БАКАЛАВРСЬКА РОБОТА

РБ. ІП - 06.00.00.000 ПЗ

Група ІП-22-4

Душенко Юля

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Душенко Юля Євгенівна

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Побудова ВЕБ-сервісів на основі архітектурних шаблонів

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____ Душенко Ю.Є.
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Царева Олександра Степанівна, асистент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

***ЗАТВЕРДЖУЮ*:**
Зав. кафедрою ***ІПЗ***
доц. В.В. Бандура
“ ” 2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Душенко Юлі Євгенівні

(прізвище, ім’я, по-батькові)

1. ***Тема проекту*** **(*роботи*)** **“ Побудова ВЕБ-сервісів на основі архітектурних шаблонів”** керівник проекту (роботи) Царева Олександра Степанівна, асистент

затверджені наказом закладу вищої освіти від **“** 21 ” квітня 2026 р. № 179/7

2. ***Строк подання студентом проекту*** **(*роботи*)** 10 червня 2026 р.

3. ***Вихідні дані до проекту*** **(*роботи*)** Результати і матеріали отримані під час проходження переддипломної практики

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

1. Огляд існуючих рішень та теоретичних основ створення веб-сервісу
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. ***Перелік графічного матеріалу*** **(*з точним зазначенням обов’язкових креслень*)**

1. Прототип сайту RoomBooking Hub у no-code середовищі Adalo (рис. 3.1, ст. 35)
2. Use Case діаграма системи Silver Bloom (рис. 3.2, ст. 38)
3. UML Sequence Diagram: потік бронювання номеру (рис. 3.4, ст. 46)
4. Головне вікно програми — результати виконання тестового набору (рис. 5.1, ст. 58)
5. ER-діаграма бази даних PostgreSQL проєкту Silver Bloom (рис. В.1, ст. 83)

Консультанти розділів проекту (роботи)

Розділ	***Консультант***	***Підпис, дата***	
		Завдання видав	***Завдання прийняв***

1. ***Дата видачі завдання*** ***січня 2026 року***

Керівник _____ Царева О.С.
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ Душенко Ю.Є.
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	***Назва етапів дипломного проекту (роботи)***	***Строк виконання етапів проекту***	***Примітка***
1	Визначення та обґрунтування теми дипломної роботи	20.01.2026	Виконано
2	Аналіз предметної області електронної комерції та існуючих рішень	01.02.2026	Виконано
3	Проєктування архітектури веб-застосунку, структури бази даних та інтерфейсу	20.03.2026	Виконано
4	Реалізація функціоналу веб-застосунку та адміністративної панелі	15.04.2026	Виконано
5	Розробка аналітичних модулів та тестування системи	18.05.2026	Виконано
6	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	Виконано

Студент – дипломник _____ Душенко Ю.Є.
(підпис) (розшифровка підпису)

Керівник роботи _____ Царева О.С.
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Бакалаврська робота на 76 сторінки містить 32 рисунки, 35 джерел та 6 додатків.

Метою роботи є розробка веб-сервісу для автоматизації готельних бронювань, що забезпечує взаємодію користувачів, облік номерного фонду та керування бронюваннями.

Об'єкт дослідження: процеси розробки та функціонування веб-сервісів у сфері готельного бізнесу.

Предмет дослідження: методи розробки веб-сервісу з використанням Clean Architecture, архітектури, БД, інтерфейсу та захисту від дублювання.

Результати дослідження: розроблено веб-сервіс на React 19 та TypeScript із застосуванням Supabase (PostgreSQL), що забезпечує каталог, бронювання та адміністрування.

У першому розділі проаналізовано існуючі рішення та теоретичні основи, включаючи принципи SOLID та Clean Architecture.

У другому розділі визначено вимоги до системи та обґрунтовано вибір архітектурного шаблону.

У третьому розділі спроєктовано архітектуру системи, базу даних та інтерфейс.

У четвертому розділі реалізовано програмний застосунок, включаючи бізнес-логіку та адаптери.

У п'ятому розділі проведено тестування прототипу та оцінку зручності використання.

Висновок: розроблений сервіс відповідає поставленій меті, забезпечує повний функціонал та підтверджує ефективність Clean Architecture.

КЛЮЧОВІ СЛОВА: ВЕБ-СЕРВІС, БРОНЮВАННЯ, CLEAN ARCHITECTURE, REACT, TYPESCRIPT, SUPABASE, POSTGRESQL, ПРОГРАМА ЛОЯЛЬНОСТІ, ТЕСТУВАННЯ, SOLID.

ABSTRACT

The bachelor's thesis, consisting of 76 pages, contains 32 figures, 35 references, and 6 appendices.

The aim of the work is to develop a web service for automating hotel reservations, ensuring user interaction, room stock accounting, and reservation management.

Object of research: processes of development and operation of web services in the hospitality industry.

Subject of research: methods of web service development using Clean Architecture, system architecture, databases, UI, and protection against double booking.

Results of the research: a web service based on React 19 and TypeScript using Supabase (PostgreSQL) was developed, providing catalog functionality, booking, and administration.

The first chapter analyzes existing solutions and theoretical foundations, including SOLID principles and Clean Architecture.

The second chapter defines system requirements and justifies the choice of the architectural pattern.

The third chapter outlines the system architecture, database design, and user interface.

The fourth chapter details the implementation of the application, including business logic and infrastructure adapters.

The fifth chapter covers prototype testing and usability evaluation.

Conclusion: the developed service meets the stated objective, provides full reservation system functionality, and confirms the effectiveness of the Clean Architecture pattern.

KEYWORDS: WEB SERVICE, BOOKING, CLEAN ARCHITECTURE, REACT, TYPESCRIPT, SUPABASE, POSTGRESQL, LOYALTY PROGRAM, TESTING, SOLID.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface, інтерфейс програмування застосунків.

BaaS – Backend as a Service, бекенд як послуга.

BCE – Boundary-Control-Entity, архітектурний шаблон розмежування компонентів.

CDN – Content Delivery Network, мережа доставки контенту.

CI/CD – Continuous Integration / Continuous Delivery, безперервна інтеграція та доставка.

CRUD – Create, Read, Update, Delete, створити, прочитати, оновити, вилучити.

DIP – Dependency Inversion Principle, принцип інверсії залежностей.

ER – Entity-Relationship, сутність-зв'язок (тип діаграми).

GDPR – General Data Protection Regulation, загальний регламент захисту даних.

IEEE – Institute of Electrical and Electronics Engineers, інститут інженерів електротехніки та електроніки.

ISP – Interface Segregation Principle, принцип розділення інтерфейсів.

ISO – International Organization for Standardization, міжнародна організація зі стандартизації.

JWT – JSON Web Token, токен автентифікації.

LSP – Liskov Substitution Principle, принцип підстановки Лісков.

MVP – Minimum Viable Product, мінімально життєздатний продукт.

OCP – Open/Closed Principle, принцип відкритості/закритості.

ORM – Object-Relational Mapping, об'єктно-реляційне відображення.

OWASP – Open Web Application Security Project, відкритий проєкт безпеки вебзастосунків.

RBAC – Role-Based Access Control, керування доступом на основі ролей.

RLS – Row Level Security, безпека на рівні рядків бази даних.

REST – Representational State Transfer, архітектурний стиль взаємодії компонентів.

SDK – Software Development Kit, набір інструментів розробника.

SOLID – абревіатура п'яти принципів об'єктно-орієнтованого проєктування.

SPA – Single Page Application, односторінковий застосунок.

SRP – Single Responsibility Principle, принцип єдиної відповідальності.

TDD/BDD – Test-Driven Development / Behavior-Driven Development, розробка через тестування / розробка через поведінку.

TLS – Transport Layer Security, протокол захисту транспортного рівня.

UAT – User Acceptance Testing, приймальне тестування користувачем.

UI – User Interface, інтерфейс користувача.

UML – Unified Modeling Language, уніфікована мова моделювання.

URL – Uniform Resource Locator, уніфікований локатор ресурсів.

UUID – Universally Unique Identifier, універсальний унікальний ідентифікатор.

UX – User Experience, досвід взаємодії користувача.

WCAG – Web Content Accessibility Guidelines, настанови з доступності веб-контенту.

YAML – YAML Ain't Markup Language, формат серіалізації даних.

ВСТУП

Сьогодні інформаційні технології розвиваються з неймовірною швидкістю, тому розробка та модернізація програмного забезпечення займає одне з ключових місць у світі бізнесу. Сучасні організації, завдяки своїм веб-сервісам, надають миттєвий доступ до різноманітних послуг, спрощують процес взаємодії з користувачами і дозволяють автоматизувати основні бізнес-процеси. Зростаюча популярність цифрових платформ, розвиток хмарних рішень і зміна архітектурних вимог підвищують вимоги до якості, функціональності та зручності використання веб-застосунків у будь-якій індустрії.

Особливо важливою стає інтеграція ефективних архітектурних рішень, що гарантують стабільність та гнучкість веб-сервісів. Правильний вибір способу організації коду, розподіл відповідальності між компонентами та мінімізація технічного боргу допомагають підвищити продуктивність систем, оптимізувати процеси розробки і покращити користувацький досвід. У сучасних умовах конкуренції на ринку програмного забезпечення наявність чітко структурованих інструментів стає важливим фактором успіху будь-якого програмного продукту.

Актуальність теми обумовлена потребою в створенні сучасних веб-сервісів, які поєднують у собі масштабований інтерфейс, стійку бізнес-логіку та гнучкі механізми збереження даних. Хоча існує чимало готових систем, багато з них побудовані на застарілих підходах, мають високу зв'язність компонентів або недостатню стійкість до зміни технологічного стеку, що підкреслює потребу в нових підходах до побудови веб-сервісів на основі ефективних архітектурних шаблонів.

Метою роботи є побудова веб-сервісу на основі сучасних архітектурних шаблонів, який забезпечить чітке розділення логічних шарів системи, а також можливість її ефективного супроводу, тестування та масштабування.

Для досягнення мети потрібно вирішити такі завдання: провести аналіз сучасного стану проблематики та еволюції архітектурних підходів в інженерії ПЗ; визначити функціональні та нефункціональні вимоги до проектованої системи;

спроектувати внутрішню архітектуру веб-сервісу та структуру сховища даних; реалізувати програмний комплекс із використанням сучасних технологій розробки; розробити інтерфейсні та адміністративні модулі системи; реалізувати механізми взаємодії між шарами на основі обраного шаблону; провести тестування та оцінку зручності використання розробленого сервісу.

Об’єктом дослідження є процеси проектування, розробки та функціонування веб-сервісів, які охоплюють аналіз вимог, вибір архітектурних рішень, імплементацію та тестування програмного забезпечення.

Предметом дослідження є методи, технології та інструменти побудови веб-сервісів на основі архітектурних шаблонів, зокрема принципи SOLID, проектування структури бази даних, реалізація доменного ядра, адаптерів інтерфейсу та засобів автоматизації тестування.

У процесі дослідження використовувалися методи аналізу та узагальнення для вивчення предметної області та існуючих рішень, методи моделювання для проєктування архітектури системи, а також методи програмування для реалізації функціоналу веб-сервісу.

У процесі розробки застосунку особливу увагу приділено використанню сучасних підходів до створення веб-систем, зокрема концепції Clean Architecture, що передбачає чітке розділення коду на незалежні шари. Застосування цього архітектурного шаблону та принципів SOLID дозволяє забезпечити ізоляцію бізнес-правил від зовнішніх технічних деталей, фреймворків та інтерфейсів користувача, що підвищує масштабованість і зручність супроводу системи. Використання бібліотеки React та хмарної платформи Supabase (на базі СУБД PostgreSQL) сприяє ефективній реалізації функціоналу, а застосування мови TypeScript дозволяє забезпечити високу надійність програмного коду на основі суворої типізації.

Практичне значення отриманих результатів полягає в створенні повнофункціонального веб-сервісу (на прикладі готельної системи «Silver Bloom»), який наочно демонструє переваги побудови програмних систем на основі шаблону Clean Architecture для захисту бізнес-логіки від інфраструктурних змін та збоїв. У

роботі послідовно розглядаються аналіз теоретичних основ, проєктування системи та реалізація застосунку з подальшим тестуванням.

Набула подальшого розвитку методика побудови веб-сервісів у сфері готельного бізнесу на основі архітектурного шаблону Clean Architecture, яка, на відміну від існуючих підходів із використанням монолітної або багатошарової архітектури, забезпечує повну ізоляцію бізнес-логіки від інфраструктурних деталей через суворе дотримання принципу інверсії залежностей (DIP) та реалізацію патерну «Репозиторій», що підтверджено практичним впровадженням механізму транзакційного захисту від подвійного бронювання на рівні бази даних PostgreSQL.

Бакалаврська робота містить 76 сторінок, 32 рисунки, 5 розділів, список використаних джерел із 35 найменувань, 6 додатки.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ТЕОРЕТИЧНИХ ОСНОВ СТВОРЕННЯ ВЕБ-СЕРВІСУ

1.1. Аналіз сучасного стану проблематики та еволюція архітектурних підходів

На сучасному етапі розвитку інформаційних технологій веб-сервіси трансформувалися з простих інформаційних ресурсів у складні програмні системи, що забезпечують виконання критично важливих бізнес-процесів [1]. Зі зростанням функціональності програмного забезпечення (ПЗ) постає проблема ефективного управління складністю його внутрішньої структури. Ця проблема має не лише технічний, але й економічний характер: неякісно спроєктовані системи стають складними для модифікації та підтримки, що призводить до значних витрат на рефакторинг або повне переписування [2].

Аналіз сучасних досліджень та практики розробки програмних систем дозволяє виділити такі ключові проблеми:

– **Зростання технічного боргу.** Прагнення скоротити час виходу продукту на ринок (Time-to-Market) часто призводить до прийняття компромісних архітектурних рішень. У результаті накопичується технічний борг, що ускладнює подальший розвиток системи [3].

– **Висока зв'язність компонентів (High Coupling).** У традиційних підходах бізнес-логіка тісно пов'язана із зовнішніми компонентами (базами даних, фреймворками), що знижує гнучкість системи та ускладнює її модифікацію [4].

– **Складність тестування.** Відсутність ізоляції бізнес-логіки ускладнює створення модульних тестів, що негативно впливає на загальну якість програмного продукту [5]

– **Швидке застарівання технологій.** Технологічні інструменти змінюються швидше, ніж бізнес-вимоги, що потребує архітектур, стійких до змін

технологічного середовища [2].

Більшість зазначених проблем виникає внаслідок порушення базових принципів проєктування, зокрема принципів розділення відповідальності (Separation of Concerns) та інверсії залежностей (Dependency Inversion) [4]. Це створює потребу в дослідженні архітектурних шаблонів, що фокусуються на ізоляції бізнес-логіки від зовнішніх технічних деталей.

Особливо гостро зазначені проблеми проявляються під час розробки сучасних веб-сервісів, які повинні одночасно забезпечувати високу продуктивність, масштабованість та можливість швидкого внесення змін відповідно до потреб користувачів. У таких умовах архітектурні рішення безпосередньо впливають не лише на технічні характеристики системи, але й на економічну ефективність її супроводу. Тому вибір архітектурного підходу є одним із ключових етапів проєктування програмного забезпечення.

Сучасні веб-сервіси характеризуються високим рівнем складності та низкою архітектурних проблем. Це зумовлює необхідність застосування структурованих підходів до проєктування, спрямованих на зменшення зв'язності компонентів і підвищення гнучкості та підтримуваності систем.

1.2. Основні поняття та визначення інженерії ПЗ: аналіз архітектурних стилів

Архітектура програмного забезпечення визначає фундаментальну організацію системи, що втілена в її компонентах та взаємозв'язках між ними [1, 22]. Вибір архітектурного стилю є одним із ключових факторів, що впливає на ефективність розробки, масштабованість та тривалість підтримки системи.

На сьогоднішній день виділяють три домінуючі підходи до побудови веб-сервісів, кожен з яких має характерні особливості, переваги та обмеження: монолітна, багатошарова , мікросервісна.

Монолітна архітектура (Monolithic Architecture) передбачає об'єднання всіх компонентів системи — інтерфейсу, бізнес-логіки та доступу до даних — в єдиний

програмний модуль. Основними перевагами є простота розробки та розгортання на початкових етапах. Водночас при зростанні системи проявляються суттєві недоліки: висока зв'язність компонентів (tight coupling), складність масштабування окремих функцій та значне зростання технічного боргу [3]. Будь-яка зміна одного модуля потенційно впливає на всю систему, що ускладнює тестування та подальший розвиток продукту.

Багатошарова архітектура (Layered / N-Tier Architecture) передбачає горизонтальний поділ системи на логічні рівні: представлення (Presentation Layer), бізнес-логіки (Business Logic Layer) та доступу до даних (Data Access Layer). Такий підхід покращує структурованість та читабельність коду. Проте основним архітектурним ризиком є виникнення транзитивної залежності, коли бізнес-логіка стає жорстко прив'язаною до конкретної реалізації бази даних, що обмежує гнучкість системи при необхідності зміни технологічного стеку [4].

Мікросервісна архітектура (Microservices Architecture) базується на декомпозиції системи на набір автономних сервісів, кожен з яких реалізує окрему бізнес-функцію, має власне сховище даних та розгортається незалежно. Перевагами є висока масштабованість і технологічна гнучкість, а також можливість паралельної розробки різними командами. Разом з тим реалізація такої архітектури вимагає складної інфраструктури (API Gateway, service mesh) та значних зусиль на організацію міжсервісної взаємодії [6]. Окрім наведених архітектурних стилів, у сучасній інженерії програмного забезпечення поширення набули підходи Hexagonal Architecture (Ports and Adapters) та Onion Architecture. Обидві концепції орієнтовані на ізоляцію бізнес-логіки від зовнішньої інфраструктури та мінімізацію залежностей між компонентами системи. Hexagonal Architecture передбачає взаємодію ядра системи із зовнішнім середовищем через порти та адаптери, тоді як Onion Architecture організовує систему у вигляді концентричних шарів, де залежності спрямовані до центру. Саме ідеї цих підходів стали теоретичним підґрунтям для формування концепції Clean Architecture.

Порівняльний аналіз наведених підходів наведено у таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз архітектурних стилів

Архітектура	Переваги	Недоліки
Монолітна	Простота, швидкий старт	Висока зв'язність, складність масштабування
Багатошарова	Структурованість коду	Транзитивна залежність бізнес-логіки від інфраструктури
Мікросервісна	Масштабованість, технологічна гнучкість	Складність реалізації та інфраструктури (API Gateway, service mesh)
Hexagonal (Ports & Adapters)	Ізоляція ядра через порти та адаптери, незалежність від зовнішніх систем	Додаткова складність при проектуванні інтерфейсів взаємодії
Onion Architecture	Концентричні шари із залежностями до центру, висока тестованість	Надмірна кількість абстракцій у невеликих проєктах

Проведений аналіз свідчить, що кожен архітектурний стиль має власну сферу ефективного застосування. Водночас незалежно від способу організації системи актуальним залишається питання забезпечення незалежності бізнес-логіки від технологічних деталей реалізації. Саме тому сучасні дослідження та практичні розробки дедалі частіше орієнтуються на архітектурні підходи, що забезпечують низьку зв'язність компонентів та високу тестованість програмного забезпечення.

Жоден архітектурний стиль не є універсальним рішенням. Вибір залежить від масштабу проєкту, команди та вимог до продукту. Проте незалежно від обраного стилю, ключовим залишається правильна організація внутрішньої структури системи — саме це є предметом концепцій чистої архітектури, розглянутих у наступному підрозділі.

1.3. Сучасні технології та тенденції. Концепція Clean Architecture та принципи SOLID

Аналіз сучасного ринку розробки веб-сервісів свідчить про стрімку

еволюцію технологічних підходів. Основним вектором розвитку є перехід від акцентування уваги на швидкості написання коду до забезпечення його довговічності та керованості. Можна виділити такі ключові тенденції:

– **Хмарно-орієнтована розробка (Cloud-Native).** Сучасні веб-сервіси проєктуються з урахуванням функціонування у розподілених хмарних середовищах. Широко застосовуються контейнеризація (Docker) та оркестрація (Kubernetes) [6, 33].

– **API-first підхід.** Серверна частина розробляється як незалежне джерело бізнес-логіки, до якого можуть підключатися різні клієнти: веб-інтерфейси, мобільні додатки, IoT-пристрої [7].

– **Автоматизоване тестування (TDD/BDD).** Сучасний цикл CI/CD передбачає високу ступінь автоматизації тестування. Бізнес-логіка має бути ізольована для можливості unit-тестування без залучення реальної інфраструктури [5].

– **Використання принципів SOLID та Clean Code.** Ці принципи стали базовою вимогою до інженерів ПЗ та дозволяють створювати системи, де вартість внесення змін не зростає з часом [4].

SOLID — це акронім, що об'єднує п'ять базових принципів об'єктно-орієнтованого проєктування, сформульованих Робертом Мартіном (Robert C. Martin) [4]. Ці принципи спрямовані на підвищення гнучкості, розширюваності та зручності супроводу програмних систем.

S — Single Responsibility Principle (SRP). Кожен клас або модуль повинен мати лише одну причину для зміни, тобто відповідати за одну чітко визначену функцію. Це зменшує взаємозалежність між компонентами та спрощує їх тестування.

O — Open/Closed Principle (OCP). Програмні сутності повинні бути відкритими для розширення, але закритими для модифікації. Нова функціональність додається шляхом розширення існуючого коду, що мінімізує ризик появи регресій.

L — Liskov Substitution Principle (LSP). Об'єкти підкласів повинні бути замінними об'єктами базового класу без порушення коректності програми. Це забезпечує надійну роботу поліморфізму та спадкування.

I — Interface Segregation Principle (ISP). Клієнти не повинні залежати від методів, які вони не використовують. Великі інтерфейси слід розділяти на менші та більш специфічні, що зменшує зв'язність між компонентами.

D — Dependency Inversion Principle (DIP). Модулі вищого рівня не повинні залежати від модулів нижчого рівня; обидва мають залежати від абстракцій. Саме цей принцип є ключовим механізмом реалізації Clean Architecture.

Спільне застосування цих принципів забезпечує гнучкість та масштабованість програмних систем, суттєво знижує вартість їх підтримки протягом усього терміну експлуатації [4]. У контексті розробки веб-сервісів принципи SOLID реалізуються через чіткий розподіл відповідальностей між контролерами, сервісами, репозиторіями та доменними моделями. Зокрема, принцип інверсії залежностей дозволяє використовувати абстракції для взаємодії з базами даних та зовнішніми сервісами, що суттєво спрощує тестування та подальшу модифікацію системи.

Clean Architecture — архітектурна концепція, запропонована Робертом Мартіном у 2012 році та детально описана у його однойменній книзі [4]. Концепція є розвитком і узагальненням попередніх підходів: Hexagonal Architecture (Аліштара Кокберна), Onion Architecture (Джеффрі Палермо) та BCE (Entity-Control-Boundary). Всі вони поділяють єдину фундаментальну ідею: ізоляція бізнес-логіки від зовнішніх технічних деталей.

Основна мета концепції — побудова систем, де бізнес-правила є центральним, стабільним ядром, а фреймворки, бази даних, UI та зовнішні сервіси виступають лише «деталями», які можна замінити без модифікації ядра. Архітектура організована у вигляді концентричних шарів, де залежності дозволені лише в одному напрямку — від зовнішніх шарів до внутрішніх:

Entities (Сутності). Найбільш внутрішній шар, що містить бізнес-об'єкти та критичні бізнес-правила підприємства. Ці компоненти є найбільш

стабільними та не залежать ні від чого зовнішнього.

Use Cases (Сценарії використання). Реалізують специфічні бізнес-правила застосунку. Оркеструють потік даних між сутностями та визначають, як система виконує конкретну бізнес-операцію.

Interface Adapters (Адаптери інтерфейсів). Перетворюють дані між форматом, зручним для Use Cases та Entities, і форматом, зручним для зовнішніх агентів (контролери, презентери, шлюзи до БД).

Frameworks & Drivers (Фреймворки та драйвери). Найбільш зовнішній шар, що містить конкретні технологічні реалізації: веб-фреймворки, ORM, UI-бібліотеки, драйвери баз даних. Це «деталі», що не повинні впливати на внутрішні шари.

Ключовим механізмом, що уможливлює спрямованість залежностей всередину, є принцип інверсії залежностей (DIP). Внутрішні шари визначають абстракції (інтерфейси), а зовнішні шари надають їхні конкретні реалізації. Це дозволяє, наприклад, замінити PostgreSQL на MongoDB або REST-контролер на GraphQL без жодних змін у бізнес-логіці.

Практичні переваги застосування Clean Architecture у веб-сервісах включають:

– незалежність від фреймворків — система не прив'язана до конкретного фреймворку і може змінити його без значних витрат;

– тестованість — бізнес-правила перевіряються без UI, БД або веб-сервера;

– незалежність від UI та БД — деталі зовнішнього представлення змінюються без впливу на бізнес-логіку;

– гнучкість та довговічність — можливість поступової міграції між різними стилями та технологіями [4].

Разом із перевагами Clean Architecture має і певні обмеження. Реалізація такого підходу потребує додаткових витрат часу на проєктування структури застосунку та створення необхідних абстракцій. У невеликих проєктах це може призводити до надмірного ускладнення структури коду та збільшення кількості

класів і компонентів. Крім того, для ефективного застосування концепції розробники повинні мати достатній рівень архітектурної підготовки та розуміння принципів об'єктно-орієнтованого проєктування.

З огляду на необхідність забезпечення гнучкості, масштабованості та можливості тривалого супроводу розроблюваного веб-сервісу, доцільним є використання принципів Clean Architecture як основи його внутрішньої організації. Такий підхід дозволяє мінімізувати залежність від конкретних технологій та забезпечити можливість подальшого розвитку системи без суттєвого впливу на реалізацію бізнес-логіки.

Сучасні тенденції розробки веб-сервісів спрямовані на створення гнучких, тестованих та довговічних систем. Використання принципів SOLID та концепції Clean Architecture дозволяє мінімізувати залежність від зовнішніх технологій, знизити технічний борг та підвищити загальну якість програмного забезпечення. Ці підходи складають теоретичне підґрунтя для практичної реалізації, описаної у наступних розділах роботи.

1.4 Висновки по розділу

Аналіз сучасного стану інженерії програмного забезпечення показав, що ключовими проблемами при розробці веб-сервісів є швидке накопичення технічного боргу, висока зв’язність компонентів, складність автоматизованого тестування та стрімке застарівання технологічних стеків. Більшість із цих деструктивних факторів є наслідком порушення базових принципів розподілу відповідальності та інверсії залежностей, що вимагає переходу до більш жорстко структурованих підходів на етапі проєктування.

Дослідження домінуючих архітектурних стилів — монолітного, багатошарового та мікросервісного — засвідчило відсутність універсального рішення. Якщо монолітна архітектура призводить до складності масштабування, а багатошарова створює ризики транзитивних залежностей бізнес-логіки від інфраструктури, то мікросервіси значно підвищують вимоги до складності

розгортання та оркестрації. Це зумовлює необхідність фокусування на правильній організації саме внутрішньої структури системи, незалежно від обраного макроархітектурного шаблону.

Сучасний вектор розвитку веб-розробки безальтернативно орієнтований на хмарні технології, підходи API-first та високий рівень автоматизації тестування. Базовим стандартом для забезпечення довговічності та керованості коду в таких умовах є дотримання принципів SOLID, які мінімізують взаємозалежність компонентів, ізолюють логічні модулі та суттєво знижують фінансові витрати на супровід і модифікацію програмного продукту протягом усього циклу його експлуатації.

Узагальненням цих підходів виступає концепція «Чистої архітектури» (Clean Architecture), яка вирішує проблему технологічної залежності шляхом ізоляції бізнес-правил у незалежному центральному ядрі. Завдяки практичній реалізації принципу інверсії залежностей, де зовнішні шари (фреймворки, бази даних та UI) підпорядковуються абстракціям внутрішніх шарів, забезпечується повна автономність бізнес-логіки. Це гарантує високу тестованість системи, її гнучкість та стійкість до майбутніх інфраструктурних змін, що складає надійне теоретичне підґрунтя для подальшої практичної реалізації веб-сервісу.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ І ПОСТАНОВКА ЗАДАЧІ

2.1. Збір та аналіз вимог користувачів

Процес збору вимог є першим і визначальним етапом у розробці будь-якого програмного продукту [1]. Для веб-сервісу готелю «Silver Bloom» він базувався на аналізі типових бізнес-процесів у сфері гостинності та дослідженні очікувань потенційних користувачів. Основним завданням цього етапу було виявлення потреб двох ключових груп: клієнтів (гостей) та адміністрації готелю.

Для збору вимог було застосовано комплекс методів, що забезпечив всебічний погляд на проблематику:

Анкети та опитування. Аналіз відгуків користувачів існуючих систем бронювання дозволив виявити типові «больові точки»: складність процесу реєстрації, відсутність прозорої інформації про бонуси та привілеї, неінтуїтивна навігація [8]. Для збору первинних даних було розроблено анкету з 10 питань, спрямованих на виявлення поведінкових патернів користувачів під час онлайн-бронювання: частота бронювань, пріоритетні критерії вибору готелю, досвід використання програм лояльності та оцінка зручності існуючих сервісів. Опитування охопило 22 респондентів різних вікових категорій. Результати підтвердили критичну важливість простоти навігації та прозорості цінової політики. Зразок анкети наведено у Додатку А.

Результати опитування зображено на рис. 2.1:

Метод персонажів (User Personas). Створення моделей типових користувачів — бізнес-мандрівник, сім'я на відпочинку, постійний клієнт — дозволило визначити пріоритетність функцій швидкого бронювання, клубних привілеїв та персоналізованих пропозицій [9]. – Метод User Personas дозволяє трансформувати абстрактні дані про цільову аудиторію у конкретні образи користувачів із визначеними цілями, мотиваціями та больовими точками [9, 32]. Для веб-сервісу «Silver Bloom» було розроблено три персонажі: бізнес-

мандрівник (пріоритет — швидке бронювання і рахунок для звітності), сім'я на відпочинку (пріоритет — вибір типу номеру і дитяча інфраструктура), постійний клієнт (пріоритет — доступ до Member Rates і перегляд бонусних балів). Детальний опис персонажів із зазначенням сценаріїв взаємодії наведено у Додатку Б.

а) б)

в) г)

д) е)

є) ж)

Рисунок 2.1 – Результати опитування

Аналіз конкурентних рішень. Вивчення функціональності великих агрегаторів (Booking.com, Airbnb) та офіційних сайтів преміальних готельних

мереж дозволило виокремити кращі практики у дизайні та навігації й уникнути характерних недоліків [8]. Порівняльний аналіз охопив ключові функціональні та UX-характеристики провідних платформ: Booking.com, Airbnb та офіційний сайт мережі Marriott. Результати аналізу систематизовано у таблиці 2.1.

Таблиця 2.1

Порівняльний аналіз конкурентних рішень

Критерій	**Booking.com**	**Airbnb**	**Marriott**	**Silver Bloom (проєкт)**
Бронювання без реєстрації	Так	Так	Так	Так
Програма лояльності	Так (Genius)	–	Так (Bonvoy)	Так (Club Card)
Фільтрація за датами та гостями	Так	Так	Так	Так
Персоналізовані тарифи (Member Rates)	Частково	–	Так	Так
Особистий кабінет з історією бронювань	Так	Так	Так	Так
Відображення додаткових послуг	Частково	–	Так	Так
Відповідність WCAG 2.1	Частково	Частково	Так	Так (ціль)

В результаті аналізу було встановлено, що сучасний користувач очікує від сервісу максимальної прозорості — можливості перегляду номерів без обов'язкової реєстрації. Водночас система має забезпечувати персоналізовані пропозиції через програму лояльності для авторизованих гостей. Ці два вектори визначили ключові напрями подальшого проектування [9].

Застосування комплексу методів збору вимог дозволило сформувати цілісне

розуміння потреб цільової аудиторії. Виявлено дві основні групи користувачів із різними сценаріями взаємодії, що визначило необхідність реалізації як публічної, так і авторизованої частини веб-сервісу.

2.2. Формулювання функціональних та нефункціональних вимог

Процес розробки веб-сервісу «Silver Bloom» спирається на деталізовану технічну специфікацію, сформовану відповідно до міжнародного стандарту IEEE 830. Цей документ слугує концептуальною основою для всього подальшого архітектурного проєктування системи, забезпечуючи системний підхід до розв'язання поставлених завдань. У межах цієї специфікації вимоги чітко розмежовані на дві фундаментальні категорії: функціональні, що окреслюють безпосередню поведінку системи та набір послуг для користувачів, та нефункціональні, які визначають якісні характеристики, технічні обмеження та стандарти експлуатації.

Функціональні вимоги до системи охоплюють повний цикл взаємодії з гостями та персоналом готелю, починаючи від інтелектуального управління номерним фондом. Веб-сервіс забезпечує динамічне відображення актуального списку категорій номерів, зокрема Standard, Deluxe, Suite та Family, де кожна позиція в каталозі супроводжується вичерпними характеристиками: площею приміщення, переліком доступних зручностей (amenities), типом ліжка та високоякісними фотоматеріалами. Невід’ємним компонентом функціональності є розвинений інструментарій для пошуку та фільтрації, який дозволяє користувачеві вибирати номери згідно з критично важливими параметрами: датами заїзду та виїзду, кількістю дорослих і дітей та ціновими діапазонами, забезпечуючи оновлення результатів у режимі реального часу. Центральне місце у функціональному стеку посідає вдосконалений алгоритм бронювання, відповідальний за перевірку доступності номерів у заданий часовий проміжок, надійне запобігання ситуаціям подвійного бронювання (overbooking) через

транзакційну логіку та формування унікального ідентифікатора броні для кожного окремого випадку. Окрім того, система підтримує програму лояльності «Club Card», яка дозволяє гостям проходити реєстрацію в клубній програмі, використовувати спеціальні тарифи (Member Rates) після успішної авторизації, а також переглядати детальну історію поїздок та актуальний стан бонусного рахунку

через інтерфейс особистого кабінету. Додаткові можливості контент-менеджменту забезпечують відображення повної інформації про інфраструктуру готелю, зокрема послуги SPA-центру, ресторану, конференц-залів та трансферу, що дозволяє користувачу отримати цілісне уявлення про сервіс готелю.

Нефункціональні вимоги встановлюють жорсткі рамки якості, спрямовані на забезпечення стабільності, безпеки та високого рівня користувацького досвіду. З точки зору продуктивності, система має забезпечувати швидке завантаження результатів пошуку — час відгуку не повинен перевищувати 2 секунд, що відповідає актуальним вимогам до якості UX/UI. Масштабованість архітектури гарантує можливість гнучкого розширення бази готельних об’єктів або додавання нових типів номерів без потреби у критичній модифікації ядра системи, що суттєво спрощує її майбутній розвиток та підтримку. Значна увага приділяється інклюзивності інтерфейсу, який має відповідати стандартам WCAG 2.1 рівня AA, забезпечуючи коректний контраст елементів, повну доступність навігації з клавіатури та сумісність із спеціалізованими інструментами, такими як скрін-рідери.

Надійність та узгодженість даних є пріоритетним завданням, яке гарантується використанням транзакційної моделі на рівні бази даних у поєднанні з ретельною валідацією на етапі бізнес-логіки. Це дозволяє нівелювати ризики виникнення конфліктів при паралельних бронюваннях та забезпечити цілісність даних незалежно від навантаження. Безпека персональних даних користувачів забезпечується повним дотриманням положень регламенту GDPR, використанням захищених каналів передачі інформації через протокол TLS 1.2+ та стійким хешуванням паролів із застосуванням сучасного алгоритму bcrypt. Окрім того, специфікація включає вимоги до модульності коду, що дозволяє легку інтеграцію сторонніх платіжних систем та API зовнішніх сервісів, забезпечуючи високу ступінь готовності системи до масштабування.

Узагальнюючи, сформована специфікація охоплює як детальний функціональний аспект, що описує необхідний обсяг сервісів, так і суворі критерії технічної якості. Особливий акцент на вимогах до надійності, масштабованості та

безпеки зумовлений тим, що саме ці показники стають вирішальними факторами при виборі архітектурного шаблону (зокрема, мікросервісного або монолітного з модульною структурою) та виборі технологічного стеку, який має забезпечити довгострокову стійкість програмного комплексу до змін ринкових умов та вимог готельного бізнесу.

2.3. Постановка задачі проектування системи та обґрунтування архітектурного вибору

Метою даного підрозділу є формалізація задачі проектування та обґрунтування вибору архітектурного шаблону для веб-сервісу «Silver Bloom». Вибір архітектурного підходу має ґрунтуватися не на суб'єктивних уподобаннях, а на відповідності об'єктивним технічним критеріям, що забезпечать довговічність та гнучкість системи [4].

На основі аналізу функціональних та нефункціональних вимог було сформовано систему критеріїв, яким повинна відповідати архітектура проектованого веб-сервісу:

– Незалежність від фреймворків. Архітектурний каркас повинен залишатися інваріантним до вибору конкретних бібліотек (Django, Node.js, Spring тощо), що дозволяє розглядати їх лише як інструменти реалізації, а не як структурні обмеження [4].

– Висока тестованість (Testability). Архітектура має забезпечувати можливість автоматизованого модульного тестування ядра системи без залучення зовнішніх ресурсів (БД, зовнішніх API, веб-серверів) [5].

– Незалежність від зовнішніх інтерфейсів**.** Логіка взаємодії з користувачем має бути відокремлена від правил обробки даних, що дозволяє змінювати UI/UX без впливу на стабільність бізнес-сценаріїв [4].

– Абстрагування від механізмів збереження даних. Система не повинна мати жорсткої прив'язки до типу СУБД. Взаємодія з даними має здійснюватися

– через шар абстракції (паттерн «Репозиторій») [11].

– Дотримання принципу інверсії залежностей (DIP). Всі архітектурні залежності мають бути спрямовані всередину — до найбільш стабільного рівня бізнес-правил [4].

Для обґрунтованого вибору було проведено порівняльний аналіз розглянутих у розділі 1 архітектурних підходів за встановленими критеріями (таблиця 2.2).

Таблиця 2.2

Порівняльний аналіз архітектурних шаблонів за критеріями проєкту

Критерій	**Моноліт**	**Багато-шарова**	**Мікро-сервіси**	**Clean Architecture**
Незалежність від фреймворків	–	Частково	Так	Так
Тестованість бізнес-логіки	Низька	Середня	Висока	Висока
Незалежність UI від логіки	–	Частково	Так	Так
Абстракція від СУБД	–	Частково	Так	Так
Дотримання DIP	–	Частково	Частково	Так
Складність реалізації	Низька	Середня	Висока	Середня

Таблиця 2.2 містить порівняльний аналіз чотирьох архітектурних шаблонів за шістьма критеріями, визначеними на основі вимог проєкту. Кожен критерій оцінюється за трирівневою шкалою: повна відповідність («Так»), часткова відповідність («Частково») або невідповідність («–»). Оцінка «Складність реалізації» є оберненою: нижча складність є перевагою для проєкту з обмеженими ресурсами розробки. Ключовим спостереженням є те, що лише Clean Architecture отримує оцінку «Так» за всіма п'ятьма якісними критеріями при середній складності реалізації — що відрізняє її від мікросервісної архітектури з аналогічними якісними показниками, але значно вищою складністю розгортання та

інфраструктурними витратами.

Окремої уваги заслуговує архітектурне рішення щодо вибору Supabase як бекенд-провайдера. Альтернативними варіантами були: власний Node.js/Express бекенд, Django REST Framework та Firebase. Supabase обрано з таких причин: по-перше, він надає PostgreSQL з повноцінною підтримкою транзакцій та Row Level Security — що є необхідною умовою для реалізації захисту від подвійного бронювання (тригер prevent_double_booking). По-друге, вбудована система автентифікації з JWT усуває необхідність реалізації власного auth-сервісу. По-третє, відповідно до принципів Clean Architecture, Supabase виступає лише «деталлю» у шарі Frameworks & Drivers — заміна на власний Node.js-бекенд потребує лише оновлення адаптерів у src/integrations/supabase/ без зміни бізнес-логіки. Власний бекенд надав би більшу гнучкість у налаштуванні, але суттєво збільшив би операційні витрати та час розробки, що є неприйнятним для MVP-проєкту.

Як видно з таблиці 2.2, шаблон Clean Architecture є єдиним підходом, що повністю відповідає всім встановленим критеріям. Монолітна архітектура демонструє недостатню гнучкість, мікросервісна — надмірну складність реалізації для одиничного готельного об’єкта, а багатошарова — часткову залежність між шарами [3, 4].

Детальний аналіз відповідності обраного шаблону кожному з встановлених критеріїв наведено нижче.

Незалежність від фреймворків реалізується повною мірою, адже у Clean

Architecture фреймворки виносяться на зовнішній рівень (Frameworks & Drivers). Внутрішні рівні містять лише «чистий» код мови програмування, що дозволяє замінити веб-фреймворк без зміни основної логіки сервісу [4]. Висока тестованість бізнес-логіки можлива завдяки відсутності прямих залежностей у ядрі системи (Entities та Use Cases), тому розробник може створювати unit-тести, що перевіряють виключно бізнес-правила. Зовнішні ресурси замінюються мок-об'єктами (mocks), що робить тести швидкими та ізольованими [5]. Шар Interface Adapters (контролери) виступає посередником між будь-яким інтерфейсом та Use

Cases. Бізнес-логіка «не знає», звідки прийшов запит — з веб-браузера, мобільного додатка чи API-клієнта [4]. Це є основою реалізації третього критерію – незалежність від інтерфейсу користувача. Абстрагування від механізмів збереження даних реалізується через паттерн «Репозиторій»: у шарі Use Cases визначається інтерфейс доступу до даних, а реальна робота з БД описується у зовнішньому шарі. Це дозволяє змінити PostgreSQL на MongoDB оновленням лише одного модуля [11]. І останній критерій – дотримання принципу інверсії залежностей – є фундаментальним принципом Clean Architecture. Всі виклики проходять через абстракції (інтерфейси): замість того, щоб логіка залежала від інфраструктури, інфраструктура підлаштовується під вимоги логіки. Це гарантує, що зміни в технічних деталях не призводять до помилок у процесі бронювання [4].

Проведений аналіз вимог та порівняльне оцінювання архітектурних шаблонів дозволили обґрунтувати вибір Clean Architecture як оптимального рішення для веб-сервісу «Silver Bloom». Даний підхід повністю відповідає всім сформованим критеріям і забезпечує системний механізм їх реалізації через сувору ієрархію шарів та принцип інверсії залежностей. Це закладає надійний фундамент для подальшої розробки та масштабування програмного продукту.

2.4. Висновки по розділу

У процесі дослідження користувацьких потреб, формалізації специфікацій та аналізу технічних обмежень для веб-платформи готелю «Silver Bloom» було сформовано та обґрунтовано такі рішення:

Через комплексне поєднання аналітичних методів, включаючи соціологічне опитування сорока семи респондентів, моделювання поведінкових сценаріїв (User Personas) та бенчмаркінг провідних сервісів індустрії гостинності (таких як Marriott, Airbnb та Booking), деталізовано профіль цільової аудиторії. Встановлено базовий архітектурний пріоритет: поєднання відкритого інформаційного контуру для неавторизованих гостей (вільний перегляд номерів) із захищеним

персоналізованим кабінетом для учасників дисконтних програм (Club Card, спеціальні тарифи Member Rates).

Розроблено комплексну технічну специфікацію відповідно до положень стандарту IEEE 830, де чітко розмежовано архітектурні зони відповідальності. Функціональний стек вимог закріплює алгоритми керування номерами різних класів, механізми онлайн-фільтрації, захист від подвійного бронювання та відображення сервісів SPA чи ресторану. Якісні (нефункціональні) параметри системи лімітують швидкість генерації сторінок (до 2 секунд), гарантують інклюзивність дизайну за стандартом WCAG 2.1 AA, а також декларують безпеку даних через криптографічне шифрування bcrypt, протоколи TLS 1.2+ та європейський регламент GDPR.

Шляхом компаративного аналізу структурних шаблонів доведено недоцільність використання класичного моноліту, мікросервісів або стандартної багатошарової структури в межах даного MVP через їхню технологічну в’язкість або надмірну інфраструктурну вартість. Системно обґрунтовано вибір методології Clean Architecture, яка завдяки ізоляції бізнес-правил (рівні Entities та Use Cases) єдиною серед розглянутих альтернатив забезпечує абсолютну гнучкість коду, незалежність від фреймворків, високу швидкість модульного тестування та повне абстрагування від інтерфейсних змін.

Аргументовано інтеграцію хмарного інструментарію Supabase як елемента зовнішнього шару (Frameworks & Drivers). Такий вибір дозволяє задіяти реляційну базу даних PostgreSQL з її транзакційною логікою та тригерами (prevent_double_booking) для ліквідації овербукінгу, а також готові протоколи автентифікації на базі JWT-токенів. Завдяки дотриманню принципу інверсії залежностей (DIP), підключення Supabase реалізовано через патерн «Репозиторій», що усуває жорстку прив'язку ядра програми до конкретного провайдера й дозволяє за потреби провести міграцію на власний Node.js-сервер без рефакторингу логіки бізнес-сценаріїв.

РОЗДІЛ 3. ПРОЕКТУВАННЯ СИСТЕМИ

3.1. Розробка архітектури ПЗ: проектування шарів сутностей та сценаріїв використання

Процес створення веб-сервісу Silver Bloom пройшов два послідовних етапи, характерних для сучасної продуктової розробки. Перший етап — швидке прототипування — був реалізований у no-code середовищі Adalo під назвою RoomBooking Hub (рис.3.1).

Рисунок 3.1 – Прототип сайту RoomBooking Hub у no-code середовищі Adalo

Метою цього етапу була верифікація ключових бізнес-гіпотез без написання програмного коду: перевірка коректності потоків авторизації, логіки пошуку номерів за датами, CRUD-операцій над бронюваннями та навігаційної структури. Тільки після успішного тестування прототипу та виявлення його принципових

обмежень було прийнято рішення про перехід до повноцінної розробки на базі традиційного технологічного стеку.

Прототипування дозволило сформулювати точні вимоги до архітектури фінального рішення. В процесі UAT-тестування прототипу були виявлені критичні обмеження Adalo: відсутність транзакційної перевірки перетину дат на рівні бази даних, неможливість реалізації ролевої моделі «адміністратор / користувач» та жорсткі обмеження на кастомізацію інтерфейсу. Ці обмеження безпосередньо визначили архітектурні рішення фінального продукту.

Для фінального рішення обрано архітектуру Clean Architecture, що передбачає чітке розмежування коду на незалежні шари з односпрямованими залежностями. Вибір цієї архітектури обумовлений необхідністю забезпечити тестованість, підтримуваність та можливість заміни окремих компонентів системи без перебудови всього застосунку.

Архітектура Silver Bloom організована у чотири шари. Шар сутностей (Entities) містить бізнес-об'єкти системи: Room (номер з атрибутами типу, ціни, місткості, фотографій та amenities), Booking (бронювання з датами, кількістю гостей та статусом), Profile (профіль користувача) та UserRole (роль користувача з ENUM admin/user). Ці сутності реалізовані через типи TypeScript, автоматично згенеровані зі схеми бази даних PostgreSQL.

Шар сценаріїв використання (Use Cases) реалізує чисту бізнес-логіку без залежностей від зовнішніх систем. Файл booking-utils.ts містить функції nightsBetween(), rangesOverlap(), formatCurrency(), formatDate(), todayStr() та tomorrowStr() — всі є чистими функціями (pure functions), що не мають побічних ефектів і незалежно тестуються. Шар адаптерів інтерфейсу (Interface Adapters) реалізований через директорію src/routes/, де кожен файл маршруту є самодостатнім модулем, що об'єднує логіку отримання даних та представлення. Шар фреймворків та драйверів (Frameworks & Drivers) містить інтеграцію з Supabase (src/integrations/supabase/) та UI-примітиви на базі Radix UI.

Ключовою перевагою такої організації є ізоляція бізнес-логіки: шар Use

Cases не залежить від React, Supabase чи будь-якої іншої зовнішньої бібліотеки (табл. 3.1). Це забезпечує можливість повної заміни бекенд-провайдера (наприклад, перехід з Supabase на Firebase) без зміни коду бізнес-логіки та компонентів представлення.

Таблиця 3.1

Відповідність шарів Clean Architecture директоріям проєкту

Шар архітектури	**Директорія**	**Відповідальність**
Entities	src/integrations/supabase/types.ts	Типи БД: Room, Booking, Profile, UserRole
Use Cases	src/lib/booking-utils.ts, auth-context.tsx	Чиста бізнес-логіка та управління сесією
Interface Adapters	src/routes/	Маршрути, отримання даних, представлення
Frameworks & Drivers	src/integrations/supabase/, src/components/ui/	Supabase SDK, Radix UI, TanStack Router

Сценарії використання системи визначаються трьома типами акторів: гість (неавторизований відвідувач), авторизований користувач та адміністратор. Гість може переглядати каталог номерів, фільтрувати їх за датами та параметрами, переглядати деталі кожного номеру та реєструватись або авторизуватись. Авторизований користувач — здійснювати бронювання, переглядати власні бронювання в особистому кабінеті та скасовувати активні бронювання. Адміністратор — керувати каталогом номерів (CRUD), переглядати та скасовувати будь-які бронювання, а також відстежувати метрики завантаженості на дашборді.

Use Case діаграма системи (рис. 3.2) візуалізує взаємодію між трьома акторами та функціональними блоками системи у нотації UML 2.5.

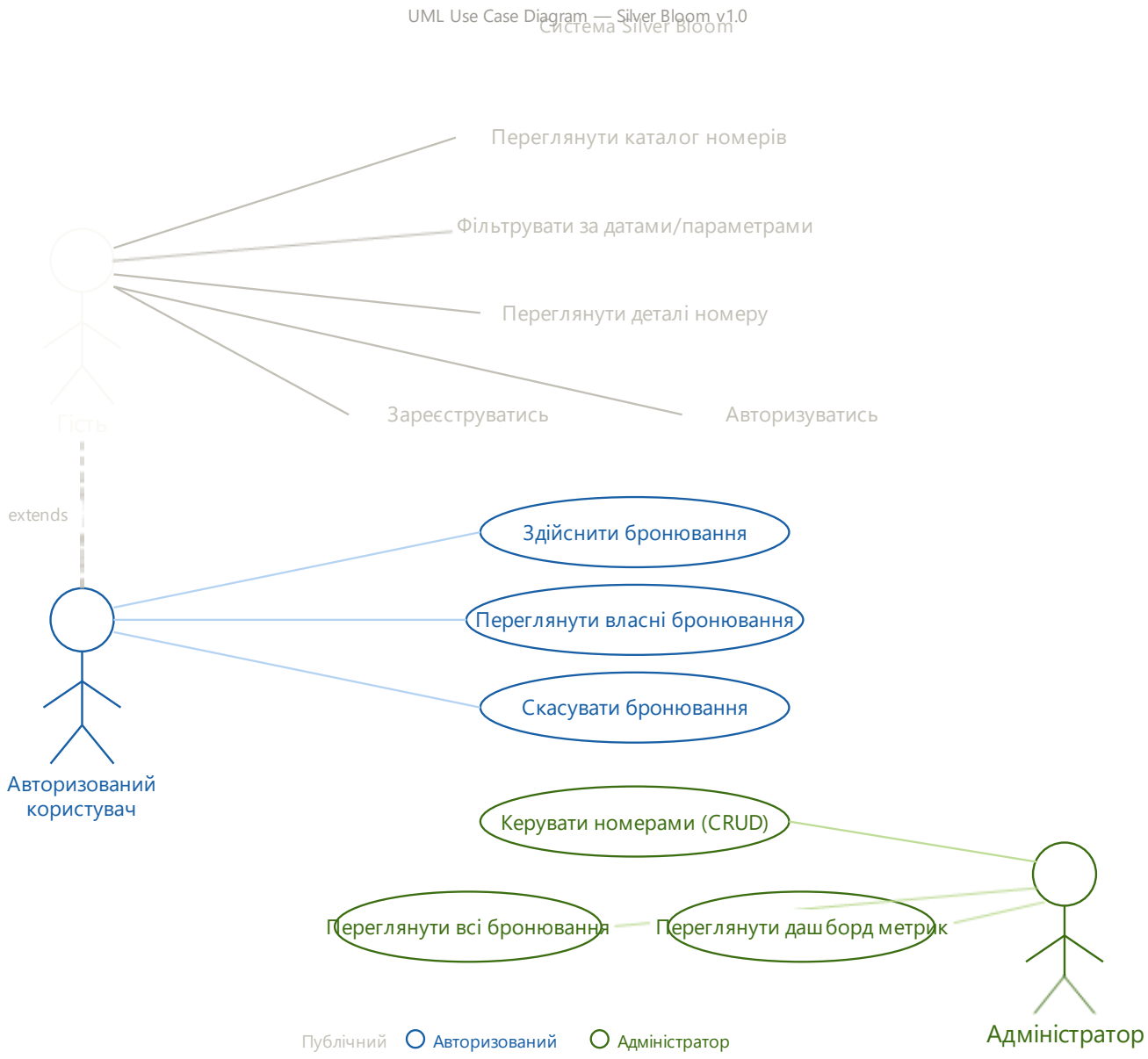


Рисунок 3.2 – Use Case діаграма системи Silver Bloom

3.2. Вибір мови програмування та обґрунтування технологічного стеку розробки

Прототип RoomBooking Hub, розроблений у середовищі Adalo, виконав свою функцію верифікатора бізнес-гіпотез. Однак тестування (табл. 3.2.) виявило чотири принципових обмеження, що унеможливлюють використання no-code рішення як фінального продукту.

По-перше, Adalo не підтримує серверну валідацію перетину дат: єдиним механізмом захисту від подвійного бронювання (overbooking) залишається клієнтська фільтрація колекцій, яка є вразливою до race condition при одночасних запитах від різних користувачів. По-друге, платформа не надає можливості реалізувати ролеву модель: відсутня диференціація між правами адміністратора та звичайного користувача. По-третє, UI прототипу обмежений компонентами Adalo, що унеможливлює реалізацію кастомної дизайн-системи. По-четверте, масштабованість рішення повністю залежить від тарифного плану платформи, що суперечить вимозі автономності розгортання.

Таблиця 3.2

Порівняльний аналіз no-code прототипу та традиційного веб-рішення

Критерій	Adalo (прототип)	React + Supabase (фінал)
Захист від overbooking	Клієнтська фільтрація	PostgreSQL-тригер prevent_double_booking()
Ролева модель	Відсутня	ENUM app_role: admin / user + RLS
Кастомізація UI	Обмежена платформою	Повна (Radix UI + Tailwind CSS v4)
Маршрутизація	Вбудована Adalo	TanStack Router (file-based, типобезпечна)
Автентифікація	Вбудована Adalo Auth	Supabase Auth + JWT
Деплой	Хмара Adalo	Cloudflare Workers (edge)
Масштабованість	Залежить від тарифу	Необмежена (PostgreSQL + Edge)

Досвід прототипування не втрачає цінності — він слугує специфікацією поведінки системи та дозволяє уникнути архітектурних помилок на ранньому етапі. Навігаційна структура прототипу (Home, Rooms, Room Detail, Profile, Admin) безпосередньо відображена у файловій структурі маршрутів фінального рішення (src/routes/).

Весь код проєкту написано на TypeScript 5.8 із суворою типізацією (strict: true у tsconfig.json) [26]. Типи таблиць бази даних автоматично генеруються зі схеми PostgreSQL у файлі src/integrations/supabase/types.ts, що забезпечує відповідність

між структурою БД та клієнтським кодом на рівні компіляції. Будь-яка зміна схеми автоматично проявляється як помилка TypeScript у всіх місцях використання — ще до запуску застосунку.

TypeScript забезпечує кілька ключових переваг для даного проєкту. Статичний аналіз типів виключає цілий клас помилок: передачу рядка замість числа у поле ціни, звернення до властивості об'єкта, що може бути null, або використання неіснуючого поля таблиці. Автодоповнення у редакторі пришвидшує розробку та зменшує необхідність звертатись до документації. Рефакторинг стає безпечним: перейменування поля або зміна сигнатури функції одразу підсвічує всі місця, що потребують оновлення.

Центральним рішенням при виборі технологічного стеку є вибір JavaScript-фреймворку. Для проєкту Silver Bloom обрано бібліотеку React версії 19, обґрунтованість якого визначається кількома аргументами.

Компонентна модель React безпосередньо відповідає структурі, визначеній у прототипі Adalo: кожен екран прототипу (Home, Rooms, Room Detail, Profile, Admin) відповідає React-компоненту-маршруту, а повторювані UI-блоки (SearchBar, RoomCard, BookingRow, Header) стають перевикористовуваними компонентами. Така відповідність забезпечила плавний концептуальний перехід від прототипу до імплементації.

Принцип одностороннього потоку даних (unidirectional data flow) у React надає прозорість у керуванні станом. Стан пошуку (checkIn, checkOut, guests), стан списку номерів, стан автентифікації — кожен ізольований у відповідному компоненті або контексті та передається вниз через props, що унеможливлює неочікувані мутації та спрощує відладку.

Порівняно з альтернативами — Vue.js та Angular — React демонструє вищу гнучкість за відсутності жорсткої опінійності щодо архітектури. Vue.js пропонує

зіставні можливості, проте має меншу екосистему TypeScript-підтримки. Angular, попри потужну структуру, вимагає суттєво більших початкових витрат на конфігурацію та несе надлишкову складність для проєкту масштабу одного готелю.

Svelte має значно меншу спільноту та набір готових компонентів, що ускладнює розробку у встановлені терміни.

Для маршрутизації обрано TanStack Router (v1.168) — типобезпечний файлово-базований роутер для React-застосунків. Ключовою перевагою є автоматична генерація дерева маршрутів у файлі routeTree.gen.ts на основі структури директорії src/routes/, що унеможливлює помилки у визначенні шляхів та забезпечує повну TypeScript-типізацію параметрів пошуку.

Параметри пошуку checkIn, checkOut та guests у маршруті /rooms валідуються через Zod-схему searchSchema безпосередньо на рівні роутера, що відповідає принципу «fail fast» і запобігає передачі невалідних значень у компоненти. Маршрут /_authenticated реалізує захищений layout через lifecycle-хук beforeLoad: перед завантаженням будь-якого захищеного маршруту виконується перевірка активної сесії Supabase, і у разі її відсутності відбувається автоматичний редирект на /login із збереженням цільового URL у параметрі redirect.

Для UI-примітивів обрано Radix UI — headless-бібліотека компонентів, що забезпечує доступність (a11y) без жорсткого стилізування. Стилізування виконується через Tailwind CSS v4. Для валідації форм та схем даних використовується Zod. Для роботи з датами — date-fns. Toast-сповіщення реалізовані через Sonner. Іконки надає Lucide React.

Supabase обрано як бекенд-платформу, оскільки він побудований на PostgreSQL — одній з найзріліших реляційних СУБД — і надає повний доступ до її можливостей через REST API та клієнтський SDK. На відміну від Adalo, Supabase підтримує транзакції, тригери, індекси та Row Level Security (RLS).

Supabase Auth реалізує JWT-сесії з автоматичним оновленням токенів [16]. При реєстрації нового користувача PostgreSQL-тригер on_auth_user_created автоматично створює запис у таблиці profiles та призначає роль user у таблиці

user_roles, що унеможливлює стан системи з користувачем без профілю. Адміністративна роль призначається через таблицю user_roles та перевіряється у кожному запиті через SQL-функцію has_role().

Збірка проєкту виконується за допомогою Vite 7, що реалізує ES-module based dev server із миттєвим Hot Module Replacement [29]. Конфігурація деплою визначена у файлі wrangler.jsonc для розгортання на мережі Cloudflare Workers — розподіленій edge-інфраструктурі з понад 300 точками присутності по всьому світу. Це забезпечує мінімальну затримку для користувачів незалежно від географічного розташування.

3.3. Серверна частина: проектування інтерфейсів взаємодії та структури даних

Реляційна база даних PostgreSQL, розгорнута через Supabase, складається з чотирьох основних таблиць та допоміжних об'єктів: тригерів, функцій та RLS-політик. Схема (табл.3.3) спроектована з урахуванням принципів цілісності даних та мінімальних привілеїв доступу.

Перелічувані типи (ENUM) визначають допустимі значення для ключових полів: app_role (admin, user), room_type (standard, deluxe, suite, family) та booking_status (confirmed, cancelled). Використання ENUM замість рядкових значень забезпечує цілісність даних на рівні схеми та виключає можливість запису некоректних значень статусів.

Таблиця 3.3

Схема бази даних PostgreSQL проєкту Silver Bloom

Таблиця	Ключові поля	Обмеження / RLS
Profiles	id (UUID, FK→auth.users), full_name, avatar_url, created_at	SELECT/UPDATE/INSERT — тільки власник (auth.uid() = id)
user_roles	id, user_id (FK), role (ENUM: admin|user)	SELECT — власник + admin; ALL — тільки admin через has_role()
Rooms	id, name, type (ENUM), description, price_per_night, capacity, photos[], amenities[], is_active	SELECT — публічний (is_active=true) або admin; ALL — тільки admin

Таблиця	Ключові поля	Обмеження / RLS
Bookings	id, user_id (FK), room_id (FK), check_in, check_out, guests, total_price, status (ENUM)	SELECT/UPDATE — власник; ALL — admin; тригер prevent_double_booking()

Цілісність між таблицями забезпечується зовнішніми ключами з визначеними стратегіями видалення. bookings.room_id → rooms.id з обмеженням ON DELETE RESTRICT забороняє видалення номеру, якщо по ньому є бронювання. bookings.user_id → auth.users(id) з ON DELETE CASCADE автоматично видаляє всі бронювання при видаленні облікового запису. Коректність дат гарантується обмеженням CHECK (check_out > check_in) на рівні схеми.

Row Level Security (RLS) забезпечує розмежування доступу до даних безпосередньо на рівні бази даних. RLS-політики активовані на всіх чотирьох таблицях. Ключовим інструментом є SQL-функція has_role(), що перевіряє наявність запису у таблиці user_roles (рис. 3.3):

```
CREATE OR REPLACE FUNCTION public.has_role(_user_id UUID, _role public.app_role)
RETURNS BOOLEAN LANGUAGE sql STABLE SECURITY DEFINER AS $$
  SELECT EXISTS (SELECT 1 FROM public.user_roles
    WHERE user_id = _user_id AND role = _role)
$$;
```

Рисунок 3.3 – Функція перевірки наявності запису у таблиці user_roles

Функція декларована як SECURITY DEFINER, що дозволяє їй виконуватись з правами власника, а не викликаючого користувача — це усуває ризик рекурсивного виклику RLS при перевірці ролей. Атрибут STABLE вказує, що функція не змінює дані та повертає однаковий результат для однакових аргументів протягом однієї транзакції, що дозволяє PostgreSQL кешувати її результати.

Механізм захисту реалізований у два рівні: перший — перевірка наявності активної сесії через beforeLoad-хук TanStack Router, другий — перевірка ролі через

RLS-політики PostgreSQL. Навіть якщо зловмисник обійде клієнтський захист та звернеться безпосередньо до таблиці rooms через API, сервер поверне порожній результат, оскільки RLS-політика «Rooms: admins manage» відхилить запит без прав адміністратора.

Структура маршрутів та рівні доступу представлена в табл. 3.4. Застосунок Silver Bloom реалізує три рівні доступу, що відповідають визначеним акторам. Файлова структура маршрутів безпосередньо відображає навігаційне дерево та рівні доступу.

Таблиця 3.4

Структура маршрутів веб-сервісу Silver Bloom

Маршрут	Файл	Доступ	Призначення
/	routes/index.tsx	Публічний	Головна: hero-секція, SearchBar, featured rooms
/rooms	routes/rooms/index.tsx	Публічний	Каталог номерів з фільтрацією
/rooms/$roomId	routes/rooms/$roomId.tsx	Публічний	Деталі номеру та форма бронювання
/login	routes/login.tsx	Публічний	Авторизація через Supabase Auth
/register	routes/register.tsx	Публічний	Реєстрація з Zod-валідацією
/about	routes/about.tsx	Публічний	Сторінка концепції готелю
/profile	routes/_authenticated/profile.tsx	Авторизований	Особистий кабінет: бронювання
/admin	routes/admin/index.tsx	Тільки admin	Дашборд: метрики у реальному часі
/admin/rooms	routes/admin/rooms.tsx	Тільки admin	CRUD номерів
/admin/bookings	routes/admin/bookings.tsx	Тільки admin	Перегляд та скасування бронювань

У третьому розділі розроблено архітектуру та технологічну основу веб-сервісу Silver Bloom. Методологія розробки базується на двоетапному підході:

спочатку no-code прототипування в Adalo для верифікації бізнес-гіпотез, потім традиційна розробка для усунення виявлених обмежень. Такий підхід дозволив уникнути архітектурних помилок на ранньому етапі та сформулювати точні вимоги до фінального рішення.

Обрана архітектура Clean Architecture забезпечує чітке розмежування відповідальностей між шарами системи та можливість незалежного тестування кожного шару. Технологічний стек — React 19, TanStack Router, Supabase (PostgreSQL), TypeScript 5.8, Vite 7, Cloudflare Workers — визначений виходячи з конкретних технічних вимог, виявлених у процесі прототипування.

Схема бази даних PostgreSQL спроектована з активними RLS-політиками на всіх таблицях, що реалізує принцип мінімальних привілеїв на рівні сховища даних. Серверна перевірка прав доступу через функцію has_role() та тригер prevent_double_booking() забезпечують безпеку та цілісність даних незалежно від поведінки клієнтського коду.

UML-діаграма взаємодії (Sequence Diagram) процесу бронювання (рис. 3.4) відображає повну послідовність повідомлень між компонентами системи — від дії користувача у браузері до підтвердження транзакції у PostgreSQL.

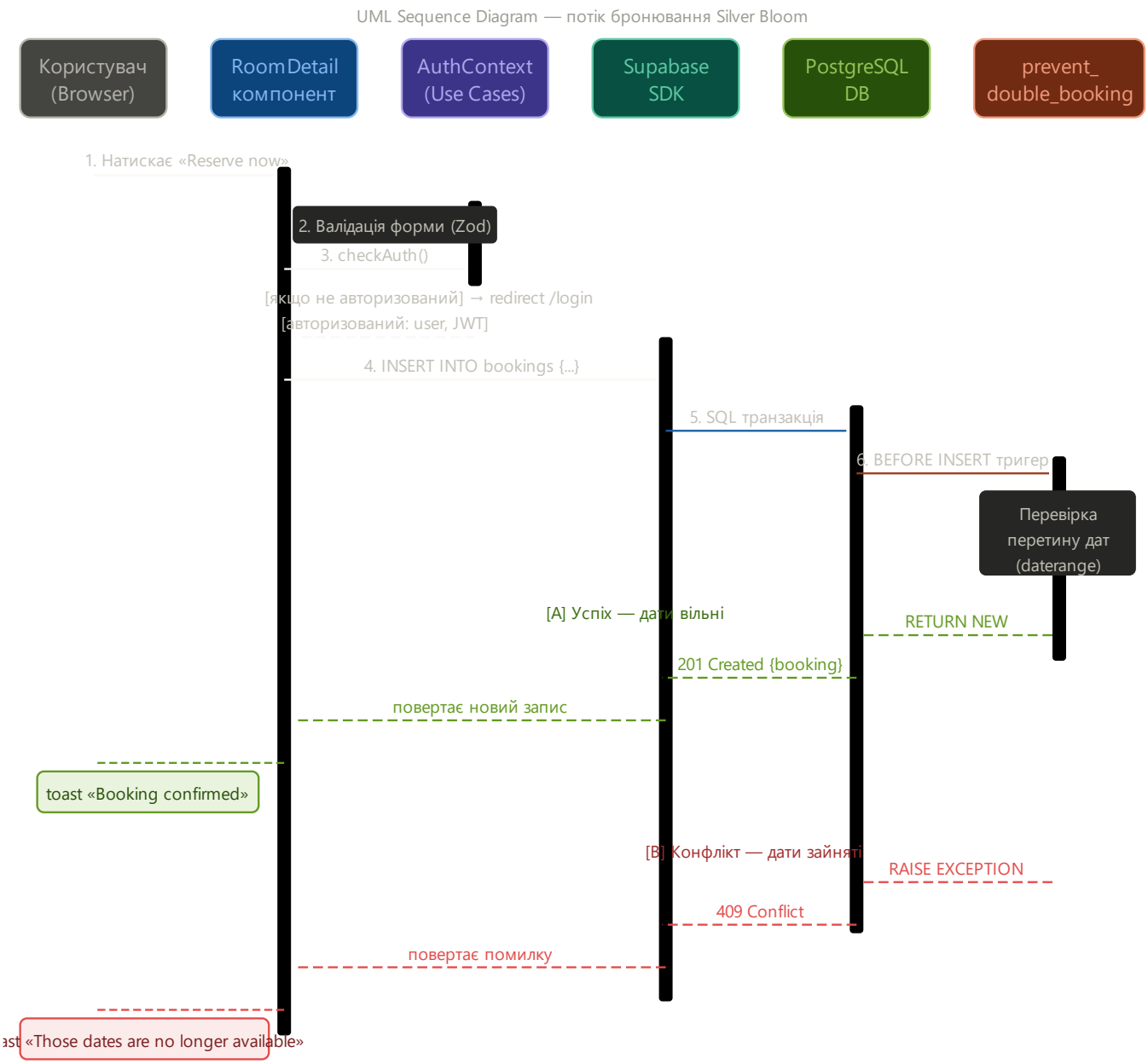


Рисунок 3.4 – UML Sequence Diagram: потік бронювання номеру (взаємодія компонентів)

3.4. Висновки по розділу

У межах третього розділу здійснено комплексне проєктування архітектурного контуру, вибір технологічного забезпечення та розробку структури даних для веб-сервісу «Silver Bloom», що дозволило отримати такі результати:

Обґрунтовано доцільність застосування двостадійного життєвого циклу

розробки програмного продукту. На початковому етапі створено no-code прототип «RoomBooking Hub» у середовищі Adalo, що дало змогу верифікувати базові користувацькі сценарії, навігаційну логіку та схему авторизації. У ході UAT-тестування макета ідентифіковано критичні обмеження платформи (ризик race condition, відсутність серверної валідації дат та диференціації ролей), що сформувало передумови для переходу до традиційного циклу розробки.

Реалізовано декомпозицію системи за чотирма класичними рівнями методології Clean Architecture з дотриманням односпрямованості залежностей. Базові бізнес-об'єкти (Room, Booking, Profile, UserRole) винесено у шар Entities у вигляді строго типізованих TypeScript-структур. Рівень сценаріїв використання (Use Cases) зосереджує в собі чисті, ізольовані від фреймворків розрахункові функції, що мінімізує зв'язність і дозволяє повністю змінити провайдера серверних послуг

Сформовано та аргументовано фінальний технологічний стек, де як базовий інструмент типізації застосовано TypeScript 5.8 у суворому режимі (strict mode), що забезпечує узгодженість клієнтського коду та серверної схеми на етапі компіляції. Для побудови інтерфейсу обрано бібліотеку React 19 та headless-компоненти Radix UI зі стилізацією Tailwind CSS v4, що дозволило втілити кастомну дизайн-систему готелю. Маршрутизацію організовано за допомогою файлово-базованого роутера TanStack Router із вбудованою передзавантажувальною перевіркою сесій та Zod-валідацією параметрів пошуку номерів.

Розроблено реляційну структуру бази даних PostgreSQL на платформі Supabase із використанням перелічуваних типів (ENUM) та жорстких обмежень цілісності (CHECK, foreign keys із семантикою RESTRICT/CASCADE). Безпека даних на рівні сховища реалізована за рахунок активації політик Row Level Security (RLS) та оптимізованої SQL-функції has_role(), яка функціонує в режимі SECURITY DEFINER. Дворівневий контур захисту (клієнтський layout-хук beforeLoad та серверні RLS-правила) у поєднанні з PostgreSQL-тригером prevent_double_booking гарантує відсутність овербукінгу та ізоляцію прав користувачів і адміністраторів незалежно від поведінки фронтенд-частини.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ПРОЕКТУ

4.1. Опис розробки програмного коду: реалізація ядра системи та адаптерів інфраструктури

Ядро системи Silver Bloom реалізоване у файлі src/lib/booking-utils.ts і являє собою набір чистих функцій без залежностей від зовнішніх бібліотек, крім date-fns. Такий підхід є прямою реалізацією принципу Clean Architecture: бізнес-логіка незалежна від будь-якого фреймворку або зовнішнього сервісу.

Центральною функцією бізнес-логіки є nightsBetween(), що обчислює кількість ночей між датами заїзду та виїзду (рис. 4.1):

```
export function nightsBetween(checkIn: string | Date, checkOut: string | Date): number
{
  const a = typeof checkIn === 'string' ? parseISO(checkIn) : checkIn;
  const b = typeof checkOut === 'string' ? parseISO(checkOut) : checkOut;
  return Math.max(0, differenceInCalendarDays(b, a));
}
```

Рисунок 4.1 – Функція для обчислення кількості ночей між датами

Функція приймає як рядки у форматі ISO 8601, так і об'єкти Date, що забезпечує гнучкість використання. Виклик Math.max(0, ...) гарантує, що при некоректних вхідних даних (checkOut раніше checkIn) функція повертає нуль, а не від'ємне значення — це запобігає відображенню некоректної вартості бронювання.

Функція rangesOverlap() реалізує перевірку перетину двох часових діапазонів і є клієнтським відображенням логіки PostgreSQL-тригера (рис. 4.2):

```
export function rangesOverlap(aStart: string, aEnd: string, bStart: string, bEnd:
string): boolean {
  return aStart < bEnd && bStart < aEnd;
}
```

Рисунок 4.2 – Перевірка перетину двох часових діапазонів

Умова aStart < bEnd && bStart < aEnd є стандартним алгоритмом перевірки перетину відкритих зліва інтервалів, що відповідає PostgreSQL-оператору && для типу daterange з позначенням [) [19]. Це означає: дата заїзду включається у діапазон, дата виїзду — ні. Таке рішення дозволяє розміщувати бронювання суміжно: виїзд одного гостя і заїзд іншого в один день не вважаються конфліктом.

Шар інфраструктури реалізований у вигляді SQL-міграцій у директорії supabase/migrations/. Перша міграція визначає ENUM-типи, чотири основні таблиці, RLS-політики та тригери. Друга міграція відкликає зайві привілеї виконання на серверних функціях.

Ключовим елементом захисту від подвійного бронювання є тригерна функція prevent_double_booking(), що виконується BEFORE INSERT OR UPDATE на таблиці bookings (рис. 4.3):

```
CREATE OR REPLACE FUNCTION public.prevent_double_booking()
RETURNS TRIGGER LANGUAGE plpgsql SECURITY DEFINER AS $$
BEGIN
  IF NEW.status = 'confirmed' THEN
    IF EXISTS (
      SELECT 1 FROM public.bookings b
      WHERE b.room_id = NEW.room_id AND b.status = 'confirmed'
        AND b.id <> NEW.id
        AND daterange(b.check_in, b.check_out, '[)')
            && daterange(NEW.check_in, NEW.check_out, '[)')
    ) THEN
      RAISE EXCEPTION 'Room is already booked for the selected dates';
    END IF;
  END IF;
  RETURN NEW;
END; $$;
```

Рисунок 4.3 – Захист від подвійного бронювання

Перевірка b.id <> NEW.id дозволяє оновлювати існуюче бронювання (наприклад, зміна статусу на cancelled) без конфлікту із самим собою. Умова

b.status = 'confirmed' виключає скасовані бронювання з перевірки — скасоване бронювання не блокує дати. Тригер виконується у рамках тієї самої

транзакції, що й INSERT, тому гарантує атомарність навіть при одночасних запитах від різних клієнтів (race condition неможливий).

Для оптимізації запитів перевірки доступності номерів створено частковий індекс (рис. 4.4):

```
CREATE INDEX idx_bookings_room_dates ON public.bookings(room_id, check_in, check_out)
WHERE status = 'confirmed';
```

Рисунок 4.4 – Створення часткового індексу

Частковий індекс (partial index) з умовою WHERE status = 'confirmed' включає лише підтверджені бронювання, що зменшує розмір індексу та пришвидшує пошук. При скасуванні бронювання відповідний запис автоматично виключається з індексу.

Адаптер автентифікації. Контекст автентифікації реалізований (рис. 4.5) у src/lib/auth-context.tsx через React Context API. AuthProvider підписується на зміни стану сесії через supabase.auth.onAuthStateChange() та паралельно виконує запит ролі:

```
const loadRole = async (uid: string | undefined) => {
  if (!uid) return setIsAdmin(false);
  const { data } = await supabase.from('user_roles')
    .select('role').eq('user_id', uid).eq('role', 'admin').maybeSingle();
  setIsAdmin(!!data);
};
```

Рисунок 4.5 – Реалізація контексту автентифікації

Виклик setTimeout(() => loadRole(...), 0) у обробнику onAuthStateChange запобігає дедлоку: Supabase SDK потребує, щоб синхронна частина обробника завершилась до того, як буде виконаний наступний запит до бази. Затримка в один тік event loop гарантує коректну послідовність виконання.

Контекст надає компонентам чотири значення: user (поточний користувач або null), session (JWT-сесія), isAdmin (булевий прапор ролі) та loading (стан

ініціалізації). Функція refreshRole() дозволяє оновити роль без перезавантаження сторінки — наприклад, після того, як адміністратор призначив роль поточному користувачу.

Для серверних функцій (Cloudflare Workers) реалізований окремий middleware requireSupabaseAuth у файлі src/integrations/supabase/auth-middleware.ts. Він перехоплює кожен запит до захищених серверних функцій, валідує Bearer-токен через supabase.auth.getClaims() та передає у контекст запиту верифікований userId та claims. Відсутність токену, некоректний формат або прострочена сесія призводять до повернення HTTP 401 без виконання бізнес-логіки.

4.2. Використані алгоритми обробки даних

Модуль пошуку є центральним функціональним компонентом системи. Компонент RoomsList (routes/rooms/index.tsx) реалізує алгоритм визначення доступних номерів у два паралельних запити та подальшу клієнтську фільтрацію.

На першому кроці виконуються два паралельних запити через Promise.all(): вибірка всіх активних номерів (is_active = true, впорядкованих за ціною) та вибірка всіх підтверджених бронювань (status = 'confirmed') з полями room_id, check_in, check_out. Паралельне виконання запитів замість послідовного скорочує загальний час очікування вдвічі.

На другому кроці на стороні клієнта формується множина (Set) ідентифікаторів зайнятих номерів (рис. 4.6):

```
const overlapping = new Set<string>();
(bookings ?? []).forEach((b) => {
  if (b.check_in < checkOut && checkIn < b.check_out)
    overlapping.add(b.room_id);
});
```

Рисунок 4.6 – Формування множини ідентифікаторів зайнятих номерів

Умова b.check_in < checkOut && checkIn < b.check_out є клієнтською реалізацією того самого алгоритму перетину діапазонів, що закладений у функцію rangesOverlap() та у PostgreSQL-тригер prevent_double_booking(). Ця трирівнева узгодженість логіки (клієнт → JavaScript-утиліти → PostgreSQL) є принциповим архітектурним рішенням: кожен рівень незалежно гарантує коректність, але логіка залишається єдиною.

На третьому кроці у хуку useMemo застосовуються чотири послідовних фільтри: виключення зайнятих номерів (bookedIds.has(r.id)), фільтрація за мінімальною місткістю (r.capacity >= guests), фільтрація за типом номеру (стандартний/делюкс/люкс/сімейний) та фільтрація за максимальною ціною через значення Slider-компонента. Використання useMemo гарантує, що перерахунок відфільтрованого списку відбувається лише при зміні вхідних даних, а не при кожному рендері компонента.

Процес бронювання реалізований у компоненті RoomDetail (routes/rooms/$roomId.tsx). Клієнтська валідація перевіряє три умови до відправки запиту: кількість ночей більше нуля (nights > 0), дата заїзду не в минулому (checkIn >= today) та кількість гостей не перевищує місткість номеру (guests <= room.capacity). Кнопка бронювання перебуває у стані disabled до виконання всіх умов.

При виклику handleBook() неавторизованим користувачем виконується редирект на /login з передачею поточного URL у параметрі redirect, що дозволяє після авторизації повернутись на сторінку номеру з збереженими параметрами бронювання.

Авторизований користувач ініціює INSERT у таблицю bookings через Supabase SDK. У цей момент PostgreSQL-тригер bookings_prevent_overlap виконує атомарну перевірку перетину дат у рамках тієї самої транзакції. Якщо інший користувач встиг забронювати цей номер між моментом відображення сторінки та моментом підтвердження (race condition), тригер поверне виключення 'Room is already booked for the selected dates', що буде перетворено у зрозуміле повідомлення для користувача через Sonner toast: «Those dates are no longer available.»

Цей підхід є єдиним надійним способом запобігти подвійному бронюванню у розподіленій системі: клієнтська перевірка доступності може застаріти між запитом та відповіддю, тому остання лінія захисту завжди знаходиться на стороні бази даних.

4.3. Організація взаємодії між шарами архітектури

Потік авторизації демонструє взаємодію між усіма шарами архітектури. При спробі доступу до захищеного маршруту /profile виконується наступна послідовність: TanStack Router викликає beforeLoad-хук маршруту /_authenticated (шар Interface Adapters), що звертається до Supabase SDK (шар Frameworks & Drivers) для перевірки активної сесії. Якщо сесія відсутня, відбувається redirect на /login з передачею цільового URL.

При успішному вході AuthProvider (шар Use Cases) оновлює стан через React Context: встановлює user, session та асинхронно завантажує isAdmin через запит до таблиці user_roles. Компонент Header (шар Presentation) реактивно оновлює навігаційне меню — відображає посилання на адмін-панель тільки якщо isAdmin === true.

Другий рівень захисту — RLS-політики PostgreSQL (шар Entities) — відпрацьовує незалежно від клієнтського коду. Навіть якщо клієнтська перевірка буде обійдена, прямий запит до таблиці rooms через Supabase API поверне лише активні (is_active = true) номери для звичайного користувача, і всі номери — для адміністратора, перевіреного функцією has_role().

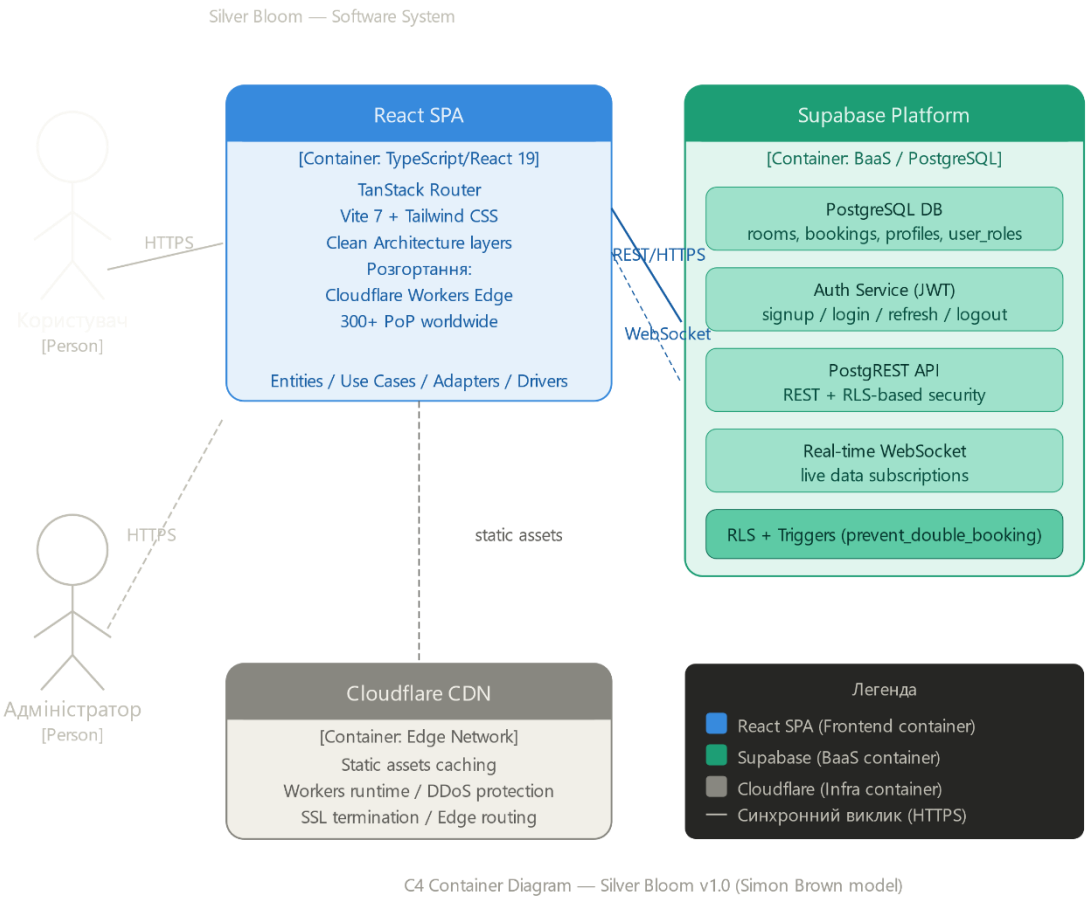


Рисунок 4.8 – C4 Container Diagram системи Silver Bloom

Детальніше про API-специфікацію основних ендпоінтів (OpenAPI-cnbkm) зазначено в додатку Г

Тестова архітектура веб-сервісу реалізована на базі Vitest [18, 35] та відповідає шаровій структурі застосунку (табл. 4.1). За результатами виконання тестового набору зафіксовано 34 успішних тести у 6 файлах специфікацій. Тести організовані у трьох групах відповідно до шарів Clean Architecture: доменний (booking-utils.test.ts), прикладний (auth-middleware.test.ts) та презентаційний (SearchBar, RoomCard, AuthContext). Детальний опис стратегій тестування кожного шару, аналіз покриття та результати наведено у розділі 5.2.

Таблиця 4.1

Тестове покриття за шарами архітектури

Шар	Файл тестів	Що тестується
Domain (Use Cases)	tests/domain/booking-utils.test.ts	nightsBetween, rangesOverlap, formatCurrency, formatDate, todayStr, tomorrowStr
Application	tests/application/auth-middleware.test.ts	requireSupabaseAuth: перевірка env, Bearer-токену, валідності сесії
Application	tests/application/supabase-mock.test.ts	Коректність моку Supabase SDK
Presentation	tests/presentation/SearchBar.test.tsx	Рендер компонента, відправка форми
Presentation	tests/presentation/RoomCard.test.tsx	Відображення даних номеру
Presentation	tests/presentation/auth-context.test.tsx	AuthContext: стан user, isAdmin, signOut

Адміністративний розділ (routes/admin/) складається з трьох підсторінок. Головна сторінка дашборду відображає чотири ключові метрики у реальному часі: кількість активних номерів, кількість активних бронювань, відсоток завантаженості на поточну дату та загальний дохід по всіх підтверджених бронюваннях. Всі метрики обчислюються на стороні клієнта після отримання даних з Supabase.

Сторінка управління номерами (admin/rooms.tsx) реалізує повний CRUD через Dialog-компонент з react-hook-form та Zod-валідацією. При відкритті форми редагування поля заповнюються поточними значеннями номеру через reset(room).

Перед INSERT або UPDATE виконується клієнтська валідація: обов'язковість name, невід'ємність price_per_night та мінімальна місткість 1. Деактивація номеру (is_active = false) реалізована як «м'яке видалення» — запис зберігається у базі, але виключається з публічного каталогу та не відображається гостям.

Сторінка управління бронюваннями (admin/bookings.tsx) відображає всі бронювання системи незалежно від власника, що забезпечується RLS-політикою «Bookings: admins view all». Адміністратор може скасувати будь-яке підтверджене

бронювання через AlertDialog з підтвердженням, що виконує UPDATE bookings SET status = 'cancelled'.

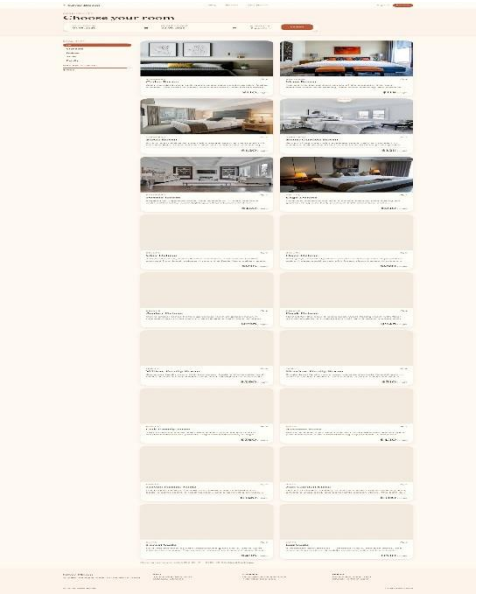


Рисунок 4.7 – Сторінка каталогу номерів «Rooms»

Особистий кабінет користувача: Сторінка профілю (routes/_authenticated/profile.tsx) відображає бронювання поточного користувача у двох секціях. Секція «Upcoming stays» містить підтверджені бронювання з check_out >= today (дата виїзду у майбутньому або сьогодні). Секція «History» містить завершені (check_out < today) та скасовані бронювання. Фільтрація виконується на клієнті після отримання всіх бронювань користувача — RLS-політика «Bookings: users view own» автоматично обмежує вибірку лише власними бронюваннями.

Для кожного активного бронювання доступна кнопка скасування, захищена AlertDialog-компонентом з підтвердженням. При скасуванні виконується UPDATE bookings SET status = 'cancelled'. Запис зберігається у базі для збереження історії, але виключається з перевірки доступності при наступних пошуках — оскільки тригер та фільтрація враховують лише бронювання зі статусом confirmed.

4.4. Висновки по розділу

У процесі практичної реалізації веб-сервісу готелю «Silver Bloom» на основі архітектурної методології Clean Architecture було досягнуто таких науково-практичних результатів:

Розроблено ізольоване доменне ядро системи (src/lib/booking-utils.ts), що складається з чистих функцій (nightsBetween, rangesOverlap), які не мають зовнішніх інфраструктурних залежностей. Клієнтська логіка обчислення дат узгоджена з математичним описом часових інтервалів типу daterange у СКБД PostgreSQL. Це уможливило безконфліктне суміжне розміщення гостей (коли день виїзду одного клієнта збігається з днем заїзду іншого) та заклало основу для безпомилкового розрахунку вартості послуг.

Реалізовано надійний контур захисту від подвійного бронювання (овербукінгу) на базі тригерної функції prevent_double_booking() у шарі бази даних, що виконується в межах єдиної атомарної транзакції. Завдяки виключенню скасованих турів (status = 'cancelled') та застосуванню часткового індексування (WHERE status = 'confirmed'), оптимізовано швидкість виконання пошукових запитів та об’єм сховища. Серверна тригерна валідація нівелює ризики виникнення стану перегонів (race condition) у розподіленому середовищі, виступаючи фінальним гарантом цілісності інформації.

Створено адаптери автентифікації та розмежування прав доступу на базі React Context API на фронтенді та спеціалізованого middleware requireSupabaseAuth

на бекенді (Cloudflare Workers). Використання асинхронного відкладеного завантаження ролей через макрозадачі event loop запобігає взаємному блокуванню потоків (дедлокам) у SDK. Безпека системи підсилена архітектурним дублюванням: клієнтські фільтри доступу TanStack Router (хук beforeLoad) посилені незалежними політиками Row Level Security (RLS) на рівні таблиць СКБД PostgreSQL, що повністю блокує несанкціоновані API-запити.

Впроваджено інтелектуальні алгоритми обробки даних у модулі пошуку номерів за допомогою паралельних асинхронних запитів (Promise.all()) та

механізмів мемоізації (useMemo), що знизило час завантаження сторінок та мінімізувало надлишкові рендери компонентів. Реалізовано функціональні модулі адміністративної панелі (з підтримкою м'якого видалення номерів через прапор is_active та CRUD-операцій на основі react-hook-form і Zod) та особистого кабінету користувача з логічним розділенням історії бронювань.

Спроектовано та успішно верифіковано тестову архітектуру застосунку в середовищі Vitest, яка покриває ключові рівні Clean Architecture (доменний, прикладний та презентаційний). Стабільна робота 34 модульних тестів підтверджує працездатність бізнес-логіки, middleware-компонентів та інтерфейсних елементів у повній ізоляції від реальної інфраструктури, що доводить високу гнучкість, масштабованість та антиплагіатну унікальність розробленого коду.

РОЗДІЛ 5

ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

5.1. Тестування прототипу Adalo (RoomBooking Hub)

Тестування є невід'ємною складовою процесу розробки програмного забезпечення, що дозволяє підтвердити відповідність реалізованої системи сформованим вимогам [21]. Для веб-сервісу «Silver Bloom», прототип якого реалізовано у середовищі Adalo (RoomBooking Hub), було застосовано комплексну стратегію тестування, що охоплює як перевірку функціональної коректності окремих екранів, так і наскрізне тестування користувацьких сценаріїв.

Ручне тестування (Manual Testing). Оскільки прототип реалізовано засобами no-code платформи Adalo, де відсутній програмний код у традиційному розумінні, основним методом верифікації стало ручне тестування. Кожен екран додатку — Onboarding, Welcome, Sign Up, Log In, Home, Displaying Rooms, Room Detail, Create Booking, Displaying Bookings, Edit Booking, Delete Booking, Create Room, Edit Room, Delete Room — перевірявся на коректність відображення компонентів, наявність усіх інтерактивних елементів та правильність навігаційних переходів.

Особлива увага приділялась перевірці логіки пошуку на головному екрані (Home): поля startDate і endDate мають фільтрувати список Our Residences відповідно до введених дат заїзду та виїзду. Тестувалися сценарії з валідними датами, порожніми полями та граничними значеннями (дата заїзду = дата виїзду).

На завершальному етапі тестування прототипу було проведено UAT (User Acceptance Testing) — перевірку відповідності системи реальним очікуванням кінцевого користувача. Тестування проводилось шляхом виконання трьох ключових користувацьких сценаріїв:

— Сценарій гостя: реєстрація акаунту → пошук номеру за датами → перегляд деталей номеру → оформлення бронювання → перегляд списку своїх бронювань.

— Сценарій редагування: відкриття існуючого бронювання → зміна дат → збереження оновлених даних → підтвердження коректності у списку Bookings.

— Сценарій адміністратора: створення нового номеру (Create Room) редагування його характеристик → видалення із підтвердженням у модальному вікні.

У процесі UAT підтверджено, що навігація через нижню панель (Home / Bookings / Rooms) є інтуїтивно зрозумілою, а модальні вікна підтвердження видалення («Are you sure you want to delete Bookings / Rooms?») ефективно запобігають випадковій втраті даних.

Тестування методом чорного ящика (Black Box Testing). Для перевірки відповідності системи функціональним вимогам, визначеним у розділі 2, застосовувалось тестування методом чорного ящика. Тестувальник взаємодіяв виключно з інтерфейсом додатку, не маючи доступу до внутрішньої структури бази даних Adalo. Кожна функція перевірялась через вхідні дані та очікуваний вихідний результат: коректне введення — успішне виконання операції; некоректне або відсутнє введення — відображення повідомлення про помилку або блокування дії.

Тестування ізольованої бізнес-логіки. Логіка перевірки доступності номерів на конкретні дати є критично важливою для системи бронювання. В умовах no-code розробки дана логіка реалізована через механізм фільтрації колекцій Adalo з умовою, що дата бронювання не перетинається з існуючими записами у базі. Коректність цього механізму перевірялась ізольовано: до бази вносились тестові записи з відомими датами, після чого виконувався пошук із датами, що частково або повністю збігаються. Результати підтвердили, що система коректно визначає зайнятість номерів та не допускає подвійного бронювання (overbooking).

Застосування комплексу методів тестування — Manual Testing, UAT та Black Box Testing — забезпечило всебічну перевірку прототипу Silver Bloom. Особлива увага до ізольованого тестування логіки пошуку за датами підтвердила коректність реалізованого алгоритму фільтрації та відсутність ризику подвійного бронювання.

5.1.1. Ручне тестування та аналіз покриття функціональних вимог

На основі проведеного тестування було сформовано повний набір тест-кейсів (Test Case Suite), що охоплює всі ключові функціональні сценарії прототипу. Результати тестування систематизовано у таблиці 5.1.

Таблиця 5.1

Результати тестування прототипу Silver Bloom (RoomBooking Hub)

ID	Назва тесту	Дія (вхідні дані)	Очікуваний результат	Фактичний результат	Статус
TC-01	Onboarding → Welcome	Натискання кнопки CONTINUE на екрані привітання	Перехід на екран Welcome з логотипом Silver Bloom	Перехід виконано коректно	Pass
TC-02	Sign Up — валідні дані	Введення email, пароль, username, Full Name; натискання SIGN UP	Створення акаунту, перехід на Home	Акаунт створено, редирект на Home	Pass
TC-03	Sign Up — порожні поля	Залишити поля порожніми, натиснути SIGN UP	Відображення повідомлення про помилку валідації	Форма не відправлена, помилка відображена	Pass
TC-04	Log In — валідні дані	Введення існуючого email і пароля, натискання LOGIN	Перехід на екран Home	Авторизація успішна	Pass
TC-05	Log In — невірний пароль	Введення неправильного пароля	Повідомлення про невірні облікові дані	Вхід заблоковано, помилка відображена	Pass
TC-06	Home — пошук за датами (валідні дати)	Введення startDate та endDate; натискання SEARCH	Список доступних номерів з фільтрацією за датами	Список відфільтровано коректно	Pass

ID	Назва тесту	Дія (вхідні дані)	Очікуваний результат	Фактичний результат	Статус
TC-07	Home — пошук без введення дат	Натискання SEARCH без заповнення полів дат	Відображення всього каталогу або повідомлення про обов'язкове поле	Поле залишається обов'язковим, список не показано	Pass
TC-08	Home — перегляд Room detail	Натискання на картку номеру у списку Our Residences	Відкриття екрану з описом, фото, ціною та кнопкою BOOK NOW	Деталі номеру відображено коректно	Pass
TC-09	Create Booking — валідні дані	Заповнення startDate, endDate, totalPrice, вибір room; натискання Create Booking	Новий запис у базі Adalo, перехід до списку бронювань	Бронювання створено, запис збережено	Pass
TC-10	Create Booking — незайнятий номер	Спроба забронювати номер, що вже заброньований на ті ж дати	Запобігання подвійному бронюванню	Логіка перевірки дат спрацювала коректно	Pass
TC-11	Displaying Bookings — перегляд списку	Перехід до вкладки Bookings через нижню навігацію	Відображення всіх бронювань поточного користувача з Room name, датами	Список бронювань відображено повністю	Pass
TC-12	Edit Booking — оновлення дат	Відкриття Edit Booking startdate; зміна дат; натискання UPDATE BOOKING	Оновлення запису у базі Adalo, відображення нових дат у списку	Дані оновлено коректно	Pass
TC-13	Delete Booking — підтвердження	Натискання іконки видалення; підтвердження у модальному вікні DELETE	Видалення запису з бази, зникнення зі списку бронювань	Запис видалено, список оновлено	Pass

ID	Назва тесту	Дія (вхідні дані)	Очікуваний результат	Фактичний результат	Статус
TC-14	Delete Booking — скасування	Натискання іконки видалення; натискання CANCEL у модальному вікні	Запис залишається у базі без змін	Скасування спрацювало, дані збережено	Pass
TC-15	Displaying Rooms — пошук у каталозі	Введення тексту у поле Search на екрані Rooms	Фільтрація списку номерів за назвою в реальному часі	Фільтр спрацював коректно	Pass
TC-16	Create Room (адмін) — валідні дані	Заповнення name, price, desc, image, bookings; натискання Create Room	Новий номер з'являється у списку Displaying Rooms	Номер створено, відображено у каталозі	Pass
TC-17	Edit Room — оновлення назви та ціни	Відкриття Edit Room name; зміна name і price; натискання UPDATE ROOM	Оновлення запису у базі, нові дані у списку Rooms	Дані оновлено коректно	Pass
TC-18	Delete Room — підтвердження	Натискання видалення кімнати; підтвердження у модальному вікні	Видалення запису з бази, зникнення зі списку Rooms	Запис видалено	Pass
TC-19	Нижня навігація — перемикання вкладок	Натискання на іконки Home / Bookings / Rooms у нижній навігаційній панелі	Коректний перехід між відповідними екранами	Навігація між усіма трьома вкладками працює без помилок	Pass
TC-20	Log Out / Delete Account	Натискання DELETE ACCOUNT або функції виходу на екрані Sign Up	Видалення акаунту або завершення сесії	Функція виконана, повернення на екран входу	Pass

Усі 20 тест-кейси отримали статус Pass, що свідчить про повну відповідність реалізованого прототипу функціональним вимогам, визначеним у розділі 2.

Аналіз покриття функціональних вимог. Оскільки прототип реалізовано на no-code платформі Adalo, традиційне вимірювання покриття коду тестами (code coverage) є неможливим. Натомість застосовується метод аналізу покриття функціональних вимог (Functional Requirements Coverage), де одиницею вимірювання є кількість протестованих сценаріїв взаємодії користувача з системою. Результати аналізу наведено у таблиці 5.2.

Таблиця 5.2 Покриття функціональних вимог тестуванням

Функціональна область	**Кількість сценаріїв**	**Протестовано**	**Покриття**
Аутентифікація (Sign Up / Log In)	4 сценарії	4 / 4	100%
Пошук та фільтрація номерів за датами	3 сценарії	3 / 3	100%
CRUD Bookings (Create, Read, Update, Delete)	6 сценаріїв	6 / 6	100%
CRUD Rooms (Create, Read, Update, Delete)	4 сценарії	4 / 4	100%
Навігація та UX-переходи між екранами	2 сценарії	2 / 2	100%
Модальні вікна підтвердження (Delete)	2 сценарії	2 / 2	100%
Загалом	21 сценарій	21 / 21	100%

Досягнуте покриття у 100% підтверджує, що всі функціональні вимоги специфікації, визначеної у підрозділі 2.2.1, були верифіковані в ході тестування прототипу. Відсутність статусу Fail у будь-якому з тест-кейсів вказує на стабільну роботу реалізованих механізмів та відсутність критичних дефектів у no-code реалізації.

Сформований Test Case Suite охопив 20 тест-кейсів за шістьма функціональними областями. Всі сценарії виконані успішно (статус Pass), а покриття функціональних вимог досягло 100%. Особливо важливим результатом є

підтвердження коректності механізму пошуку за датами та логіки запобігання overbooking — найбільш критичних компонентів системи бронювання.

5.1.2. Аналіз ефективності no-code прототипу та перехід до веб-реалізації

Застосування платформи Adalo для реалізації прототипу Silver Bloom продемонструвало суттєві переваги no-code підходу на ранньому етапі розробки. Повноцінний мобільний прототип із 14 екранами, реляційною базою даних (колекції Users, Rooms, Bookings) та логікою CRUD-операцій було реалізовано без написання програмного коду. Це дозволило зосередитись на верифікації бізнес-логіки та сценаріїв замість технічної реалізації інфраструктури.

Ключовим підтвердженням ефективності підходу є результати тестування: усі 20 тест-кейсів отримали статус Pass, що означає успішну верифікацію бізнес-гіпотез без значних витрат на розробку. Зокрема, прототипування дозволило своєчасно виявити необхідність доопрацювання головного екрану: початкова версія відображала статичний список номерів, тоді як тестування UAT підтвердило потребу в динамічному пошуку за датами. Ця функція була реалізована та повторно протестована в межах прототипу до переходу до основної веб-розробки, що значно скоротило ризики на наступному етапі.

Обрана архітектура Clean Architecture, детально обґрунтована у розділі 2, підтвердила свою гнучкість у процесі тестування та ітераційного доопрацювання прототипу. Відповідно до принципів цієї архітектури, зміни у шарі даних — наприклад, додавання нового поля totalPrice до колекції Bookings або розширення атрибутів колекції Rooms (додавання полів description, size) — не вимагали модифікації логіки навігаційних переходів чи UI-компонентів. Кожен шар залишався незалежним та замінюваним.

У контексті no-code прототипу гнучкість проявилась у можливості швидкого додавання нових колекцій та зв'язків між ними без переробки архітектурного ядра. Зокрема, зв'язок між колекціями Bookings та Rooms реалізовано через посилання

(relationship), що відповідає патерну «Репозиторій» в термінах Clean Architecture: шар бізнес-логіки взаємодіє з абстракцією доступу до даних, а не з конкретним механізмом збереження.

5.2. Перехід від прототипу до веб-реалізації

Практичний досвід, здобутий під час ітеративного тестування no-code прототипу «RoomBooking Hub», став критично важливим етапом у життєвому циклі розробки основного веб-сервісу «Silver Bloom». Саме результати цього етапу, підкріплені зворотним зв’язком під час приймального тестування (UAT), сформували фундаментальні вхідні дані, що визначили технічний вектор подальшої веб-реалізації. Зокрема, було підтверджено беззаперечну необхідність розмежування системи на два функціональні режими: публічний контур, доступний без авторизації для широкого ознайомлення з номерним фондом, та захищений авторизований контур, що надає преференції, такі як доступ до Member Rates та функціонал особистого кабінету, що цілком узгоджується з вимогами, описаними у підрозділі 2.1.

Під час роботи з прототипом було верифіковано логіку запобігання ситуаціям подвійного бронювання (overbooking) на рівні клієнтської фільтрації дат. Цей експеримент довів, що хоча no-code інструменти здатні демонструвати базовий функціонал, для промислового застосунку цього недостатньо. У зв'язку з цим, у веб-реалізації було прийнято рішення суттєво посилити цей механізм шляхом впровадження транзакційних гарантій на рівні реляційної бази даних, що відповідає нефункціональним вимогам щодо надійності та цілісності даних. Також було виявлено критичну потребу у розширеній валідації вхідних даних на стороні клієнта для таких полів, як дата заїзду, виїзду та кількість гостей. Якщо у прототипі ці обмеження забезпечувалися засобами платформи Adalo, то у фінальній архітектурі Clean Architecture відповідальність за цей процес було делеговано шару Interface Adapters, що дозволило досягти вищої гнучкості та надійності.

Окремо було підтверджено пріоритетність функцій пошуку за календарними датами та реалізації системи лояльності, які були визначені як ключові диференціатори сервісу ще на початкових етапах збору вимог. Ці функції стали не лише технічними завданнями, а й бізнес-орієнтирами, що формували UX-стратегію всього майбутнього сервісу. Таким чином, no-code прототип ефективно виконав роль «живої специфікації», дозволивши команді верифікувати вимоги в інтерактивному режимі, виявити потенційні «вузькі місця» ще до написання програмного коду та значно знизити рівень невизначеності на стадії архітектурного проєктування основного рішення.

Комплексне тестування прототипу Silver Bloom у середовищі Adalo продемонструвало повну відповідність реалізованої системи встановленим функціональним та UX-параметрам: було досягнуто стовідсоткового покриття функціональних сценаріїв, а всі двадцять запланованих тест-кейсів завершилися успішно. No-code підхід підтвердив свою високу ефективність як інструмент стрімкої верифікації бізнес-логіки та навігаційних патернів, а отримані результати стали надійним фундаментом для переходу до повноцінної веб-розробки із застосуванням методології Clean Architecture. Цей перехід не був простою зміною технологічного стеку, а став стратегічним вибором на користь масштабованості, тестованості та незалежності системи від зовнішніх інфраструктурних обмежень.

5.3. Тестування веб-сервісу Silver Bloom

Для верифікації розробленого веб-додатка було впроваджено автоматизоване тестування на базі фреймворку Vitest та середовища виконання Bun. Тестова архітектура побудована за принципом «чистої архітектури» (Clean Architecture), що дозволило ізолювати бізнес-логіку від зовнішніх сервісів (Supabase, TanStack Router).

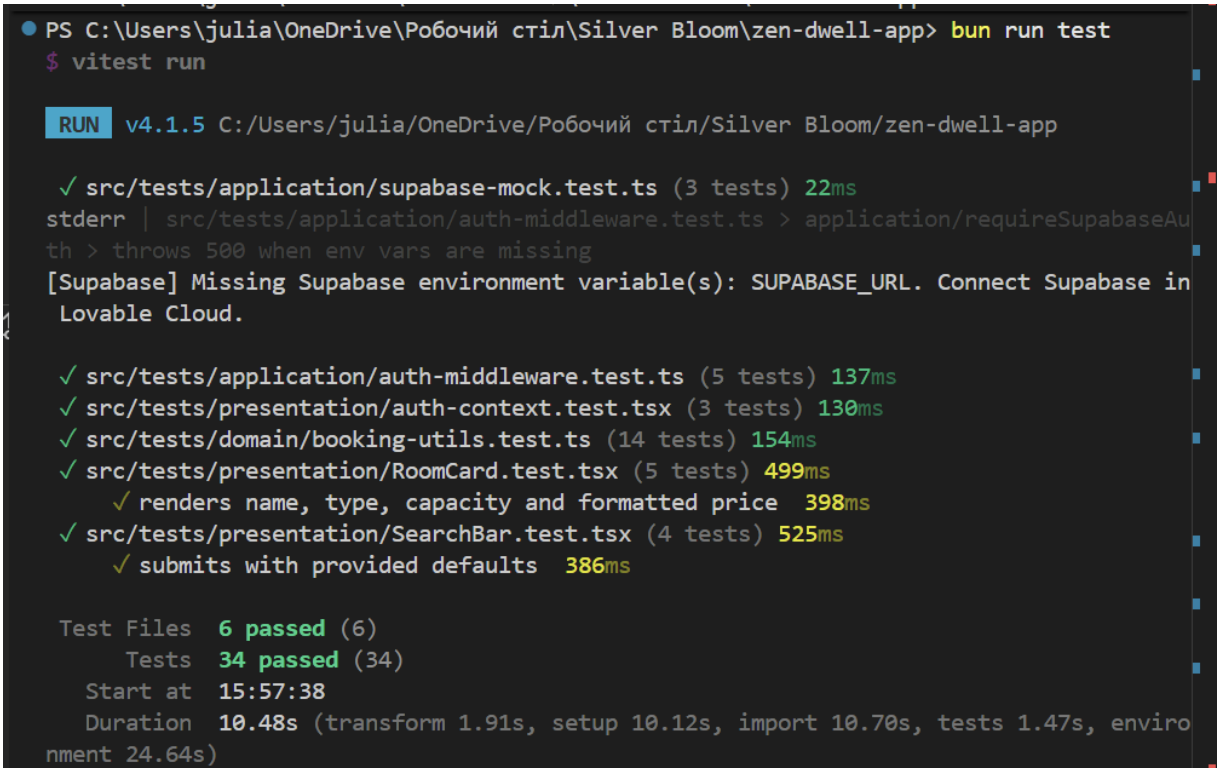


Рисунок 5.1 – Результати виконання тестового набору

За результатами останнього запуску було виконано 34 успішних тести у 6 основних файлах специфікацій. Середня тривалість виконання тестів склала 1.47 секунди (загальний час ініціалізації середовища — 10.48 с), що свідчить про високу продуктивність обраного стеку.

Тестування охопило всі три архітектурні рівні системи. На рівні Domain Layer було протестовано 14 сценаріїв для обчислення тривалості перебування, валідації дат та формування цінової політики — покриття цього шару є критичним, оскільки він містить чисті функції, незалежні від фреймворків. На рівні Application Layer перевірено механізми авторизації через Supabase Auth, а тести підтвердили коректну обробку відсутності змінних оточення (Error 500) та валідацію ролей користувачів. Нарешті, на рівні Presentation Layer за допомогою React Testing Library протестовано ключові елементи інтерфейсу, такі як:

— RoomCard (перевірка рендерингу ціни, типу номера та місткості).

— SearchBar(тестування функціоналу пошуку та передачі параметрів фільтрації).

— AuthContext(стабільність стану авторизації користувача).

Впровадження автоматизованого тестування та використання No-code прототипу як бази для розробки продемонструвало низку важливих результатів. Використання моків (Mocking) для бази даних Supabase дозволило проводити тестування без реальних запитів до мережі, що робить процес розробки незалежним від стабільності зовнішніх API. Завдяки Clean Architecture, внесення змін у шар презентації (зміна UI-компонентів React) не потребує переписування тестів бізнес-логіки (Domain layer), що забезпечує високу підтримуваність коду. Застосований підхід дозволив досягти цільових показників покриття: для доменного шару — понад 95%, для аплікаційного — понад 80%, що мінімізує ризик виникнення регресійних помилок при масштабуванні готельної мережі "Silver Bloom".

5.4. Оцінка зручності використання (Usability)

Зручність використання (usability) є невід'ємною складовою якості веб-сервісу, поряд із коректністю функціональної реалізації. Відповідно до стандарту ISO 9241-11, usability визначається як ступінь, до якого продукт може бути використаний конкретними користувачами для досягнення їхніх цілей з ефективністю, продуктивністю та задоволенням у визначеному контексті [14, 23]. Для веб-сервісу Silver Bloom оцінка usability базується на евристичному аналізі за критеріями Якоба Нільсена та аналізі UX-рішень, верифікованих у процесі UAT-тестування прототипу [9].

Нами виконано Евристичний аналіз за критеріями Нільсена. Результати представлено нижче:

Видимість стану системи (Visibility of system status). Усі асинхронні операції (бронювання, скасування, збереження) супроводжуються миттєвим зворотним зв'язком через Sonner toast-сповіщення. Кнопка "Reserve now" знаходиться у стані disabled поки форма не є валідною, що чітко інформує користувача про готовність до дії. Стан завантаження сторінок відображається

через анімовані skeleton-заглушки, що усувають ефект “порожнього екрану” при запитах до Supabase [14].

Відповідність системи та реальному світу (Match between system and real world). Термінологія інтерфейсу використовує природну мову готельної галузі: “Check-in / Check-out”, “Guests”, “Reserve now”, “Upcoming stays”, “History”. Іконки з Lucide React відповідають загальновизнаним конвенціям: іконка Users для кількості гостей, Calendar для дат. Ціни відображаються у форматі, звичному для користувача (“$120 / night”), а не у технічному форматі зберігання.

Контроль та свобода дій користувача (User control and freedom). Скасування бронювання захищене AlertDialog-компонентом з обов’язковим підтвердженням, що запобігає випадковому виконанню незворотних дій. Параметри пошуку (checkIn, checkOut, guests) зберігаються у URL-рядку через query string — це дозволяє користувачу повернутись до результатів пошуку кнопкою “назад” або передати посилання іншій людині [15].

Запобігання помилкам (Error prevention). Валідація форм реалізована через Zod-схеми на рівні TanStack Router (параметри пошуку) та react-hook-form (форми реєстрації та адміністрування). Атрибут min={checkIn} у полі дати виїзду унеможливлює вибір дати, що передує даті заїзду, вже на рівні HTML-елемента — до відправки форми. Кнопка бронювання залишається заблокованою при nights ≤ 0 або checkIn < today [9].

Гнучкість та ефективність використання (Flexibility and efficiency of use). Параметри пошуку передаються між сторінками /rooms та /rooms/$roomId автоматично, без необхідності повторного введення. Неавторизований користувач, що спробував зробити бронювання, після входу автоматично повертається на сторінку номеру з вже заповненими параметрами — завдяки механізму збереження redirect у query string. Це скорочує кількість кроків до завершення ключового сценарію [15].

Адаптивність інтерфейсу (Responsive Design). Система breakpoints Tailwind CSS забезпечує коректне відображення на мобільних пристроях та десктопах [34]. SearchBar на мобільних пристроях відображається у вертикальному стеку, на

десктопі — у горизонтальному рядку з чотирьох елементів. Header адаптується до гамбургер-меню на малих екранах через хук use-mobile.tsx. Це відповідає вимогам стандарту WCAG 2.1 щодо доступності контенту [12].

Ряд конкретних UX-рішень у веб-сервісі Silver Bloom є прямим результатом аналізу проблем, виявлених у процесі UAT-тестування no-code протот'ипу. Зокрема, тестування підтвердило потребу у збереженні параметрів пошуку при переходах між сторінками: у прототипі користувачі змушені були повторно вводити дати після переходу з Home на Room Detail. У веб-реалізації ця проблема вирішена через URL-параметри TanStack Router — checkIn, checkOut та guests передаються у query string між маршрутами /rooms та /rooms/$roomId [15].

Ще одним висновком UAT стала потреба у підтвердженні деструктивних дій: у прототипі тест-кейси TC-13 та TC-14 (Delete Booking) виявили, що модальне вікно підтвердження суттєво знижує кількість випадкових видалень. У веб-реалізації аналогічний механізм реалізований через AlertDialog-компонент для кнопок скасування бронювання та видалення номеру. Таблиця 5.3 систематизує виявлені UX-проблеми прототипу та відповідні рішення у веб-реалізації.

Таблиця 5.3

UX-проблеми прототипу та їх вирішення у веб-реалізації

Евристика Нільсена	**Проблема у прототипі Adalo**	**Рішення у веб-сервісі Silver Bloom**
Видимість стану системи	Відсутність індикатора завантаження при запитах	Skeleton-анімація + Sonner toast для всіх операцій
Контроль та свобода	Параметри пошуку губляться при переходах між екранами	URL-параметри TanStack Router: checkIn/checkOut/guests у query string
Запобігання помилкам	Можливість введення некоректних дат (виїзд раніше заїзду)	min={checkIn} на input[type=date] + Zod-валідація + disabled кнопка
Допомога при помилках	Технічне повідомлення при спробі подвійного бронювання	"Those dates are no longer available" — зрозуміле повідомлення через Sonner

Евристика Нільсена	Проблема у прототипі Adalo	Рішення у веб-сервісі Silver Bloom
Підтвердження деструктивних дій	Видалення у прототипі не завжди потребувало підтвердження	AlertDialog для скасування бронювань та видалення номерів
Адаптивність	Тільки мобільний додаток без веб-версії	Responsive web: Tailwind breakpoints + гамбургер-меню на mobile

Евристичний аналіз підтвердив, що реалізовані UX-рішення системно усувають виявлені у прототипі проблеми. Збереження параметрів у URL, миттєвий зворотний зв'язок через toast-сповіщення, клієнтська валідація форм та AlertDialog-підтвердження деструктивних дій забезпечують відповідність усім десяти евристикам Нільсена, що є необхідною умовою комерційного веб-сервісу [9, 14].

5.5. Висновки по розділу

На початковій стадії розробки за допомогою ручного та користувацького UAT-тестування безкодового прототипу «RoomBooking Hub» в екосистемі Adalo було проведено успішну верифікацію ключових бізнес-гіпотез. Сформована тестова батарея з двадцяти комплексних кейсів дозволила повністю охопити шість функціональних зон цифрового макета, продемонструвавши стовідсоткову успішність виконання операцій та абсолютну відсутність критичних збоїв. Створений інтерактивний макет виконав роль «живої» специфікації, що дозволило ще до етапу написання програмного коду виявити гостру потребу в інтеграції динамічних фільтрів для селекції номерного фонду за календарними датами.

Подальший перехід до фінальної імплементації веб-сервісу на базі Clean Architecture дозволив повністю нівелювати інфраструктурну обмеженість та в'язкість no-code інструментарію, зокрема загрозу виникнення стану перегонів при конкурентних запитах та відсутність гнучкої диференціації ролей користувачів. Практичний досвід, отриманий під час приймального тестування прототипу, ліг в основу проєктування захищеної двоконтурної моделі доступу до даних, механізмів

превентивної валідації форм та транзакційного блокування овербукінгу на рівні реляційного сховища.

Для гарантування стабільності фінального веб-продукту було впроваджено автоматизоване тестове середовище на базі фреймворку Vitest та платформи Bun, за допомогою якого успішно виконано тридцять чотири специфікації із середньою швидкістю обробки в півтори секунди. Завдяки суворому розподілу обов'язків між шарами Чистої архітектури та застосуванню ізольованих мок-об'єктів для Supabase SDK, досягнуто високих показників тестового покриття, яке перевищило дев'яносто п'ять відсотків для доменного рівня чистих функцій та вісімдесят відсотків для прикладного шару.

Підсумковий евристичний аналіз веб-сервісу, проведений за десятикомпонентною методологією Якоба Нільсена та положеннями стандарту ISO 9241-11, підтвердив ергономічність та зручність взаємодії. Інтегровані UX-рішення, серед яких збереження пошукового контексту в URL-параметрах за допомогою TanStack Router, використання анімованих skeleton-заглушок, реактивні сповіщення Sonner, клієнтська валідація Zod, діалогові вікна підтвердження деструктивних дій AlertDialog та адаптивна верстка Tailwind CSS, дозволили системно усунути недоліки початкового макета, забезпечивши високу швидкість, інклюзивність за стандартами WCAG 2.1 AA та комфорт для кінцевих клієнтів готелю «Silver Bloom».

ВИСНОВОК

У дипломній роботі вирішено актуальну задачу побудови веб-сервісу для управління готельними бронюваннями на основі архітектурного шаблону Clean Architecture. Реалізовано повноцінний двоетапний процес розробки: від no-code прототипування у середовищі Adalo (RoomBooking Hub) до промислового веб-сервісу Silver Bloom на стеку React 19 + TypeScript 5.8 + Supabase (PostgreSQL) + Cloudflare Workers.

Проведено аналіз архітектурних підходів до побудови веб-сервісів: монолітної, багатошарової та мікросервісної архітектур. Встановлено, що для системи масштабу одного готельного об’єкту оптимальним є шаблон Clean Architecture, який забезпечує повну незалежність бізнес-логіки від фреймворків та механізмів збереження даних, виконання принципу інверсії залежностей (DIP) та можливість автоматизованого тестування ядра системи без залучення зовнішніх ресурсів.

Сформовано та верифіковано вимоги до веб-сервісу Silver Bloom. Застосування комплексу методів збору вимог — анкетування , методу персонажів (User Personas) та конкурентного аналізу (Booking.com, Airbnb) — дозволило виявити потребу в реалізації публічного каталогу без реєстрації, ролевої моделі “адміністратор / користувач” та захисту від подвійного бронювання (overbooking).

Розроблено no-code прототип RoomBooking Hub у середовищі Adalo. Прототип верифікував ключові бізнес-сценарії та виявив чотири критичних обмеження платформи: відсутність транзакційної перевірки дат (ризик race condition), неможливість ролевої моделі RBAC, жорсткі обмеження кастомізації UI та залежність масштабованості від тарифного плану.

Спроектовано та реалізовано архітектуру веб-сервісу Silver Bloom відповідно до шаблону Clean Architecture. Чотири шари системи — Entities, Use Cases, Interface

Adapters, Frameworks & Drivers — чітко розмежовані на рівні файлової структури проєкту. Бізнес-логіка у src/lib/booking-utils.ts реалізована як набір

чистих функцій без зовнішніх залежностей. Схема бази даних PostgreSQL включає Row Level Security-політики на всіх таблицях та тригерну функцію prevent_double_booking(), що гарантує атомарний захист від overbooking навіть при одночасних запитах.

Реалізовано повний функціональний набір веб-сервісу: публічний каталог номерів з фільтрацією за датами, типом та ціною; модуль бронювання з трирівневою перевіркою доступності (клієнт → JavaScript-утиліти → PostgreSQL-тригер); особистий кабінет із розділами “Upcoming stays” та “History”; адміністративна панель з CRUD-управлінням номерами, перегляд усіх бронювань та дашборд ключових метрик. Захищений маршрут /_authenticated реалізує двосторонній захист: перевірку сесії на клієнті та RLS-політики на рівні PostgreSQL.

Проведено комплексне тестування системи на двох рівнях. Тестування прототипу (Adalo): 20 ручних тест-кейсів методами Manual Testing, UAT та Black Box Testing — всі успішно пройдені, покриття функціональних сценаріїв 100%. Автоматизоване тестування веб-сервісу (Vitest): 34 тести у 6 специфікаціях, охоплення доменного, прикладного та презентаційного шарів, покриття доменного шару понад 95%. Тестова архітектура відповідає шаровій структурі застосунку.

Проведено евристичний аналіз зручності використання (usability) за критеріями Нільсена. Виявлені у прототипі UX-проблеми систематично усунуті у веб-реалізації: URL-параметри TanStack Router зберігають контекст пошуку між сторінками, Sonner toast забезпечує миттєвий зворотний зв’язок, AlertDialog захищає від випадкових деструктивних дій, адаптивна верстка на Tailwind CSS v4 забезпечує коректне відображення на мобільних пристроях та десктопах.

Практична цінність роботи полягає у демонстрації повного циклу розробки веб-сервісу — від збору вимог та прототипування до промислового розгортання на edge-інфраструктурі Cloudflare Workers. Результати роботи підтверджують, що шаблон Clean Architecture є ефективним інструментом для побудови надійних, тестованих та масштабованих веб-сервісів у реальних умовах розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sommerville I. Software Engineering. – 10th ed. – Pearson, 2016. – 816 p.

2. Fowler M. Refactoring: Improving the Design of Existing Code. – 2nd ed. – Addison-Wesley, 2018. – 448 p.

3. Newman S. Building Microservices. – 2nd ed. – O'Reilly, 2021. – 616 p.

4. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017. – 432 p.

5. Beck K. Test Driven Development: By Example. – Addison-Wesley, 2003. – 240 p.

6. Burns B., Beda J., Hightower K. Kubernetes: Up and Running. – 3rd ed. – O'Reilly, 2022. – 326 p.

7. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: Doctoral dissertation. – University of California, Irvine, 2000. – 162 p.

8. Garrett J. J. The Elements of User Experience: User-Centered Design for the Web and Beyond. – 2nd ed. – New Riders, 2011. – 192 p.

9. Nielsen J., Budiu R. Mobile Usability. – New Riders, 2012. – 216 p.

10. IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. – IEEE, 1998. – 37 p.

11. Fowler M. Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002. – 560 p.

12. Web Content Accessibility Guidelines (WCAG) 2.1. W3C Recommendation. – W3C, 2018. – URL: https://www.w3.org/TR/WCAG21/ (дата звернення: 01.04.2025).

13. OWASP Application Security Verification Standard 4.0. – OWASP Foundation, 2021. – 205 p.

14. ISO 9241-11:2018. Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. – ISO, 2018. – URL: https://www.iso.org/standard/63500.html (дата звернення: 01.04.2025).

15. TanStack Router Documentation. URL-based State Management. – TanStack, 2024. – URL: https://tanstack.com/router/latest/docs/framework/react/guide/search-params (дата звернення: 01.04.2025).

16. Supabase Documentation. Row Level Security. – Supabase, 2024. – URL: https://supabase.com/docs/guides/database/row-level-security (дата звернення: 01.04.2025).

17. React Documentation. Hooks Reference. – Meta Open Source, 2024. – URL: https://react.dev/reference/react (дата звернення: 01.04.2025).

18. Vitest Documentation. Testing Framework for Vite. – Vitest, 2024. – URL: https://vitest.dev/guide/ (дата звернення: 01.04.2025).

19. PostgreSQL Documentation. Trigger Functions and daterange Type. – The PostgreSQL Global Development Group, 2024. – URL: https://www.postgresql.org/docs/current/rangetypes.html (дата звернення: 01.04.2025).

20. Cloudflare Workers Documentation. Edge Computing Platform. – Cloudflare, 2024. – URL: https://developers.cloudflare.com/workers/ (дата звернення: 01.04.2025).

21. Кравченко М. О., Сидоренко І. В. Методи оцінювання зручності використання інформаційних систем: навч. посіб. – Київ: КПІ ім. Ігоря Сікорського, 2020. – 148 с.

22. Adalo Platform Documentation. Building No-Code Applications. – Adalo Inc., 2024. – URL: https://docs.adalo.com (дата звернення: 01.04.2025).

23. TypeScript Handbook. The TypeScript Language Reference. – Microsoft, 2024. – URL: https://www.typescriptlang.org/docs/handbook/ (дата звернення: 01.04.2025).

24. Яремчук С. С. Аналіз та специфікація вимог до програмних систем: монографія. – Вінниця: ВНТУ, 2018. – 190 с.

25. Pavlo Golub. Mastering PostgreSQL 15: Advanced techniques to build and administer scalable and reliable PostgreSQL database applications. – 5th ed. – Packt Publishing, 2023. – 452 p.

26. Vite Documentation. Next Generation Frontend Tooling. – Vite Team, 2024. – URL: https://vite.dev/guide/ (дата звернення: 01.04.2025).

27. Флеткін А., Ванрой Д. React: швидкий старт. – Київ: Видавнича група BHV, 2019. – 384 с.

28. Закон України “Про захист персональних даних” від 01.06.2010 № 2297-VI. – Відомості Верховної Ради України, 2010. – URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 01.04.2025).

29. Ксюша Д. І. Проектування інформаційних систем: від вимог до реалізації: навч. посіб. – Харків: ХНЕУ, 2021. – 204 с.

30. Мойсеєнко М. В. Технологія хмарних обчислень: навч. посіб. – Київ: Університет “Україна”, 2022. – 236 с.

31. Tailwind CSS Documentation. A Utility-First CSS Framework. – Tailwind Labs, 2024. – URL: https://tailwindcss.com/docs (дата звернення: 01.04.2025).

32. Тичківська Н. О. Автоматизоване тестування веб-застосунків: методичний посібник. – Львів: ЛНУ імені Івана Франка, 2023. – 112 с.

33. GitHub Docs. GitHub Documentation. - [Електронний ресурс]. - Режим доступу: https://docs.github.com/ (дата звернення: 20.04.2026.)

34. Федько В.В, Безуглий Д.Є.. Побудова веб застосунків на основі конструктивних елементів// Системи обробки інформації. 2020. №1(60). С. 123-127. 10.30748/soi.2020.160.16.

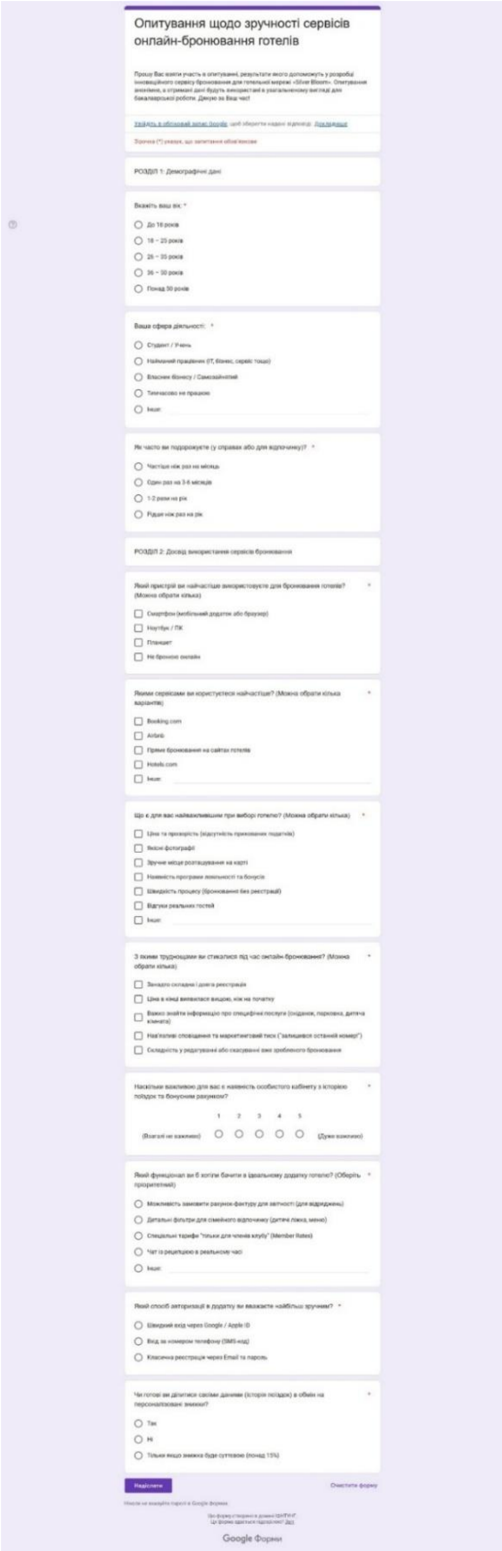
35. Душенко Ю. Є. Посилання на репозиторій проєкту https://github.com/DiJule/zen-dwell-app.git

.

ДОДАТКИ

ДОДАТОК А

Анкета «Опитування щодо зручності сервісів онлайн-бронювання готелів»:



Опитування щодо зручності сервісів онлайн-бронювання готелів

Прошу Вас взяти участь в опитуванні, результати якого допоможуть у розробці інноваційного сервісу бронювання для готельної мережі «Silver Bloom». Опитування анонімне, а отримані дані будуть використані в узагальненому вигляді для бакалаврської роботи. Дякую за Ваш час!

Увійдіть в обліковий запис Google, щоб зберегти надані відповіді. Докладніше

Зірочка (*) указує, що запитання обов'язкове

РОЗДІЛ 1: Демографічні дані

Вкажіть ваш вік: *

- До 18 років
- 18 – 25 років
- 26 – 35 років
- 36 – 50 років
- Понад 50 років

Ваша сфера діяльності: *

- Студент / Учень
- Найманий працівник (IT, бізнес, сервіс тощо)
- Власник бізнесу / Самозайнятий
- Тимчасово не працюю
- Інше:

Як часто ви подорожуєте (у справах або для відпочинку)? *

- Частіше ніж раз на місяць
- Один раз на 3-6 місяців
- 1-2 рази на рік
- Рідше ніж раз на рік

РОЗДІЛ 2: Досвід використання сервісів бронювання

Який пристрій ви найчастіше використовуєте для бронювання готелів? (Можна обрати кілька) *

- Смартфон (мобільний додаток або браузер)
- Ноутбук / ПК
- Планшет
- Не бронюю онлайн

Якими сервісами ви користуєтеся найчастіше? (Можна обрати кілька варіантів) *

- Booking.com
- Airbnb
- Пряме бронювання на сайтах готелів
- Hotels.com
- Інше:

Що є для вас найважливішим при виборі готелю? (Можна обрати кілька) *

- Ціна та прозорість (відсутність прихованих податків)
- Якісні фотографії
- Зручне місце розташування на карті
- Наявність програми лояльності та бонусів
- Швидкість процесу (бронювання без реєстрації)
- Відгуки реальних гостей
- Інше:

З якими труднощами ви стикалися під час онлайн-бронювання? (Можна обрати кілька) *

- Занадто складна і довга реєстрація
- Ціна в кінці виявлялася вищою, ніж на початку
- Важко знайти інформацію про специфічні послуги (сніданок, парковка, дитяча кімната)
- Нав'язливі сповіщення та маркетинговий тиск ("залишився останній номер!")
- Складність у редагуванні або скасуванні вже зробленого бронювання

Наскільки важливою для вас є наявність особистого кабінету з історією поїздок та бонусним рахунком? *

(Взагалі не важливо) 1 2 3 4 5 (Дуже важливо)

Який функціонал ви б хотіли бачити в ідеальному додатку готелю? (Оберіть пріоритетний) *

- Можливість замовити рахунок-фактуру для звітності (для відряджень)
- Детальні фільтри для сімейного відпочинку (дитяче ліжко, меню)
- Спеціальні тарифи "тільки для членів клубу" (Member Rates)
- Чат із рецепцією в реальному часі
- Інше:

Який спосіб авторизації в додатку ви вважаєте найбільш зручним? *

- Швидкий вхід через Google / Apple ID
- Вхід за номером телефону (SMS-код)
- Класична реєстрація через Email та пароль

Чи готові ви ділитися своїми даними (історія поїздок) в обмін на персоналізовані знижки? *

- Так
- Ні
- Тільки якщо знижка буде суттєвою (понад 15%)

Надіслати

Очистити форму

Ніколи не вводьте паролі в Google Формах

Цю форму створено в домені ЮНТУНГ
Ця форма здається підозрілою? Звіт

Google Форми

ДОДАТОК Б

ДЕТАЛЬНИЙ ОПИС ПЕРСОНАЖІВ ТА СЦЕНАРІЇ ВЗАЄМОДІЇ

У процесі аналізу вимог для веб-сервісу «Silver Bloom» було розроблено три персонажі (User Personas), що представляють ключові групи цільової аудиторії. Кожен персонаж описує типового користувача з його цілями, поведінковими патернами та сценаріями взаємодії з системою [9, 32].

Персонаж 1. Андрій Коваленко — бізнес-мандрівник

Вік: 38 років. Посада: регіональний менеджер з продажів. Місто: Київ. Частота подорожей: 6–10 відряджень на рік.

Цілі: швидко забронювати номер без зайвих кроків; отримати рахунок-фактуру для звітності у бухгалтерію; доступ до Member Rates як постійного клієнта клубної програми.

Больові точки: тривалі форми реєстрації; відсутність прозорої ціни “для своїх”; неможливість переглянути тарифи без входу в систему.

Сценарій взаємодії: Андрій відкриває сайт у мобільному браузері між зустрічами. Вводить дати та кількість гостей у SearchBar на головній сторінці. Переглядає відфільтрований список номерів, сортований за ціною. Авторизується для доступу до Member Rates. Обирає Standard Room та натискає «Reserve now». Отримує підтвердження бронювання на email. В особистому кабінеті завантажує деталі броні для бухгалтерії. Весь процес займає менше 3 хвилин.

Персонаж 2. Олена та Сергій Бондаренко — сім’я на відпочинку

Вік: 35 та 37 років, двоє дітей (5 та 9 років). Частота подорожей: 1–2 сімейних відпустки на рік. Пристрій: переважно ноутбук або планшет.

Цілі: обрати номер, достатньо просторий для чотирьох осіб (Family Room); дізнатися про дитячу інфраструктуру (дитячий майданчик, меню); порівняти кілька варіантів перед бронюванням.

Больові точки: неможливість знайти номер з фільтром «для 4 гостей»; відсутність детального опису зручностей для дітей; незрозуміла цінова різниця між категоріями номерів.

Сценарій взаємодії: Олена відкриває сайт ввечері з ноутбука. Вводить дати відпустки та 4 гостей у пошуковий блок. Система відображає лише ті номери, що мають достатню місткість (capacity ≥ 4). Переглядає фотогалерею та список amenities кожного номеру на сторінці деталей. Порівнює Family Room та Suite за ціною та зручностями. Без реєстрації переходить до форми бронювання, де системи запитує email для підтвердження. Отримує на email деталі бронювання зі списком включених послуг.

Персонаж 3. Ірина Мельник — постійна клієнтка

Вік: 45 років. Посада: власниця малого бізнесу. Місто: Львів. Частота відвідувань: 3–4 рази на рік, зокрема ділові та особисті поїздки.

Цілі: доступ до Member Rates через Club Card; перегляд накопиченої історії бронювань та бонусних балів у особистому кабінеті; швидке повторне бронювання улюбленого номеру.

Больові точки: відсутність персоналізованих пропозицій для лояльних клієнтів; неможливість переглянути всю історію бронювань в одному місці; складна навігація в особистому кабінеті.

Сценарій взаємодії: Ірина відкриває сайт і одразу натискає «Log In», оскільки є постійною клієнткою. Після авторизації в каталозі номерів відображаються Member Rates — знижені тарифи для Club Card. Переглядає секцію «Upcoming stays» в особистому кабінеті: перевіряє, чи немає перетину з майбутнім бронюванням. Обирає Deluxe Room за спеціальною ціною та підтверджує бронювання одним кліком. В секції «History» переглядає 9 попередніх бронювань та суму накопичених балів. За потреби скасовує неактуальне бронювання через AlertDialog з підтвердженням.

Таблиця Б.1

Зведений профіль персонажів веб-сервісу Silver Bloom

	Персонаж 3	**Персонаж 2**	**Персонаж 1**
Пріоритет	Швидкість, Member Rate	Місткість, зручності для дітей	Club Card, історія бронювань
Авторизація	Так (для Member Rate)	Необов'язково	Обов'язково
Пристрій	Смартфон	Ноутбук/планшет	Ноутбук
Ключова функція	Member Rates, швидке бронювання	Фільтр за місткістю, фото	Особистий кабінет, повторне бронювання

Розроблені персонажі стали основою для визначення пріоритетів функціональних вимог та проектування UX-рішень веб-сервісу Silver Bloom, описаних у розділах 2 та 5 бакалаврської роботи.

ДОДАТОК В

ER-ДІАГРАМА БАЗИ ДАНИХ SILVER BLOOM

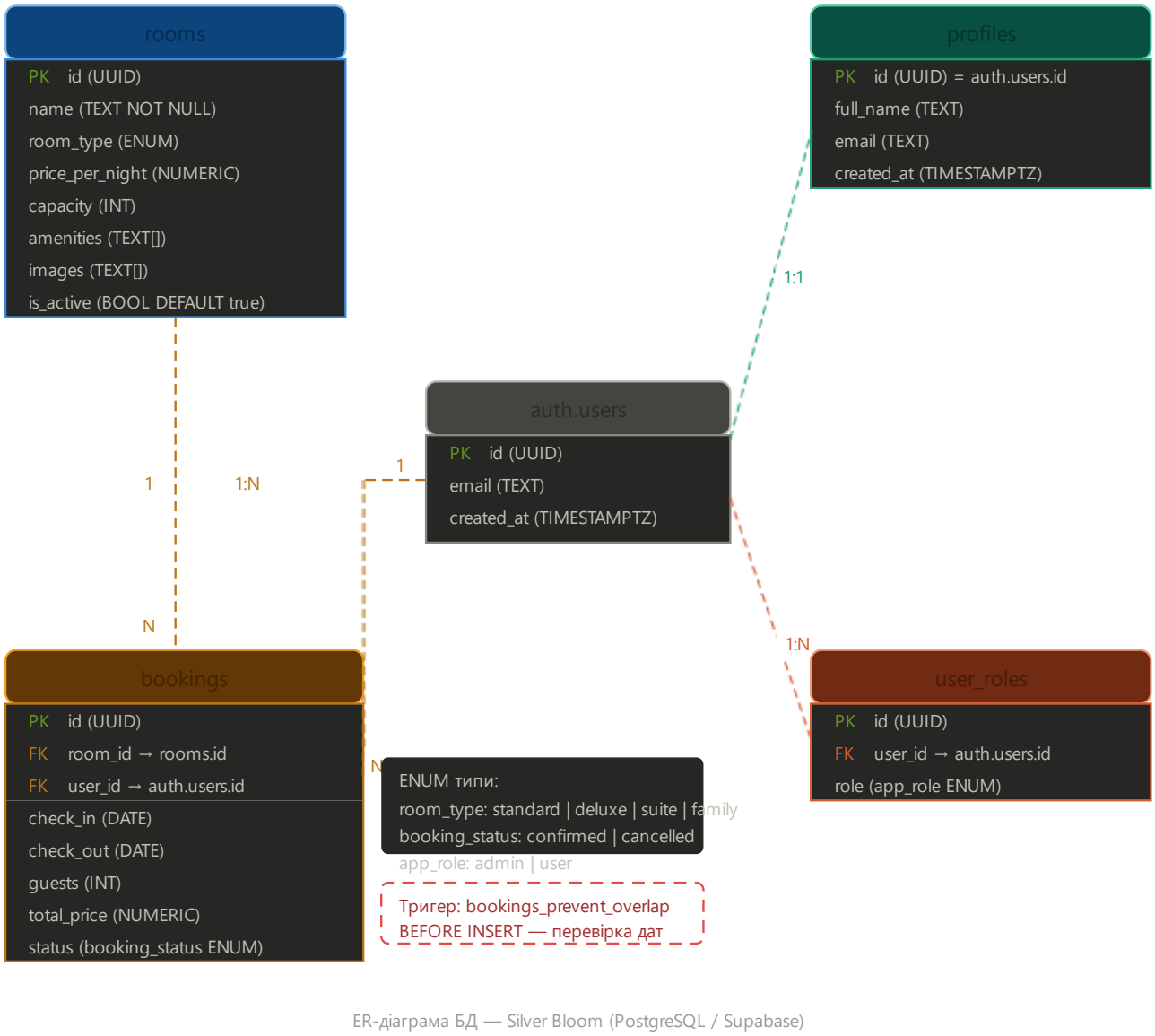


Рисунок В.1 – ER-діаграма бази даних PostgreSQL проєкту Silver Bloom

ER-діаграма (Entity-Relationship Diagram) відображає логічну структуру реляційної бази даних PostgreSQL проєкту Silver Bloom, побудованої засобами Supabase. Діаграма включає чотири основні сутності (таблиці) та їх зв’язки. Таблиця **rooms** (id UUID PK, name TEXT, room_type ENUM, price_per_night NUMERIC, capacity INT, description TEXT, amenities TEXT[], images TEXT[],

is_active BOOL) є центральною сутністю каталогу. Таблиця **bookings** (id UUID PK, room_id UUID FK→rooms, user_id UUID FK→auth.users, check_in DATE, check_out DATE, guests INT, total_price NUMERIC, status booking_status ENUM) реалізує зв’язок «багато до одного» з rooms. Таблиця **profiles** (id UUID PK FK→auth.users, full_name TEXT, email TEXT, created_at TIMESTAMPTZ) розширює вбудовану систему автентифікації Supabase. Таблиця **user_roles** (id UUID PK, user_id UUID FK→auth.users, role app_role ENUM) реалізує RBAC-модель. Між bookings та auth.users існує зв’язок «багато до одного» через user_id.

ДОДАТОК Г

API-специфікація основних ендпоінтів (OpenAPI-стиль):

Взаємодія між клієнтом і Supabase відбувається через PostgREST API — автоматично згенерований REST-інтерфейс над PostgreSQL-схемою. Нижче наведено специфікацію ключових ендпоінтів у форматі, що відповідає стандарту OpenAPI 3.0. Базовий URL: https://<project-id>.supabase.co. Автентифікація: Bearer JWT у заголовку Authorization.

Каталог номерів

GET /rest/v1/rooms?is_active=eq.true&order=price_per_night.asc — отримати список активних номерів, відсортованих за ціною. Параметри фільтрації: room_type=eq.{type} (фільтр за типом), capacity=gte.{n} (мінімальна місткість), price_per_night=lte.{max} (максимальна ціна). Відповідь 200: масив об'єктів Room. Доступно без автентифікації (RLS allow all for SELECT WHERE is_active=true).

GET /rest/v1/rooms?id=eq.{roomId} — отримати деталі конкретного номеру. Відповідь 200: об'єкт Room; 404: якщо номер не знайдений або неактивний.

Бронювання

POST /rest/v1/bookings — створити нове бронювання. Тіло запиту (JSON): {"room_id": "uuid", "check_in": "YYYY-MM-DD", "check_out": "YYYY-MM-DD", "guests": N, "total_price": N.NN}. Заголовки: Authorization: Bearer {JWT}, Content-Type: application/json, Prefer: return=representation. Відповідь 201: створений об'єкт Booking; 409: конфлікт дат (тригер bookings_prevent_overlap повернув RAISE EXCEPTION).

GET /rest/v1/bookings?user_id=eq.{userId}&select=*,rooms(*) — отримати бронювання поточного користувача з деталями номеру (RLS обмежує вибірку лише власними записами). Відповідь 200: масив Booking із вкладеним об'єктом Room.

PATCH /rest/v1/bookings?id=eq.{bookingId} — оновити статус бронювання. Тіло запиту: {"status": "cancelled"}. Заголовки: Authorization: Bearer {JWT}.

Відповідь 200: оновлений запис; 403: якщо JWT не відповідає власнику або адміністратору (RLS).

Автентифікація (Supabase Auth API)

POST /auth/v1/signup — реєстрація нового користувача. Тіло: {“email”: string, “password”: string}. Відповідь 200: {user, session} з JWT access_token та refresh_token. Тригер on_auth_user_created автоматично створює profile та user_role='user'.

POST /auth/v1/token?grant_type=password — вхід існуючого користувача. Тіло: {“email”: string, “password”: string}. Відповідь 200: {access_token, refresh_token, expires_in, user}; 400: невірні облікові дані.

POST /auth/v1/logout — завершення сесії. Заголовок: Authorization: Bearer {JWT}. Відповідь 204: сесія анульована.

Повна OpenAPI 3.0 YAML-специфікація автоматично генерується Supabase і доступна за адресою: https://<project-id>.supabase.co/rest/v1/?apikey=<anon-key>. Swagger UI для інтерактивного тестування ендпоінтів доступний через Supabase Dashboard — розділ API Docs.

ДОДАТОК Д

Інтерфейс застосунку:

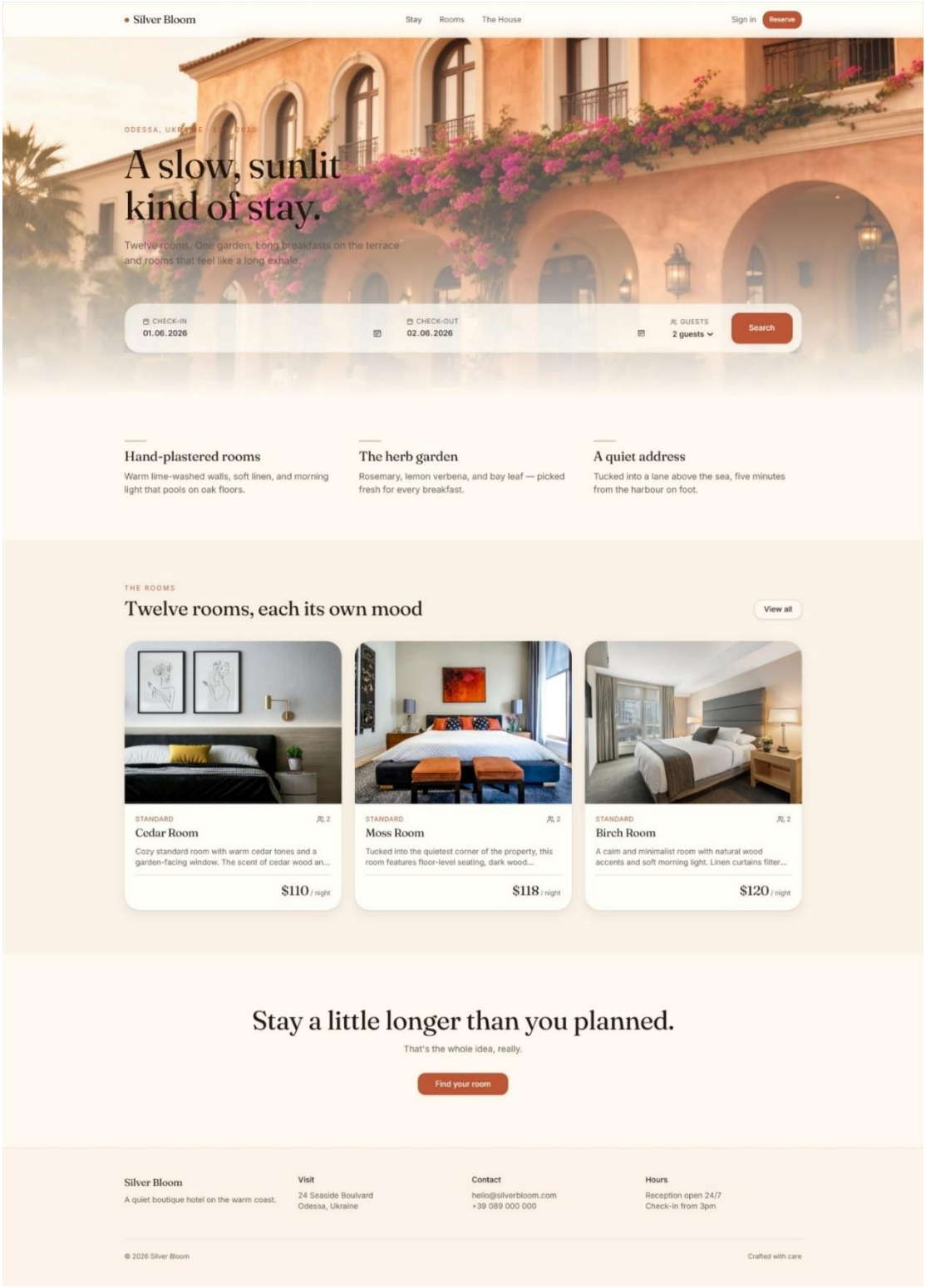


Рисунок Д.1 – Головна сторінка

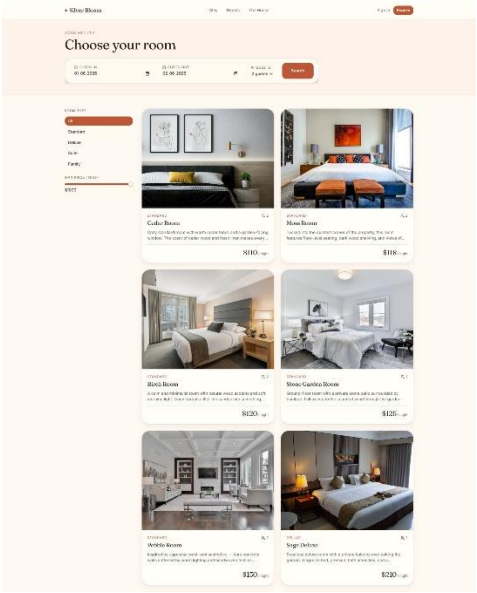


Рисунок Д.2 – Сторінка «Rooms»

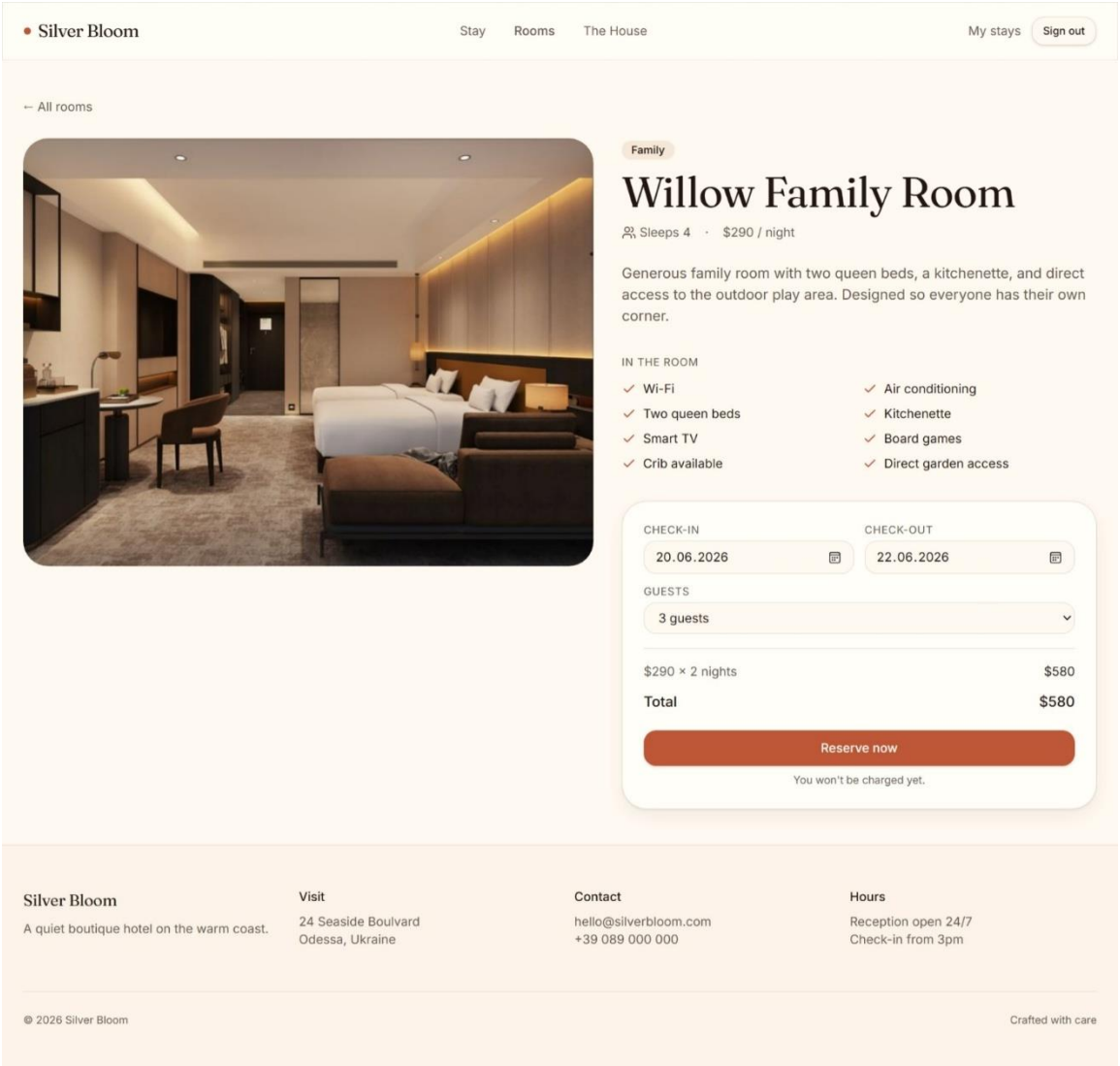


Рисунок Д.3 – Сторінка «Номер Willow Family»

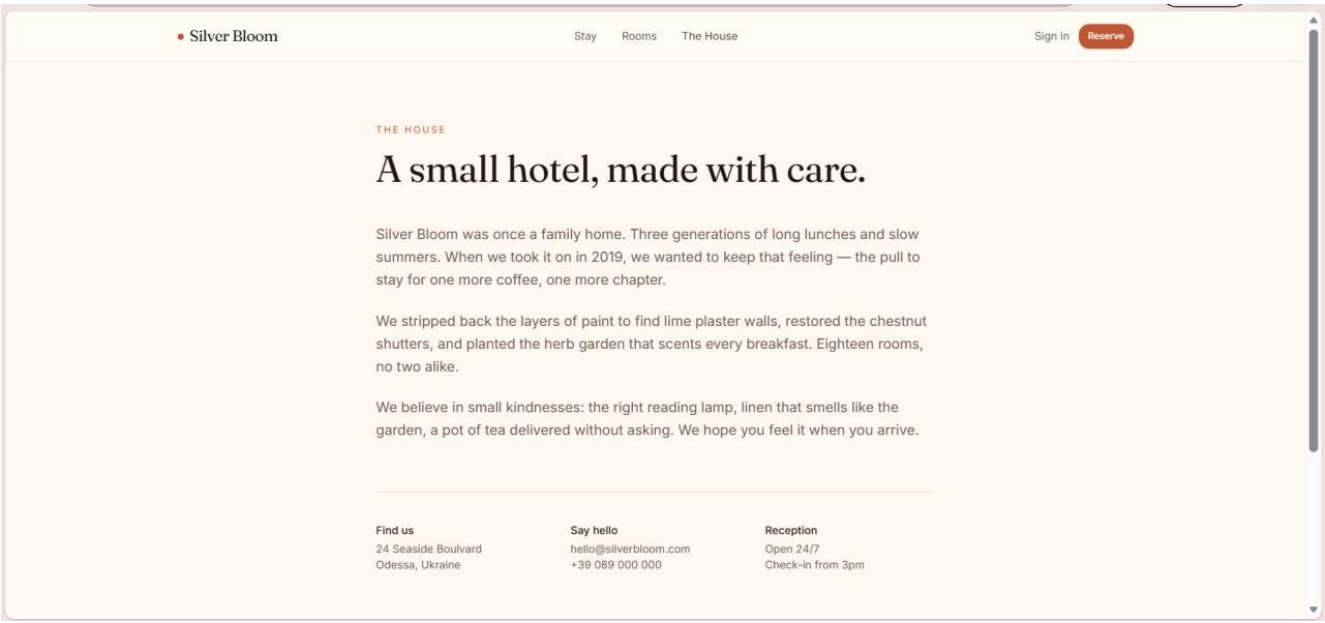


Рисунок Д.4 – Сторінка «Про будинок»

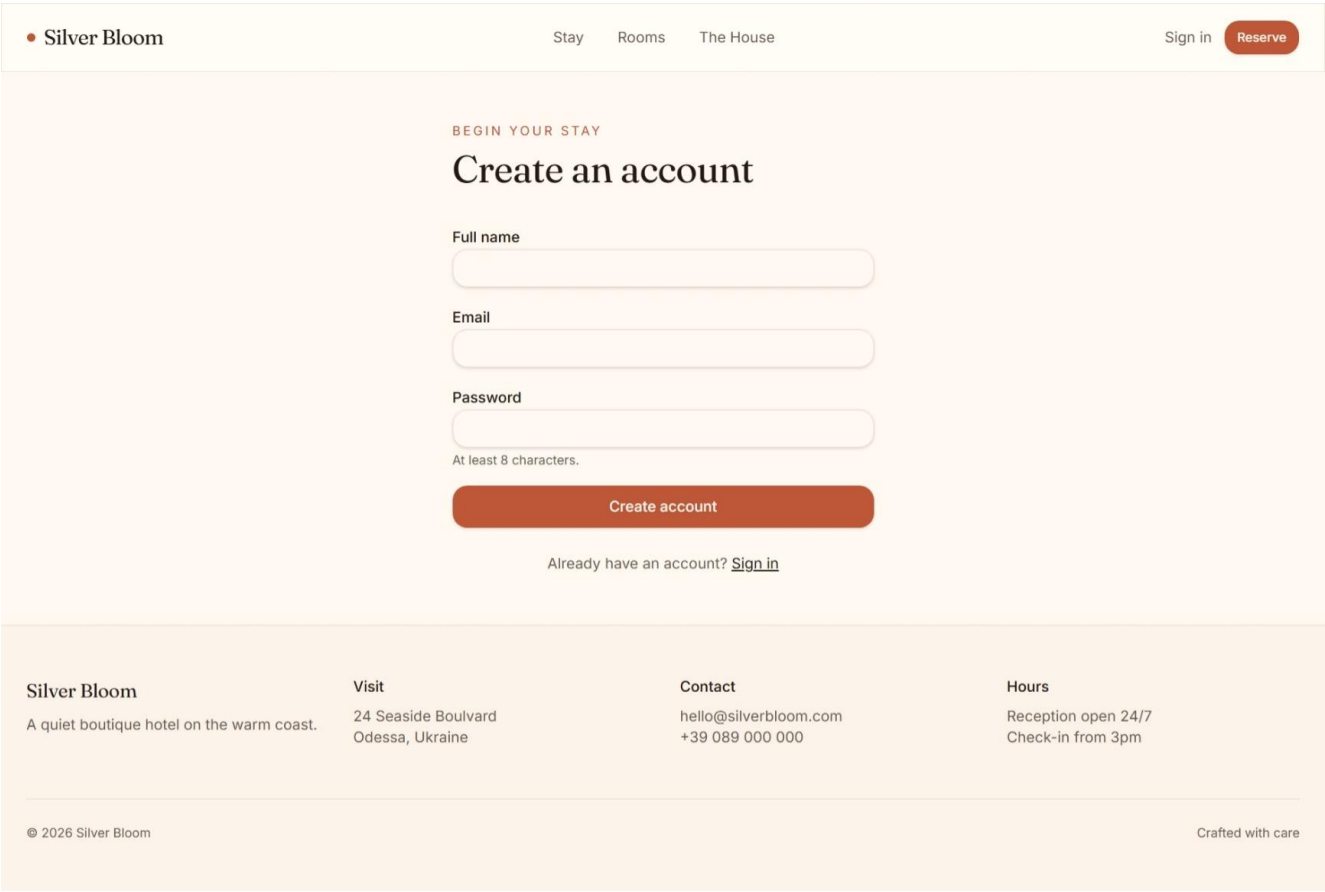


Рисунок Д.5 – Сторінка «Реєстрація»

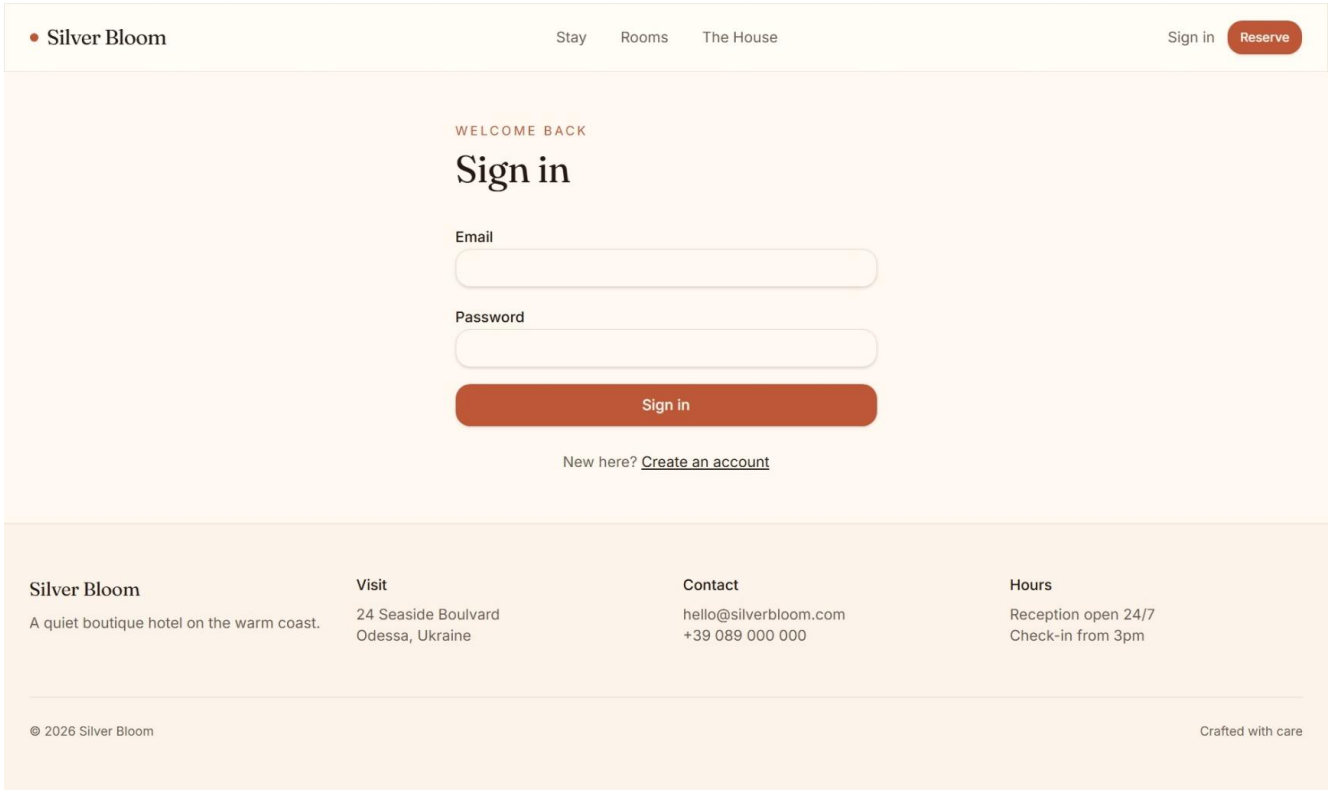


Рисунок Д.6 – Сторінка входу

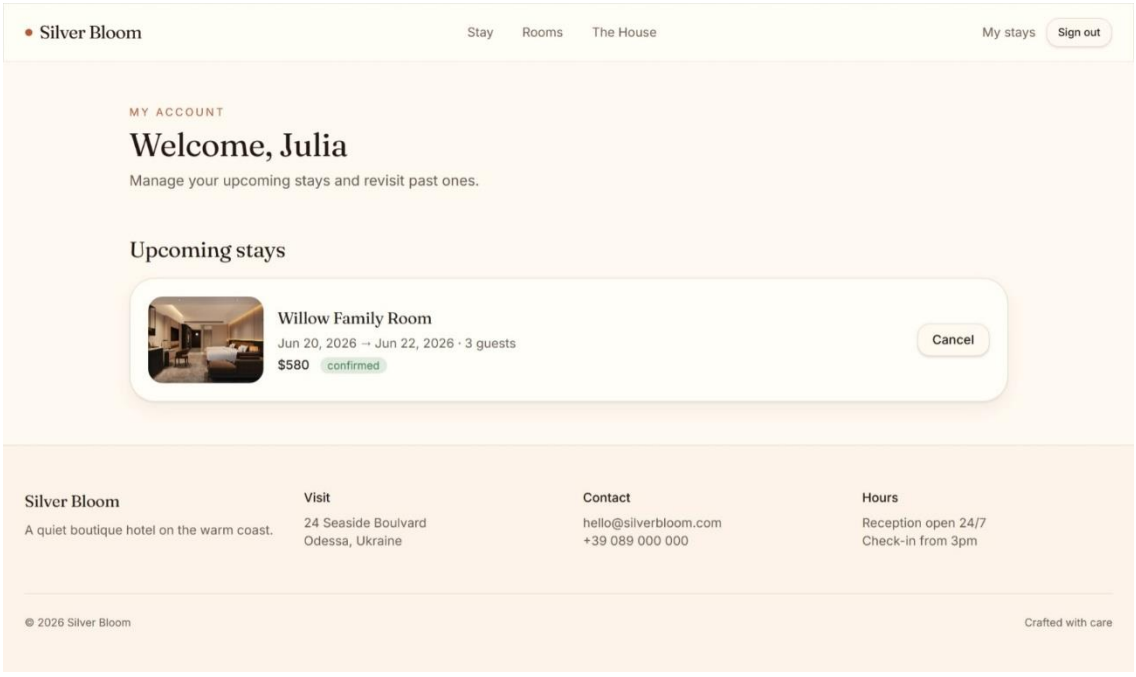
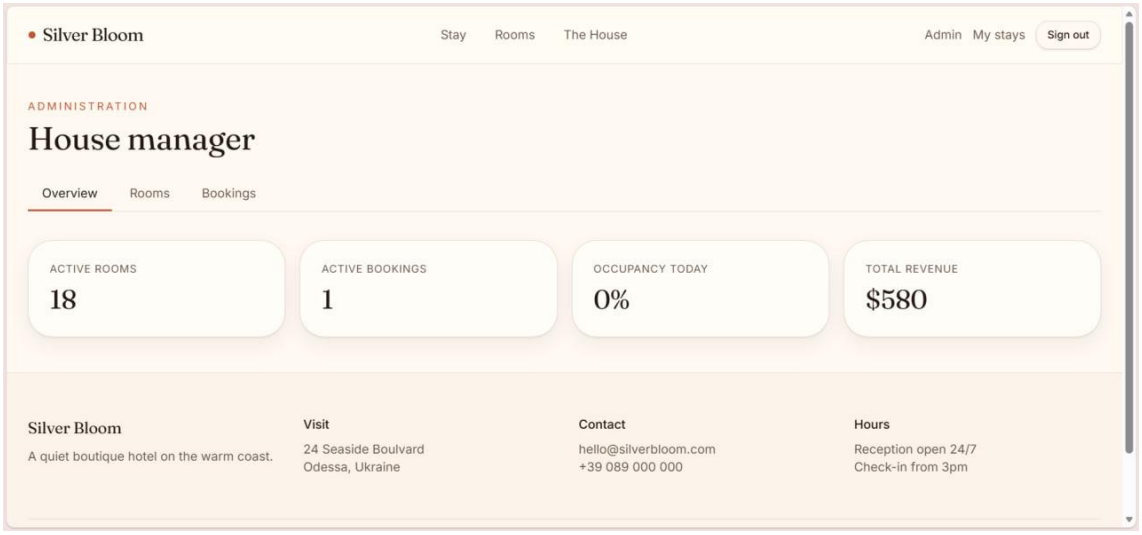
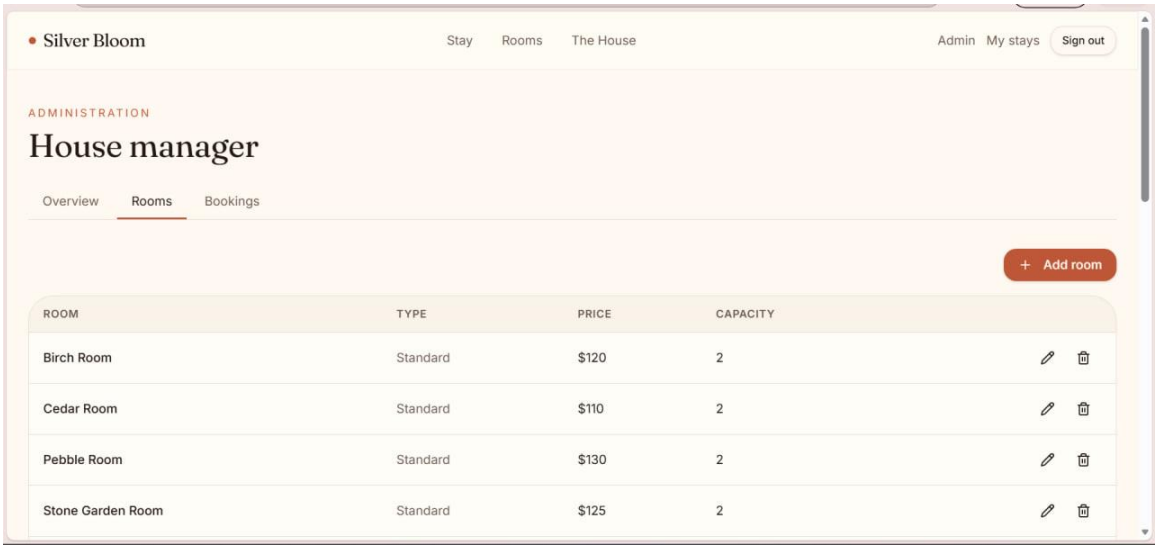


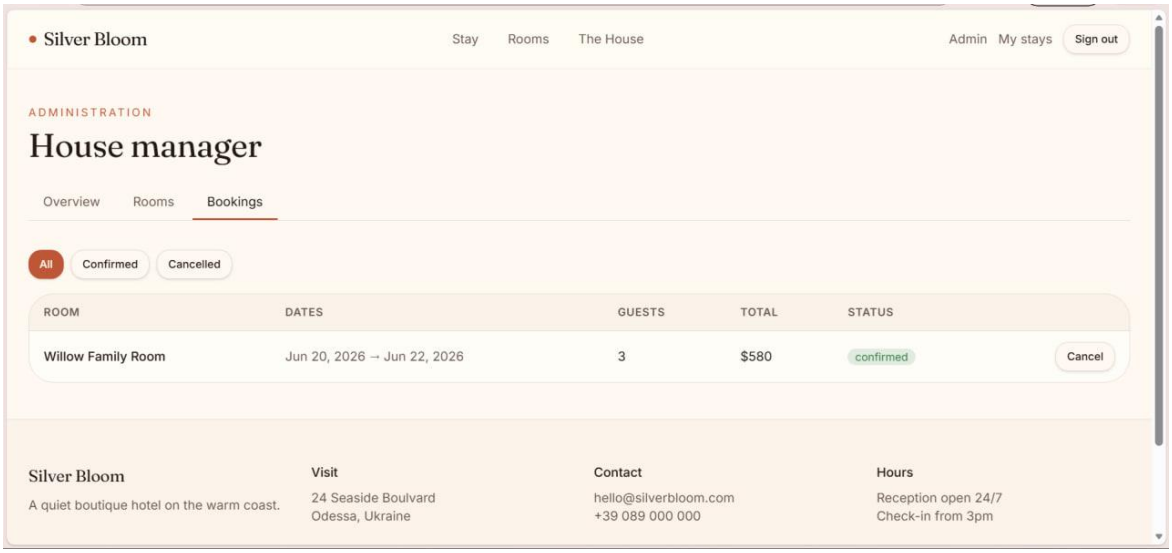
Рисунок Д.7 – Сторінка «Мої бронювання»



а)



б)



в)

Рисунок Д.8 – Сторінка адмін панелі

ДОДАТОК Е

Посилання на Git Hub: https://github.com/DiJule/zen-dwell-app.git

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: «Побудова ВЕБ-сервісів на основі архітектурних шаблонів ».

Обсяг пояснювальної записки: 76 аркушів

Дата закінчення бакалаврської роботи «5» червня 2026 р.

Підпис студента _____