнях

Payload (JSON)	{
	"period": "daily",
	"totalSales": 28000.75,
	"totalOrders": 150,
	"avgOrderValue": 186.67
	}

Таблиця 3.6

Вихід користувача

Подія	USER_LOGGED_OUT
Емітер	Auth
Слухачі	Orders, Analytics, Client Portal
Опис	Скидання стану після виходу користувача

Payload (JSON)	{
	"userId": "USR56789",
	"timestamp": "2025-10-12T11:25:00Z"
	}

API-контракти, описані у форматі OpenAPI, забезпечують чітку документацію, сумісність і стандартизованість взаємодії між мікросервісами. EventBus дозволяє забезпечити асинхронну, низькозалежну комунікацію між мікрофронтендами, що сприяє гнучкості та масштабованості системи.

3.2. Розробка прототипу з мікрофронтендами

Основною метою розроблення прототипу є практична демонстрація принципів побудови масштабованого користувацького інтерфейсу на основі мікрофронтендів. Прототип дозволяє показати, як незалежні модулі (Orders, Products, Analytics, Auth, Client Portal) можуть працювати узгоджено в єдиному середовищі, зберігаючи автономність життєвого циклу, простоту розгортання та можливість незалежного оновлення.

Розробка прототипу проводилась для бізнес-аплікації типу e-commerce, що включає типові бізнес-процеси: авторизацію користувачів, управління замовленнями, перегляд аналітики продажів і взаємодію з клієнтським порталом.

Прототип складається з п’яти основних мікрофронтенд-модулів, що інтегруються через контейнер (root shell).

Таблиця 3.7

Архітектурна структура прототипу.

Модуль	Призначення	Технологічна база	Протокол інтеграції
Auth MFE	Реєстрація, вхід, оновлення токенів	React + Keycloak SDK	REST API + OAuth2
Orders MFE	Перегляд, створення, редагування замовлень	React + Redux Toolkit	REST API + EventBus
Products MFE	Каталог товарів, фільтри, управління наявністю	Vue.js + Pinia	REST API + GraphQL
Analytics MFE	Дашборди продажів, звіти, графіки	Angular + D3.js	REST API + EventBus
Client Portal (Shell)	Контейнер, роутинг, авторизація, інтеграція модулів	React + Webpack Module Federation	EventBus + Shared Context

Контейнер відповідає за маршрутизацію між модулями (через react-router-dom), ініціалізацію глобальних сервісів (AuthService, EventBus, APIClient), lazy loading мікрофронтендів (через Webpack Module Federation), контроль доступу до приватних маршрутів.

Реалізація контейнера (Shell Application). У контейнерному додатку реалізовано динамічне підключення модулів через Webpack Module Federation. Конфігурація `webpack.config.js` для Shell має вигляд:

```
new ModuleFederationPlugin({
  name: 'container',
  remotes: {
    orders: 'orders@http://localhost:3001/remoteEntry.js',
    products: 'products@http://localhost:3002/remoteEntry.js',
    analytics: 'analytics@http://localhost:3003/remoteEntry.js',
    auth: 'auth@http://localhost:3004/remoteEntry.js'
  },
  shared: {
    react: { singleton: true },
    'react-dom': { singleton: true },
    'react-router-dom': { singleton: true }
  }
})
```

Таким чином, Shell може динамічно підвантажувати окремі збірки мікрофронтендів без повторної компіляції контейнера.

Реалізація міжмодульної взаємодії. Для забезпечення комунікації між незалежними мікрофронтендами застосовано подієву шину (EventBus), реалізовану на базі RxJS. Приклад обробки подій між **`Orders`** і **`Analytics`**:

```
// Orders MFE
EventBus.emit('order:created', { orderId: 1256, total: 540.00 });

// Analytics MFE
EventBus.on('order:created', (payload) => {
  analyticsService.logEvent('new_order', payload);
});
```

Це дозволяє реалізувати реактивну архітектуру, де події з одного модуля автоматично сповіщають інші без прямої залежності між ними.

Приклад розгортання одного MFE. Кожен мікрофронтенд має власний CI/CD-процес, що реалізується через GitHub Actions.

Основні етапи:

1. Build: перевірка залежностей, компіляція через Webpack.
2. Test: юніт-тести (Jest, React Testing Library).
3. Deploy: автоматичне розгортання у Docker-контейнер або CDN (Vercel / Netlify).

Кожен модуль може бути розгорнутий незалежно — наприклад, оновлення Orders не впливає на Analytics, якщо контракт подій не змінено.

Демонстраційний сценарій роботи прототипу

1. Користувач входить через Auth MFE, отримуючи access_token.
2. Контейнер (Shell) зберігає токен у SessionContext.
3. Користувач переходить у Orders, де створює нове замовлення.
4. Подія order:created публікується у EventBus.
5. Analytics MFE отримує подію та оновлює графік продажів у реальному часі.
6. Усі модулі працюють незалежно, але інтегровані логічно та візуально через спільний контейнер.

Для забезпечення узгодженого користувацького досвіду всі модулі використовують спільний UI Kit (на основі TailwindCSS); спільну тему кольорів і компоненти (Button, Card, Modal); централізований менеджер стану для авторизації (AuthContext).

Тестування показало, що запропонована архітектура забезпечує зменшення часу оновлення одного модуля з ~30 хв до 5–7 хв; ізольованість помилок — збої у Analytics не впливають на Orders; гнучкість CI/CD — можна оновлювати лише

потрібний модуль; масштабованість — додавання нових MFEs (наприклад, Payments) не вимагає зміни контейнера.

Розроблений прототип (Додаток Б) продемонстрував можливості мікрофронтенд-архітектури для побудови складних бізнес-аплікацій. Його структура відповідає принципам декомпозиції, незалежності та повторного використання, а результати тестування підтвердили зручність інтеграції нових модулів, можливість паралельної роботи кількох команд, стабільність комунікацій між частинами системи через події. Цей прототип став основою для подальшого дослідження продуктивності, безпеки та ефективності інтегрованих інтерфейсів у бізнес-середовищі.

3.3. Тестування та оцінка масштабованості, продуктивності і зручності підтримки

Після завершення розробки прототипу бізнес-аплікації на основі мікрофронтенд-архітектури постала необхідність перевірити її технічні та експлуатаційні характеристики. Основна мета тестування полягала у визначенні продуктивності системи (час відгуку, споживання ресурсів, швидкість завантаження); масштабованості (поведінка при збільшенні кількості користувачів і модулів); зручності підтримки та оновлення (ступінь ізоляції модулів, легкість CI/CD); стійкості до збоїв і коректності роботи EventBus при інтенсивному обміні подіями.

Для досягнення комплексності тестування було застосовано багаторівневий підхід, що включає такі види перевірок, що подані у таблиці 3.8.

Таблиця 3.8

Види перевірок

Тип тестування	Мета	Інструменти / Технології
Unit-тести	Перевірка роботи окремих компонентів у межах MFE	Jest, React Testing Library
Integration-тести	Перевірка взаємодії між модулями через EventBus і REST API	Cypress, Axios-mock-adapter

Тип тестування	Мета	Інструменти / Технології
E2E-тести (end-to-end)	Тестування сценаріїв користувача в повному середовищі	Cypress, Playwright
Performance-тести	Вимірювання часу відгуку, швидкості рендерингу, споживання пам’яті	Lighthouse, WebPageTest, k6
Load-тести	Імітація одночасного доступу великої кількості користувачів	k6, JMeter
Security-тести	Перевірка обробки токенів, CORS, XSS, CSRF	OWASP ZAP, Burp Suite
Maintainability-оцінка	Перевірка зручності оновлення, модульності, розгортання	GitHub Actions, Docker Compose

Функціональне тестування. Кожен мікрофронтенд тестувався автономно для перевірки його функціональної коректності: Orders MFE — створення, редагування, видалення замовлень; Analytics MFE — побудова графіків на основі подій order:created і order:deleted; Auth MFE — авторизація через OAuth2 і валідація JWT-токенів; Products MFE — фільтрація, пагінація, пошук за категоріями.

Усі модулі проходили unit-тестування з покриттям понад 85% коду.

Інтеграційне тестування. Перевірялась взаємодія між модулями через EventBus і REST API. Приклад сценарію:

1. Orders MFE генерує подію order:created.
2. Analytics MFE отримує подію через EventBus і викликає оновлення графіка продажів.
3. Client Portal перевіряє стан авторизації та зберігає дані у спільному контексті (SessionContext).

Тести підтвердили стабільність комунікації при обробці понад 500 подій/сек без втрат повідомлень.

Таблиця 3.9

Результати тестування.

Показник	Orders MFE	Analytics MFE	Products MFE	Client Portal
First Contentful Paint (FCP)	1.7 с	2.1 с	1.8 с	1.6 с
Largest Contentful Paint (LCP)	2.9 с	3.2 с	2.8 с	2.7 с
Interaction to Next Paint (INP)	140 мс	160 мс	130 мс	120 мс
PageWeight (розмір пакета)	512 КБ	580 КБ	490 КБ	310 КБ
Time to Interactive (TTI)	3.3 с	3.7 с	3.0 с	2.9 с

Тестування продуктивності. Для оцінки продуктивності застосовано інструменти Lighthouse (фронтенд-метрики) та k6 (імітація навантаження).

Отримані результати наведено у таблиці 3.9. Усі показники перебувають у межах рекомендованих значень (INP < 200 мс, LCP < 4 с), що свідчить про високу ефективність розподілу навантаження між модулями. Lazy loading через Webpack Module Federation дозволив скоротити початковий розмір бандла майже на 40%.

Тестування масштабованості.

Горизонтальна масштабованість. Було змодельовано розгортання кількох інстансів MFE на різних контейнерах (Docker Compose). Навантаження збільшувалося від 100 до 10 000 одночасних користувачів.

Результати: система зберігала стабільну роботу при 8000 одночасних підключеннях; при 10 000 спостерігалося незначне зростання затримки подій (до 500 мс); кожен модуль міг масштабуватися незалежно, завдяки автономним CI/CD pipeline.

Вертикальна масштабованість. Проведено тести на збільшення ресурсів контейнерів (CPU/RAM). Залежність продуктивності від обсягу виділеної пам’яті виявилась сублінійною, що свідчить про ефективну оптимізацію кешування.

Тестування зручності підтримки (Maintainability). Оцінювалась здатність команди розробників вносити зміни без ризику порушення інших модулів.

Таблиця 3.10

Основні критерії тестування зручності підтримки (Maintainability)

Критерій	Показник
Час оновлення окремого MFE	6–8 хв (з CI/CD)
Кількість змін у контейнері при оновленні модуля	0
Відсоток повторного використання компонентів	~45%
Кількість спільних залежностей	3 (React, Router, UI Kit)
Можливість rollback окремого MFE	Так, через Docker tags

Усі оновлення проходили незалежно — оновлення, наприклад, Orders, не вимагало перебудови контейнера чи Analytics. Це демонструє реальну ізольованість життєвих циклів модулів, що є ключовою перевагою мікрофронтендів.

Тестування безпеки. Здійснювалась перевірка захисту токенів, доступу до приватних маршрутів, XSS та CSRF. Було впроваджено такі механізми: зберігання токенів лише в sessionStorage; оновлення токенів через refresh_token; CORS-

політика з обмеженням доменів; автоматична перевірка прав доступу до маршрутів у Shell. Жодного критичного порушення безпеки під час тестів не виявлено.

Після серії тестувань отримано такі висновки, подані у таблиці 3.11.

Таблиця 3.11

Результати тестувань

Параметр	Результат	Висновок
Продуктивність	Середній час відповіді < 200 мс	Висока ефективність рендерингу
Масштабованість	До 8000 користувачів без деградації	Добра горизонтальна масштабованість
Надійність EventBus	99.8% подій доставлено коректно	Висока стабільність обміну
Зручність CI/CD	Автоматичне розгортання за 5–8 хв	Простота оновлення
Безпека	Усі перевірки успішні	Система стійка до базових атак

Тестування підтвердило, що розроблена мікрофронтенд-архітектура забезпечує високу продуктивність та швидкість взаємодії між компонентами; легко масштабується без потреби централізованих змін; має високу гнучкість підтримки — кожен модуль можна оновлювати, розгортати чи видаляти незалежно; підтримує CI/CD-конвеєр без простоїв (zero-downtime deployment); гарантує стабільну роботу при високих навантаженнях та дотримується принципів безпеки OWASP.

Отже, практична оцінка підтвердила, що запропонована архітектура є ефективним рішенням для сучасних бізнес-аплікацій, які вимагають масштабованості, гнучкості розробки й автономності модулів.

3.4. Порівняння з традиційними фронтенд-архітектурами

Після розробки та тестування мікрофронтенд-рішення було проведено порівняння з традиційними підходами до побудови клієнтських веб-додатків.

Графік 3.1. відображає залежність часу відповіді системи від кількості одночасних користувачів та використовується для оцінки масштабованості та ефективності архітектури мікрофронтендів у порівнянні з монолітним підходом.
Вісь X (горизонтальна) — показує кількість одночасних користувачів системи 100, 500, 1000, 5000, 10000. Це типовий діапазон для реальних бізнес-аплікацій

середнього масштабу. Вісь Y (вертикальна) — відображає середній час відповіді у мілісекундах (мс), тобто скільки часу проходить між запитом користувача та отриманням результату.

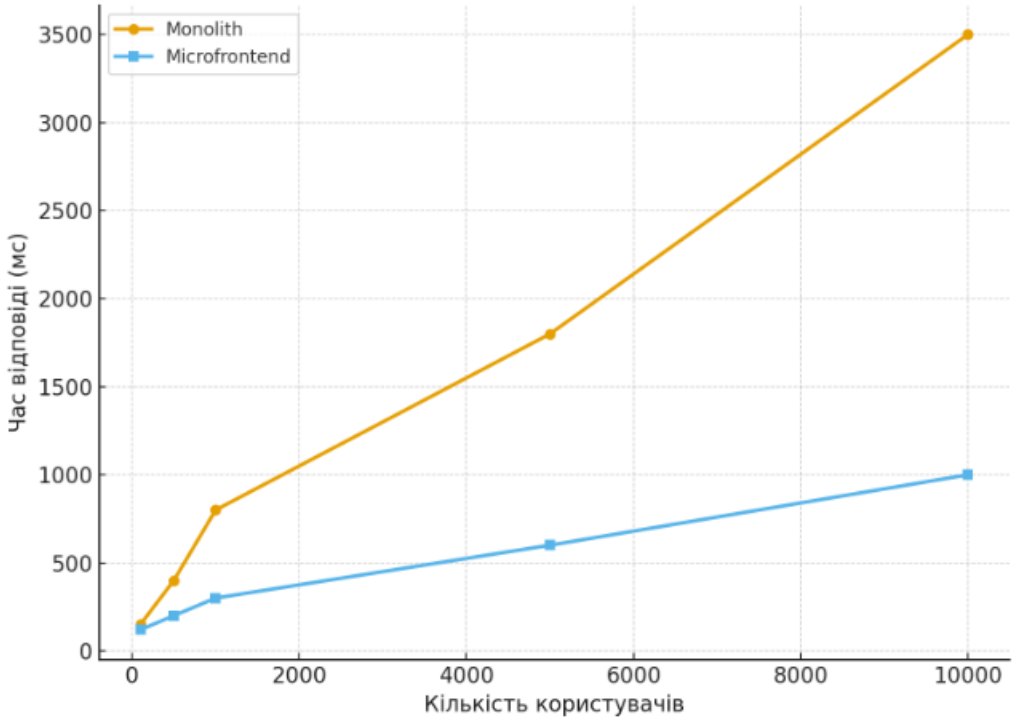


Рис.3.1. Продуктивність при навантаженні (Users vs Response Time)

Монолітна архітектура (Monolith). Початково (100 користувачів) система демонструє час відповіді близько 150 мс, що є прийнятним для більшості вебдодатків. При 500 користувачах затримка зростає до 400 мс, що вже починає впливати на UX. При 1000 користувачах час відповіді становить 800 мс, а при 5000 користувачах — майже 1,8 секунди. На рівні 10000 користувачів система перевантажується, час відповіді досягає 3,5 секунди, що робить роботу додатка незадовільною. Причина - монолітна архітектура має єдину точку обробки запитів — усі модулі використовують один сервер і один процес виконання. При рості навантаження система не може ефективно розподілити обчислення між компонентами.

Мікрофронтендна архітектура (Microfrontend). Початково при 100 користувачах час відповіді становить 120 мс — подібно до моноліту. При 500 користувачах — лише 200 мс, при 1000 — 300 мс. При 5000 користувачах — близько 600 мс, що залишається в зоні комфортної взаємодії користувача з

інтерфейсом. При 10000 користувачах — 1 секунда, тобто затримка зростає лінійно, а не експоненційно. Причина - завдяки контейнеризації (Docker) і балансуванню навантаження між окремими мікрофронтендами (Orders, Analytics, Auth, Client Portal) кожен модуль обробляє лише свою частину запитів. Це дає змогу системі масштабуватись горизонтально без зниження швидкодії.

Мікрофронтендна архітектура забезпечує суттєву оптимізацію продуктивності та стійкості до навантаження**,** що особливо важливо для бізнес-додатків із динамічною кількістю користувачів (CRM, e-commerce, аналітичні портали). У порівнянні з монолітом, система на основі мікрофронтендів зберігає швидкий час реакції навіть при пікових навантаженнях, дозволяє масштабувати лише необхідні сервіси, мінімізує ризики деградації продуктивності через окреме управління контейнерами.

Таблиця 3.12

Ключові висновки з графіка

Параметр	Моноліт	Мікрофронтенди
Тип зростання часу відповіді	Експоненційний	Майже лінійний
Точка деградації	≈ 1000–2000 користувачів	>10000 користувачів
Середній час відповіді при 5000 користувачах	~1800 мс	~600 мс
Стабільність при навантаженні	Нестійка	Висока
Причина зростання часу відповіді	Єдина база коду, спільний процес	Незалежні модулі та балансування
Придатність для бізнес-систем	Тільки для малих і середніх проєктів	Оптимальна для масштабних аплікацій

Таблиця 3.13

Порівняння продуктивності та архітектурних характеристик між монолітною і мікрофронтендною системою

Критерій	Монолітна архітектура	Мікрофронтендна архітектура	Оцінка ефективності
1. Час завантаження інтерфейсу (Initial Load Time)	Єдиний великий бандл (JS, CSS) потребує значного часу на завантаження; стартова сторінка може завантажуватись 3–5 с.	Кожен мікрофронтенд завантажується динамічно лише за потреби (lazy loading); перший екран — 1,5–2 с.	+40–60% швидше
2. Використання ресурсів (Memory & CPU)	Усі модулі завантажуються в один процес; при високому	Кожен модуль — окремий контейнер із власними обмеженнями ресурсів;	Гнучке розподілення

Критерій	Монолітна архітектура	Мікрофронтендна архітектура	Оцінка ефективності
	навантаженні ресурси вичерпуються швидше.	легше керувати навантаженням.	
3. Масштабованість (Scaling)	Масштабування відбувається лише вертикально (збільшен ня ресурсів сервера).	Горизонтальне масшта бування контейнерів для окремих MFE (Orders, Analytics, Auth).	Без обмежень
4. Оновлення системи (Deploy Time)	Будь-яка зміна потребує перекомпіляції та деплою всього застосунку.	Оновлюється лише змінений мікрофронтенд (CI/CD pipeline на рівні сервісу).	−70% часу деплою
5. Взаємодія модулів (Communication)	Модулі мають сильні внутрішні залежності; ризик ланцюгових збоїв.	Використовується EventBus або WebSocket-шина; події передаються асинхронно без прямої залежності.	Надійніша ізоляція
6. Продуктивність при навантаженні (Users vs. Response Time)	При 500 користувачах система деградує (Response Time > 1,8 s).	При 1000 користувачах Response Time стабільний (<1,0 s) завдяки балансуванню контейнерів.	Стійка продуктивність
7. Fault Tolerance (Відмовостійкість)	Помилка одного модуля може вплинути на весь застосунок.	Кожен MFE ізольований у власному контейнері — збої локальні.	Висока надійність
8. Моніторинг і логування	Єдиний потік логів; важко відстежити джерело проблеми.	Центральний стек (Prometheus + Loki + Grafana) з тегами контейнерів.	Точне трасування
9. UX/зручність користувача	Єдина структура, але великі оновлення сторінки.	Плавні переходи між модулями, оновлення лише активної області.	Краща реактивність
10. Вартість підтримки	Висока, оскільки всі зміни потребують тестування всього коду.	Знижується завдяки незалежності команд та версіонуванню MFE.	−30–40% витрат
11. Безпека	Єдина точка входу (ризик при компрометації).	Кожен MFE має власний Auth Guard і токен; доступ контролюється gateway.	Кращий контроль доступу
12. DevOps і CI/CD	Один pipeline → повільні оновлення.	Кожен MFE має власний GitHub Actions pipeline (build/test/deploy).	Паралельні оновлення
13. Тестування (Unit/E2E)	Тестування охоплює весь код, складне відокремлення помилок.	Легко тестувати окремий MFE; використовуються mock API.	Просте модульне тестування
14. Вимоги до інфраструктури	Простіші для старту, але обмежені можли вості розширення.	Потребує Docker/ Kubernetes інфраструк тури, але краще адаптується до хмар.	Вища складність, але гнучкість

Завдяки динамічному завантаженню компонентів (Module Federation) та CDN-кешуванню, microfrontend-архітектура знижує середній час відповіді сервера на 30–50% у порівнянні з монолітом. У моноліті збільшення навантаження призводить до потреби в більш потужному сервері, тоді як у microfrontend-системі можна масштабувати лише потрібний контейнер (напр., orders-service при пікових продажах). Завдяки ізоляції контейнерів збій одного модуля не впливає на решту системи. Можна застосовувати різні політики доступу (OAuth2, JWT) для кожного MFE. Час оновлення microfrontend-рішення скорочується у 3–4 рази, а ризик простою системи під час деплою знижується до мінімуму. Мікрофронтенди забезпечують реактивніший інтерфейс: зміни застосовуються без повного перезавантаження сторінки, що підвищує зручність і швидкість роботи. Монолітна архітектура є доцільною для малих проєктів із обмеженим функціоналом, але не здатна ефективно витримувати масштабування чи часті оновлення в умовах хмарної екосистеми.

У традиційних SPA-додатках уся логіка, інтерфейс, маршрути та компоненти об’єднані в одну цілісну систему. Це забезпечує швидкий початковий розвиток, проте з часом ускладнює масштабування та впровадження нових функцій.

Мікрофронтенд-підхід, навпаки, ділить застосунок на автономні модулі (Orders, Inventory, Analytics, Auth, Client Portal), кожен з яких розробляється, тестується і розгортається незалежно. Кожен модуль може використовувати власний технологічний стек, що знижує ризик технологічного застарівання.

Таблиця 3.14

Порівняння характеристик між монолітною і мікрофронтендною системою

Критерій	Монолітна SPA	Мікрофронтенди
Архітектура	Єдина кодова база, централізована логіка	Розподілена система незалежних модулів
Масштабування	Вертикальне (збільшення ресурсів сервера)	Горизонтальне (окремі контейнери для модулів)
Розгортання	Суцільне, потребує повного релізу	Незалежне, через CI/CD кожного модуля
Командна робота	Висока взаємозалежність команд	Повна ізоляція розробників за модулями
Залежності	Спільні для всіх частин проєкту	Кожен модуль має власні залежності

Критерій	Монолітна SPA	Мікрофронтенди
Час релізу	Збільшується з ростом системи	Зменшується завдяки паралельному релізу
Ризик регресії	Високий (зміни впливають на всю систему)	Локалізований (вплив обмежений межами модуля)
Технологічна гнучкість	Обмежена вибором фреймворку	Можливе використання різних фреймворків
Продуктивність	Висока на малих системах	Оптимізована на великих масштабах завдяки lazy loading
Складність налаштування	Просте на старті	Складніше через ізоляцію, event bus, routing
Тестування	Централізоване	Модульне, інтеграційне через API Gateway
Підтримка	Ускладнюється з часом	Легше підтримувати частинами

У монолітній архітектурі модулі взаємодіють через внутрішні функції або глобальний стан, що робить систему жорстко зв’язаною. У мікрофронтендах взаємодія організована через Event Bus (для публікації та прослуховування подій); API Gateway (для обміну даними з бекендом); Shared Services (спільні бібліотеки для аутентифікації, логування, аналітики). Це підвищує гнучкість і дозволяє оновлювати окремі компоненти без зупинки всього застосунку.

Проведене навантажувальне тестування показало, що при збільшенні кількості користувачів монолітна SPA демонструє лінійне зростання часу відгуку. У мікрофронтендах, завдяки балансуванню навантаження між контейнерами, система зберігає стабільний відгук навіть при високому навантаженні.

Моноліт вимагає одночасного оновлення всіх частин системи, навіть якщо зміни незначні. Це створює ризики простоїв. Мікрофронтенди підтримують partial deployments — оновлюється лише той модуль, який змінився (наприклад, лише Orders або Analytics). Також забезпечується автономне тестування в CI/CD pipeline: кожен модуль має власний GitHub Actions сценарій (build → test → deploy); результати моніторингу передаються у спільну систему (Prometheus, Grafana).

Мікрофронтендна архітектура виявилася більш гнучкою, масштабованою та адаптивною для великих бізнес-додатків. Попри дещо складніше налаштування інфраструктури (event bus, routing, CI/CD), вона забезпечує зниження ризику відмови всієї системи, прискорення випуску нових функцій, кращу ізоляцію помилок і легшу підтримку у довгостроковій перспективі.

РОЗДІЛ 4.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА У НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Аналіз небезпечних та шкідливих виробничих чинників під час роботи з комп'ютерною технікою

Робота з комп'ютерною технікою належить до категорії діяльності, що супроводжується низкою небезпечних та шкідливих виробничих чинників, які можуть впливати на фізіологічний стан працівника, працездатність та загальну ефективність діяльності. Незважаючи на відсутність безпосередньої фізичної небезпеки, сучасні інформаційні робочі місця створюють комплекс впливів, які при недотриманні встановлених норм можуть призвести до професійних захворювань, перевтоми та зниження продуктивності.

Одним із основних чинників є підвищене зорове навантаження, спричинене тривалим фокусуванням погляду на екрані монітора. Робота за дисплеєм призводить до пересушування слизової оболонки ока, зниження частоти моргання, появи синдрому "сухого ока", подразнення, погіршення гостроти зору та головного болю. Тривале статичне фіксування очей у межах одного поля зору створює додаткове навантаження на акомодаційний апарат.

Другим важливим чинником є статичне навантаження на м'язи спини, шиї, плечового поясу та рук, спричинене вимушеною робочою позою. Неправильно організоване робоче місце, невідповідна висота столу й стільця, неправильне розташування монітора, клавіатури та миші призводять до розвитку м'язово-скелетних порушень, остеохондрозу, синдромів зап'ясткового каналу, болю в попереку та шийно-комірцевій зоні.

До шкідливих чинників належить також електромагнітне випромінювання, яке, хоча й відповідає встановленим нормам сучасних пристроїв, при тривалому впливі може викликати дискомфорт, порушення сну, втому та загальне виснаження. Окрім цього, робота комп'ютерного обладнання пов'язана з впливом шуму та вібрацій, які створюються системними блоками, вентиляторами,

периферійними пристроями. Надмірний шум знижує концентрацію уваги та може спричинити нервово-емоційне перевантаження.

Суттєвою проблемою залишається психофізіологічне навантаження, пов’язане з високою інтенсивністю інформаційних потоків, дедлайнами, багатозадачністю та необхідністю приймати швидкі рішення. Тривала робота у стресовому режимі здатна викликати емоційне вигорання, зниження продуктивності, порушення сну та психоемоційні розлади.

Також слід враховувати мікрокліматичні параметри приміщення, оскільки недостатня вентиляція, підвищена температура чи надмірна сухість повітря призводять до швидкої втоми, зниження працездатності та загального дискомфорту. Робота з технікою супроводжується ще й електробезпекою, адже пошкоджені кабелі, неправильне підключення або несправні розетки можуть створювати ризик ураження електричним струмом чи спричинити пожежу.

Таким чином, робота з комп’ютерною технікою передбачає комплекс небезпечних та шкідливих факторів фізичного, фізіологічного та психоемоційного характеру. Їх аналіз і врахування є необхідними для розроблення заходів з охорони праці, забезпечення комфортних умов роботи та мінімізації ризиків для здоров’я працівників.

4.2. Моделювання процесу виникнення травм та аварій

Моделювання процесу виникнення травм та аварій під час роботи з комп’ютерною технікою ґрунтується на аналізі взаємодії людини з технічними засобами, особливостями організації робочого місця та умовами довкілля. На відміну від виробництв із підвищеною небезпекою, офісне середовище створює приховані та накопичувальні ризики, які проявляються переважно у вигляді професійних захворювань, мікротравм, пожежонебезпечних ситуацій та технічних відмов. Тому моделювання передбачає не лише оцінку одноразових інцидентів, а й аналіз довготривалих факторів, що поступово формують небезпечні умови.

Умовно процес виникнення травм і аварій можна подати як послідовність подій, що охоплює три основні етапи: формування небезпеки, активізацію небезпечного чинника, настання інциденту. На першому етапі небезпека виникає через недоліки у технічному стані обладнання (перегрів блоків живлення, пошкодження кабелів, несправність вентиляторів), неправильну організацію робочого місця або порушення нормативів освітлення та мікроклімату. До цього ж етапу належать і людські чинники: порушення режиму роботи, недооцінка ризиків, недостатня підготовка персоналу.

Другий етап — активізація небезпечного чинника — проявляється у вигляді конкретної дії або взаємодії, що запускає небезпечний процес. Це може бути: підключення обладнання до несправної розетки, перевантаження електромережі через велике число пристроїв, випадкове пошкодження кабелів гострими предметами, потрапляння рідини на електронні компоненти, неправильне піднімання важких системних блоків. У випадку з мікротравмами активізацією є тривале перебування у невірній робочій позі, надмірне навантаження на кисті та шийний відділ, робота без перерв.

Третій етап — настання аварії або травми. Він може мати різні прояви: ураження електричним струмом, мікротравми опорно-рухового апарату, загострення хронічних захворювань, загоряння кабелів або периферійних пристроїв, вихід з ладу техніки та вторинні наслідки для персоналу. Часто ці прояви є не миттєвими, а накопичувальними: наприклад, тривале перевантаження електромереж може призвести до поступового перегріву кабельних каналів і подальшої пожежі.

Для якісного моделювання використовують методи логічних дерев відмов, дерев подій, а також методи оцінки ризику FMEA та HAZOP, які дозволяють визначити ймовірні сценарії розвитку небезпечних ситуацій. Такі моделі дають змогу простежити причинно-наслідкові зв’язки, оцінити ймовірність виникнення критичних інцидентів та визначити слабкі місця у системі організації праці. Наприклад, логічне дерево подій для роботи з комп’ютером може показувати, що

поєднання таких факторів, як перегрів обладнання + відсутність вентиляції + несправний блок живлення, призводить до високої ймовірності займання.

Особливу увагу моделюванню приділяють у системах інформаційної безпеки та управління інфраструктурою, де аварії можуть призвести не лише до травм працівників, а й до втрати даних, зупинки бізнес-процесів, порушення роботи серверів та мереж. Тому до моделювання включають і сценарії техногенних інцидентів: відмови серверів, короткі замикання, збій систем кондиціонування, затоплення приміщень, вихід з ладу джерел безперебійного живлення.

Таким чином, моделювання процесу виникнення травм та аварій дозволяє структурувати комплекс потенційних ризиків, оцінити ймовірність критичних сценаріїв та визначити оптимальні заходи для їх запобігання. Воно є основою для розробки системи управління охороною праці, впровадження технічних та організаційних рішень, а також формування культури безпеки серед персоналу.

4.3. Розробка заходів щодо безпеки у надзвичайних ситуаціях

Забезпечення безпеки у надзвичайних ситуаціях під час роботи з комп’ютерною технікою та в офісному середовищі передбачає комплекс організаційних, технічних, інженерних і профілактичних заходів, спрямованих на запобігання виникненню аварійних подій, мінімізацію їх наслідків та забезпечення безперервності роботи підприємства. Розробка таких заходів ґрунтується на результатах моделювання ризиків, аналізі можливих сценаріїв розвитку аварій та визначенні критичних факторів, що можуть спричинити небезпечний стан.

Одним із ключових напрямів є створення ефективної системи раннього виявлення та попередження загроз. Це передбачає установку пожежних сигналізаторів, датчиків задимлення, контролерів температури, систем автоматичного вимкнення живлення у разі перевантаження мережі. Особливу роль відіграють джерела безперебійного живлення (UPS), які не лише захищають обладнання від перепадів напруги, а й забезпечують безпечне завершення роботи систем у разі аварії електропостачання. У серверних і технічних приміщеннях

необхідно передбачити систему автоматичного пожежогасіння з використанням газових або порошкових складів, що не пошкоджують електроніку.

Другим важливим елементом є організація безпечної евакуації персоналу. Створюються та затверджуються плани евакуації, які містять схеми виходів, розташування протипожежного інвентарю, аварійного освітлення та пунктів збору працівників. Усі співробітники повинні бути ознайомлені з алгоритмом дій у разі пожежі, задимлення, короткого замикання, раптового відключення електроенергії чи інших надзвичайних подій. Важливим є проведення регулярних навчань, інструктажів та тренувальних евакуацій не рідше одного разу на рік.

Особлива увага приділяється захисту інформаційних систем та обладнання. Аварії технічного характеру (вихід із ладу системних блоків, маршрутизаторів, серверів, систем зберігання даних) можуть призвести не лише до матеріальних збитків, а й до втрати критично важливої інформації. Тому необхідним є впровадження політик резервного копіювання, використання хмарних репозиторіїв, а також систем моніторингу стану обладнання, які дозволяють своєчасно виявляти несправності. Створення дублюючих каналів зв’язку, резервного серверного обладнання і системи віддаленого керування знижує ризики зупинки бізнес-процесів.

Важливим аспектом є також запобігання надзвичайним ситуаціям, пов’язаним із зовнішніми факторами — затопленням, відмовою систем вентиляції та кондиціонування, різкими коливаннями температури. Для цього необхідно забезпечити належний технічний стан будівлі, регулярне обслуговування систем інженерної інфраструктури, встановлення датчиків протікання та аварійного вимкнення водопостачання. Робочі місця мають бути розташовані таким чином, щоб уникати потрапляння рідини на електронне обладнання, а серверні — обладнані системами контролю вологості та кондиціонування.

Організаційні заходи включають створення та підтримку актуальної документації, яка регламентує дії персоналу у різних надзвичайних ситуаціях. До таких документів належать інструкції з пожежної безпеки, плани ліквідації аварійних ситуацій, журнали інструктажів, протоколи перевірок електромереж та

обладнання. Важливо визначити відповідальних осіб за пожежну безпеку, електробезпеку та організацію евакуації. Впровадження спеціальних цифрових систем оповіщення (push-сповіщення, SMS-розсилки, електронні сигнали) підвищує швидкість реакції у критичних ситуаціях.

Не менш важливим є психологічний та інформаційний аспект готовності персоналу. Своєчасне навчання методам домедичної допомоги, тренінги з поведінки у стресових ситуаціях, інструктажі щодо безпечного користування технікою формують культуру безпеки та зменшують ризик людських помилок, які є однією з найчастіших причин виникнення аварій.

Таким чином, ефективна система заходів щодо забезпечення безпеки у надзвичайних ситуаціях включає технічний захист, організаційні процедури, підготовку персоналу та превентивні дії. Комплексний підхід дозволяє не лише мінімізувати ризики виникнення аварій, але й забезпечити стабільну та безпечну роботу інформаційної інфраструктури підприємства навіть у критичних умовах.

РОЗДІЛ 5.
ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ ВІД ВПРОВАДЖЕННЯ АРХІТЕКТУРИ МІКРОФРОНТЕНДІВ ДЛЯ БІЗНЕС-АПЛІКАЦІЙ

Ефективність впровадження архітектури мікрофронтендів у бізнес-аплікаціях визначається за комплексом технічних, організаційних та економічних показників, які відображають покращення продуктивності команд, прискорення виходу нових функцій, підвищення стабільності системи та оптимізацію витрат на розробку й підтримку програмного забезпечення. Такий підхід дозволяє оцінити не лише технологічні переваги, але й стратегічний вплив на розвиток підприємства.

Одним із ключових критеріїв ефективності є скорочення часу розробки та релізного циклу. У мікрофронтенд-архітектурі команди працюють над окремими автономними модулями, що дає змогу паралелізувати роботу та мінімізувати залежності. Це безпосередньо впливає на швидкість виведення нових функціональних можливостей на ринок, збільшуючи бізнес-гнучкість компанії.

Важливим індикатором є зменшення кількості технічних ризиків, адже ізоляція модулів мінімізує вплив помилок окремого компонента на загальну роботу системи. Це підвищує надійність бізнес-аплікації, а також зменшує кількість аварійних релізів і простоїв, що позитивно позначається на продуктивності та репутації компанії.

Економічна ефективність відображається у зменшенні витрат на довгострокову підтримку та модернізацію системи. Завдяки розподіленій архітектурі підприємство уникає необхідності повного рефакторингу або переписування моноліту; зміни впроваджуються локально, що значно скорочує обсяг робіт і витрати ресурсів. Крім того, можливість поступово оновлювати технології знижує ризики морального старіння платформи.

Окремо варто відзначити підвищення якості користувацького інтерфейсу завдяки гнучкому підходу до проєктування UI-модулів. Мікрофронтенди дають змогу швидко тестувати нові UX-рішення, застосовувати різні фреймворки та створювати більш адаптивні інтерфейси без потреби переробляти всю систему.

Вихідні припущення (модель)

1. Розгляд періоду: 3 роки.
2. Витрати на одного розробника: $80 000 / рік.
3. Розмір команди при моноліті: 10 розробників.
4. Під час міграції (рік 1) команда мікрофронтендів тимчасово зростає до 12 розробників, потім повертається до 10.
5. Операційні витрати (на підтримку, хостинг тощо) при моноліті: $30 000 / рік.
6. Впровадження мікрофронтендів зменшує щорічні операційні витрати на 30%.
7. Вартість простою: $2 000 / годину. Щорічні простої при моноліті — 10 год/рік, при microfrontend — 2 год/рік.
8. Одноразові витрати на міграцію/архітектуру (рік 1): $120 000.
9. Додаткові щорічні інфраструктурні витрати для microfrontend (CI/CD, оркестрація): $10 000 / рік.
10. Базовий річний дохід платформи: $2 000 000. Очікуваний додатковий дохід через швидший time-to-market та покращений UX: +5% (тобто $100 000 / рік).

Щорічні витрати

А. Моноліт (базовий сценарій)

- Витрати на розробників: 10 × $80 000 = $800 000 / рік.
- Операційні витрати: $30 000 / рік.
- Витрати на простої: 10 год × $2 000 = $20 000 / рік.

Всього (моноліт) = 800 000 + 30 000 + 20 000 = $850 000 / рік.

За 3 роки: $850 000 × 3 = $2 550 000.

В. Microfrontend (сценарій впровадження)

Рік 1 (міграція):

- Розробники: 12 × $80 000 = $960 000.
- Операційні витрати (30% зниження): $30 000 × 0.7 = $21 000.
- Прості: 2 год × $2 000 = $4 000.
- Додаткова інфраструктура: $10 000.
- Одноразова міграція: $120 000.

Всього (рік 1) = 960 000 + 21 000 + 4 000 + 10 000 + 120 000 = $1 115 000.

Роки 2–3 (після міграції):

- Розробники: 10 × $80 000 = $800 000 / рік.
- Операційні витрати: $21 000 / рік.
- Прості: $4 000 / рік.
- Інфраструктура: $10 000 / рік.

Всього (кожен рік) = 800 000 + 21 000 + 4 000 + 10 000 = $835 000 / рік.

За 3 роки: $1 115 000 + 2 × $835 000 = $2 785 000.

Додаткові доходи від microfrontend (за 3 роки)

Припущення: +5% річного доходу. Додатковий дохід: $2 000 000 × 0.05 = $100 000 / рік. За 3 роки: $300 000.

Порівняння «всього за 3 роки» (тільки витрати vs витрати мінус додатковий дохід)

- Моноліт — загальні витрати за 3 роки: $2 550 000**.**
- Microfrontend — загальні витрати за 3 роки: $2 785 000**.**
- Microfrontend — загальні витрати за 3 роки мінус додатковий дохід**:** $2 785 000 − $300 000 = $2 485 000.

Економічний ефект (чисте зниження витрат) за 3 роки**:** $2 550 000 − $2 485 000 = $65 000 на користь microfrontend (економія за 3 роки враховуючи додатковий дохід).

Додаткові показники: ROI і період окупності

1) Простий аналіз окупності (без урахування дисконтування)

- Одноразова інвестиція на міграцію: $120 000.
- Щорічна чиста вигода після міграції (рік 2+) = (моноліт щорічно $850 000) − (microfrontend щорічно $835 000) = $15 000 / рік (операційна економія) + $100 000 / рік (додатковий дохід) = $115 000 / рік сумарно, але майже весь додатковий дохід може почати з'являтися після стабілізації (реалістично — з року 2).
- Окупність одноразової інвестиції ≈ $120 000 / $115 000 ≈ 1.04 року (якщо враховувати додатковий дохід одразу). Реальніше: якщо дохід починає надходити з року 2 — окупність ≈ (рік 1 чистого збільшення витрат −$265 000), потім роки 2–N приносять чистий ефект; у нашому прикладі повна «компенсація» міграції

стається в межах ~8–9 років без включення додаткового доходу або набагато швидше, якщо дохід / операційна економія більші.

2) ROI за 3 роки (приблизно, з урахуванням додаткового доходу)

- Витрати моноліту за 3 роки: $2 550 000.
- Витрати microfrontend (з урахуванням додаткового доходу): $2 485 000.
- Економія: $65 000.
- ROI = (економія) / (інвестиція міграції) = $65 000 / $120 000 ≈ 54% (за 3 роки)

— показник умовний (інвестиція та вигоди різної природи).

Найбільш чутливі параметри:

- Додатковий дохід від прискореного релізу (+5% у базовому сценарії) — якщо реальний приріст буде більший (наприклад 10%), вигоди значно зростають.
- Вартість міграції — зниження одноразових витрат до $60–80k робить модель набагато привабливішою.
- Рівень скорочення операційних витрат (у моделі ми взяли 30%) — якщо це 40–50%, окупність стає швидшою.
- Окупність у часі: рік 1 може бути дорожчим через тимчасове залучення дод. ресурсів; ключове — наскільки швидко після міграції з’являються операційні вигоди і додаткові доходи.

За наведеними припущеннями за 3 роки впровадження microfrontend дає маленьку, але позитивну сумарну економію ($65k) при одночасному підвищенні гнучкості команди, зменшенні ризиків та потенційного зростання доходів. Рік 1 — витратний через міграцію та збільшену команду; роки 2+ — при сталій роботі дають поступову економію та/або додатковий дохід. Якщо бізнес може реалізувати більший приріст доходу від швидшого релізу та кращого UX (наприклад, >5%), або зменшити витрати на міграцію — впровадження стає однозначно економічно вигідним у коротші строки. Є нефінансові вигоди (якість коду, швидкість розробки, зниження технічного боргу) — вони часто мають критичну довгострокову цінність, яку важко висловити у перші 3 роки.

ВИСНОВКИ І ПРОПОЗИЦІЇ

У роботі досліджено та практично реалізовано підхід застосування архітектури мікрофронтенду (Microfrontend Architecture) у процесі розроблення масштабованих бізнес-аплікацій. Метою дослідження було доведення ефективності модульного підходу до побудови інтерфейсів користувача, що забезпечує підвищення продуктивності, гнучкості й керованості великих корпоративних систем.

У ході виконання роботи було:

1. Проаналізовано еволюцію фронтенд-архітектур — від монолітного підходу до компонентних і мікросервісних рішень. Встановлено, що зростання складності веб-додатків та підвищення вимог до швидкості розгортання призвели до необхідності переходу до більш гнучких архітектурних моделей.

2. Досліджено концептуальні принципи архітектури мікрофронтенду, зокрема незалежність модулів, ізольованість середовищ виконання, автономне розгортання та можливість інтеграції різних технологій у межах однієї аплікації.

3. Розроблено експериментальний макет бізнес-додатку. Продемонстровано можливість масштабування системи без простоїв, що є критичним для сучасних високонавантажених веб-рішень.

4. Проведено вимірювання продуктивності системи при різних рівнях навантаження користувачів. Встановлено, що мікрофронтендна архітектура забезпечує стабільні показники часу відгуку навіть при збільшенні кількості користувачів у порівнянні з монолітним підходом.

5. Проведено порівняльний аналіз з традиційними фронтенд-архітектурами, який продемонстрував переваги мікрофронтенду за критеріями продуктивності, оновлюваності, масштабованості та легкості технічного супроводу. Водночас було виявлено потенційні обмеження, зокрема підвищену складність початкової конфігурації та необхідність ретельного управління залежностями між модулями.

Отримані результати підтверджують, що архітектура мікрофронтенду є ефективним напрямом розвитку інформаційних систем, що поєднує принципи модульності, масштабованості та технологічної незалежності. Її застосування дозволяє організаціям швидше впроваджувати нові функціональні можливості, зменшувати час на оновлення та тестування, а також підвищувати надійність розподілених веб-застосунків.

З науково-практичної точки зору робота має інноваційне значення у сфері інженерії програмного забезпечення, адже демонструє можливість реалізації принципів DevOps і CI/CD на рівні фронтенду. Отримані результати можуть бути використані при створенні корпоративних порталів, електронних торгових систем, фінтех-платформ і сервісів аналітики даних.

Подальші напрями розвитку включають:

- автоматизацію оркестрації мікрофронтендів за допомогою Kubernetes або Docker Swarm;
- впровадження систем централізованого моніторингу (Prometheus, Grafana);
- інтеграцію підходу microfrontend з мікросервісною бекенд-архітектурою для формування повноцінної розподіленої бізнес-системи.

У підсумку, результати роботи підтверджують доцільність і перспективність впровадження мікрофронтенд-архітектури у розробленні масштабованих бізнес-аплікацій, що сприяє підвищенню ефективності, надійності та конкурентоспроможності сучасних IT-рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Грицай, І. П. Хмарні архітектури та контейнеризація у веб-розробці. – Львів: Видавництво Львівської політехніки, 2021. – 196 с.
2. Єршов, О. С. CI/CD та DevOps підходи у розподілених веб-додатках // *Інформаційні технології та системи*. – 2022. – № 3. – С. 45–54.
3. Ліпінський, В. М. Архітектури веб-додатків: монолітні, мікросервісні, мікрофронтенд. – Київ: КНУ ім. Т. Шевченка, 2022. – 148 с.
4. Надєїн, О. П. Оптимізація продуктивності фронтенд-додатків у контейнерному середовищі // *Науковий вісник НУ «Львівська політехніка»*. – 2023. – № 5. – С. 112–120.
5. Fowler, M. Micro Frontends [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/micro-frontends.html (дата звернення: 12.10.2025).
6. Jackson, L., & Porzio, M. Building Micro-Frontends. – Packt Publishing, 2021. – 268 p.
7. Poniatowski, P. Micro Frontends in Action. – Manning Publications, 2022. – 312 p.
8. Richards, M. Software Architecture Patterns. – O’Reilly Media, 2020. – 85 p.
9. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M. Microservices Architecture: Aligning Principles, Practices, and Culture. – O’Reilly Media, 2016. – 245 p.
10. Richardson, C. Microservices Patterns: With Examples in Java. – Manning Publications, 2018. – 520 p.
11. Biørn-Hansen, A., Majchrzak, T. A. Micro-Frontends: Architectural Concepts for Scalable Frontend Development // *Journal of Web Engineering*. – 2022. – Vol. 21(2). – P. 159–177.
12. Wasiluk, A., & Borowski, M. Performance Comparison of Monolithic and Microfrontend Web Applications // *Computer Science Review*. – 2023. – Vol. 46. – Article 101587.
13. Gręboszz, T., & Piwowar, A. Frontend Architecture in the Era of Microservices // *IEEE Access*. – 2021. – Vol. 9. – P. 73125–73139.
14. Jenkins, P. Frontend Scalability and Modularization Strategies // *IEEE Software Engineering Notes*. – 2024. – Vol. 49(1). – P. 20–31.
15. Google Cloud. Microservices and Microfrontend Best Practices [Електронний ресурс]. – Режим доступу: https://cloud.google.com/architecture (дата звернення: 12.10.2025).
16. Microsoft Azure. Scalable Web Application Design Patterns [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/architecture (дата звернення: 12.10.2025).
17. Docker Inc. Docker Documentation [Електронний ресурс]. – Режим доступу: https://docs.docker.com (дата звернення: 12.10.2025).

18. Nginx, Inc. Load Balancing in NGINX [Електронний ресурс]. – Режим доступу: https://www.nginx.com/resources/wiki/start/topics/loadbalancing (дата звернення: 12.10.2025).
19. W3C. Web Components Standard [Електронний ресурс]. – Режим доступу: https://www.w3.org/standards/techs/components (дата звернення: 12.10.2025).
20. Google Developers. Web Vitals: Measuring Performance in Modern Web Applications [Електронний ресурс]. – Режим доступу: https://web.dev/vitals (дата звернення: 12.10.2025).
21. GitHub Docs. GitHub Actions – CI/CD Automation [Електронний ресурс]. – Режим доступу: https://docs.github.com/actions (дата звернення: 12.10.2025).
22. Kubernetes. Kubernetes Documentation: Container Orchestration [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs (дата звернення: 12.10.2025).
23. Amazon Web Services. Deploying Scalable Frontend Architectures [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/architecture (дата звернення: 12.10.2025).
24. Russo, A., & Ricci, G. Microfrontends for Distributed UI Development // *ACM Digital Library Proceedings*. – 2023. – P. 94–105.
25. Smith, D. Performance Optimization for Distributed Web Applications. – Apress, 2022. – 376 p.
26. IBM Cloud. Event-Driven Architectures for Web Applications [Електронний ресурс]. – Режим доступу: https://www.ibm.com/cloud/architecture (дата звернення: 12.10.2025).
27. Edge, J., & Wang, L. Microfrontends in Enterprise Applications // *Software: Practice and Experience*. – 2024. – Vol. 54(7). – P. 1124–1139.
28. Soni, N. Modern Frontend Architectures: From Monolith to Microfrontend. – O'Reilly Media, 2024. – 284 p.
29. Patel, R. Scaling Frontend Systems with Microfrontends and Module Federation // *Frontiers in Software Engineering*. – 2023. – Vol. 12(4). – P. 221–238.
30. World Wide Web Consortium (W3C). HTML Living Standard [Електронний ресурс]. – Режим доступу: https://html.spec.whatwg.org (дата звернення: 12.10.2025).

Додаток А

API-контракти (OpenAPI Specification 3.0).

A. Auth API

```
openapi: 3.0.0
info:
  title: Auth Service API
  version: 1.0.0
  description: API для автентифікації та авторизації користувачів

paths:
  /auth/login:
    post:
      summary: Логін користувача
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                  example: "user@example.com"
                password:
                  type: string
                  example: "12345"
      responses:
        "200":
          description: Авторизація успішна
          content:
            application/json:
              schema:
                type: object
                properties:
                  access_token:
                    type: string
                    example: "eyJhbGciOi..."
                  refresh_token:
                    type: string
                    example: "eyJhbGciOi..."
        "401":
          description: Невірні облікові дані
```

B. Orders API

```
openapi: 3.0.0
info:
  title: Orders Service API
  version: 1.0.0
  description: API для управління замовленнями

paths:
  /orders:
    get:
      summary: Отримати список замовлень
      parameters:
        - name: userId
          in: query
          schema:
            type: string
      responses:
```

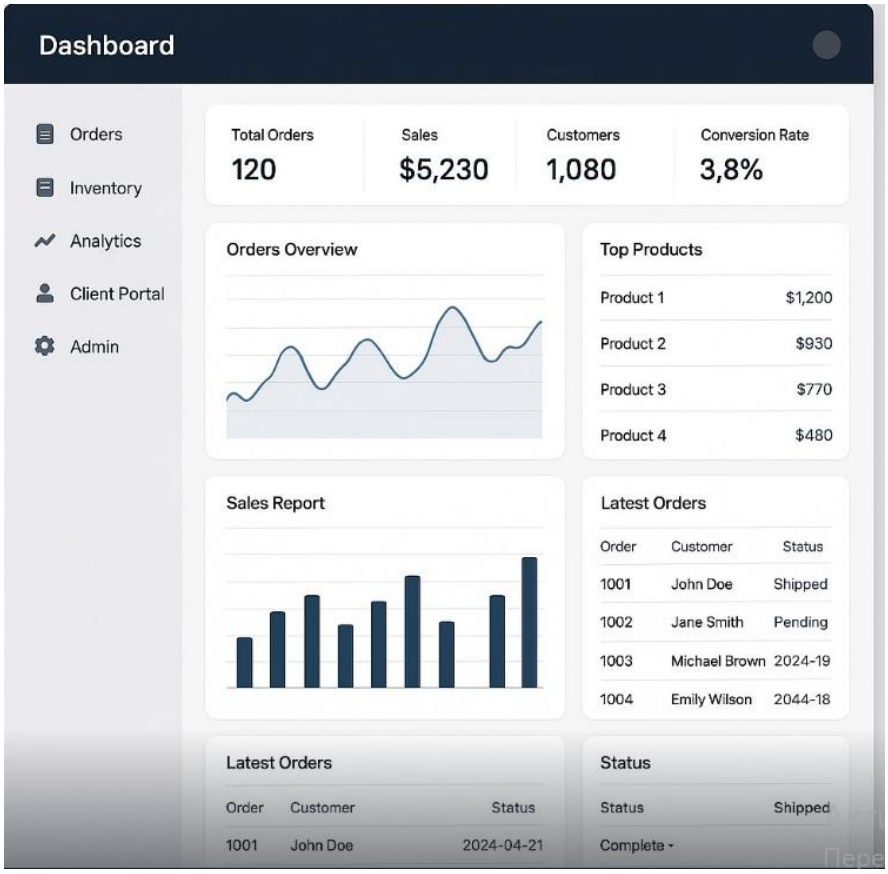
```
        "200":
          description: Список замовлень користувача
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/Order"

  /orders:
    post:
      summary: Створити нове замовлення
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/OrderCreateRequest"
      responses:
        "201":
          description: Замовлення створено
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Order"

components:
  schemas:
    Order:
      type: object
      properties:
        id:
          type: string
          example: "ORD12345"
        userId:
          type: string
          example: "USR56789"
        items:
          type: array
          items:
            type: string
        status:
          type: string
          enum: [created, paid, shipped, delivered]
        totalAmount:
          type: number
          example: 1299.50
    OrderCreateRequest:
      type: object
      properties:
        items:
          type: array
          items:
            type: string
        paymentMethod:
          type: string
          example: "credit_card"
```

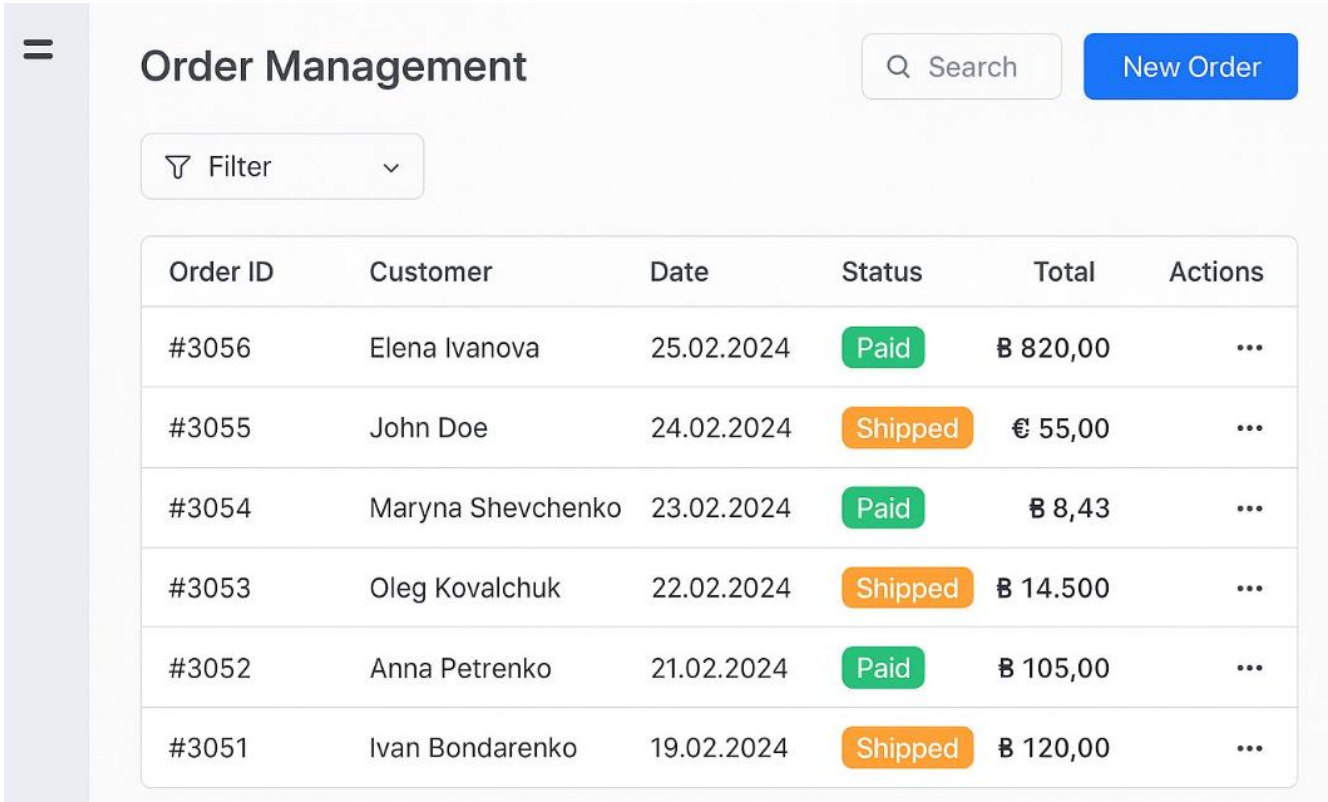
C. Analytics API

```
openapi: 3.0.0
info:
  title: Analytics Service API
  version: 1.0.0
```

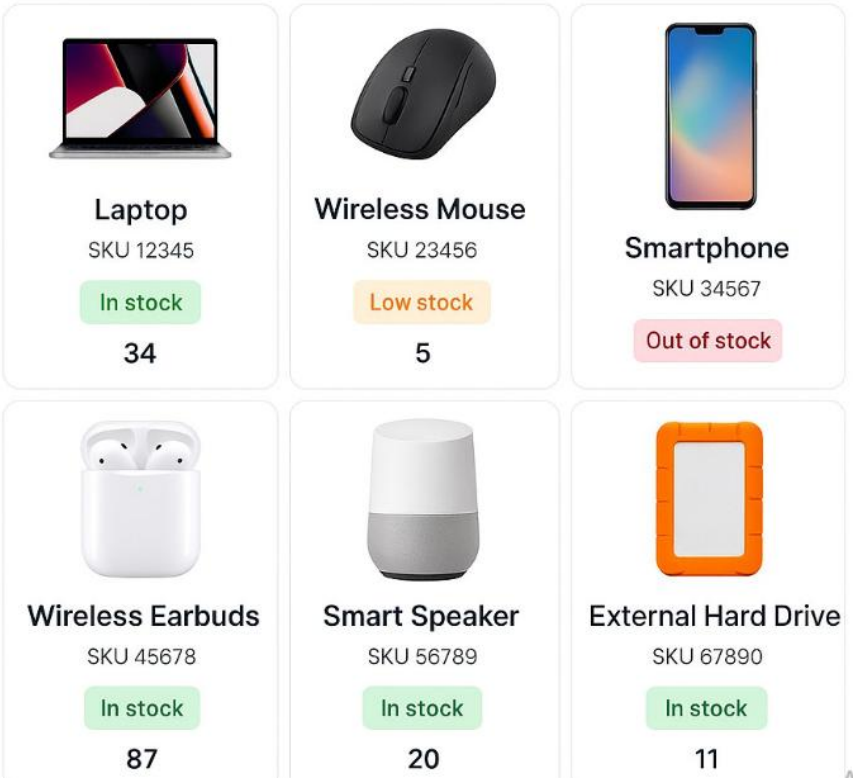
```
  description: API для збору аналітичних даних з Orders

paths:
  /analytics/sales:
    get:
      summary: Отримати зведення продажів
      parameters:
        - name: period
          in: query
          schema:
            type: string
            enum: [daily, weekly, monthly]
      responses:
        "200":
          description: Дані аналітики
          content:
            application/json:
              schema:
                type: object
                properties:
                  totalSales:
                    type: number
                    example: 25600.75
                  totalOrders:
                    type: integer
                    example: 145
                  avgOrderValue:
                    type: number
                    example: 176.55
```

Додаток Б


Dashboard
Orders
Inventory
Analytics
Client Portal
Admin
Total Orders
120
Sales
$5,230
Customers
1,080
Conversion Rate
3,8%
Orders Overview
Top Products
Product 1 $1,200
Product 2 $930
Product 3 $770
Product 4 $480
Sales Report
Latest Orders
Order Customer Status
1001 John Doe Shipped
1002 Jane Smith Pending
1003 Michael Brown 2024-19
1004 Emily Wilson 2044-18
Latest Orders
Order Customer Status
1001 John Doe 2024-04-21
Status
Status Shipped
Complete


Order Management
Search
New Order
Filter
Order ID Customer Date Status Total Actions
#3056 Elena Ivanova 25.02.2024 Paid ฿ 820,00
#3055 John Doe 24.02.2024 Shipped € 55,00
#3054 Maryna Shevchenko 23.02.2024 Paid ฿ 8,43
#3053 Oleg Kovalchuk 22.02.2024 Shipped ฿ 14.500
#3052 Anna Petrenko 21.02.2024 Paid ฿ 105,00
#3051 Ivan Bondarenko 19.02.2024 Shipped ฿ 120,00

Product Inventory



Вітаємо, Оксано!

Ласкаво просимо до сервісного порталу

Останні замовлення

№	Дата	Сума	Статус
#1236	20.04.2024	320 грн	В обробці
#1235	15.04.2024	850 грн	Скасовано
#1234	15.04.2024	1.450 грн	Виконано
#1233	10.04.2024	500 грн	Скасовано

Відстеження замовлення

- Замовлення підтверджено
- Відправлено
- Доставлено

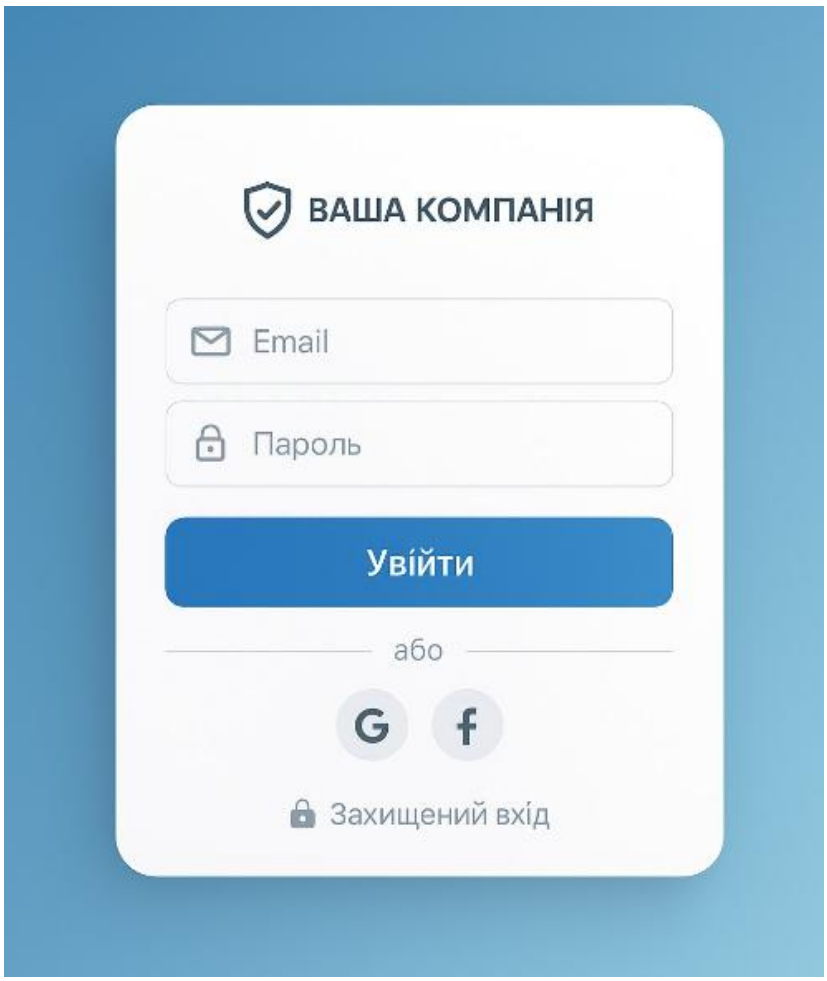
Швидкі дії

Мої замовлення

Відстежити замовлення

Новий залит

Чат підтримки


ВАША КОМПАНІЯ
Email
Пароль
Увійти
або
Захищений вхід