НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук

_____/ Голуб Б.Л., доц., к.т.н _ /
підпис ПІБ, вчене звання і ступінь

«__»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмне забезпечення для онлайн-сервісу продажу квітів»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент / Вайганг Г.О. /
Науковий ступень та вчене звання підпис ПІБ

Керівник бакалаврської кваліфікаційної роботи : / Баранова Т. А. /

Консультант бакалаврської кваліфікаційної роботи
к.т.н, доцент _____ _____ _ Даков С.Ю.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

підпис ПІБ
Виконала:_____/ _____Щур В.М. __/
підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук

_____ / Голуб Б.Л., доцент, к.т.н /
підпис
"_16_" грудня 2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студентці Щур Вероніці Миколаївні

Спеціальність 121 «Інженерія програмного забезпечення»

1. Тема роботи: Програмне забезпечення для онлайн-сервісу продажу квітів

Затверджена наказом ректора НУБіП України № 2248 "С" від 16.12.2024

2. Термін подання завершеної роботи на кафедру 2025.__.__
рік, місяць, число

3. Вихідні дані до роботи: опис програмного забезпечення

4. Перелік питань що розглядаються:

1. Аналіз проблемної області.
2. Вибір та обґрунтування засобів для розробки системи.
3. Проєктування інформаційної системи.
4. Висновки.

Керівник бакалаврської кваліфікаційної роботи _____ / Баранова Т. А. /
підпис ініціали та прізвище

Завдання прийняла до виконання _____ / Щур В.М. /
підпис ініціали та прізвище

Дата отримання завдання 2024.12.16
рік, місяць, число

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ..... 6

ВСТУП.....7

РОЗДІЛ 1 АНАЛІЗ ПРОГРАМНОЇ ОБЛАСТІ.....10

1.1 Постановка задачі.....10
1.2 Моделювання предметної області..... 12
1.3 Діаграма прецедентів.....15
1.4 Діаграма діяльності..... 18
1.5 Діаграма послідовності.....20

РОЗДІЛ 2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ..... 23

2.1 Загальні відомості про ER-діаграму.....23
2.2 Побудова ER-діаграми.....26
2.3 Вибір та обгрунтування СУБД..... 29
2.4 Створення СУБД.....35

РОЗДІЛ 3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ..... 43

РОЗДІЛ 4 ВПРОВАДЖЕННЯ СИСТЕМИ.....47

РОЗДІЛ 5 ТЕСТУВАННЯ СИСТЕМИ.....50

5.1 Адаптивність інтерфейсу користувача.....50
5.2 Мовна локалізація та відображення каталогу..... 51
5.3 Перевірка доступу до оформлення без авторизації.....52
5.4 Авторизація та реєстрація користувачів.....52
5.5 Профіль користуваача.....53
5.6 Оформлення замовлення..... 55
5.7 Панель адміністратора.....57

ВИСНОВКИ.....59

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....61

ДОДАТКИ.....63

Додаток А – Фрагменти коду frontend-частини.....63
Додаток В – Фрагменти коду backend-частини.....67

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API: Application Programming Interface – інтерфейс прикладного програмування

БД: База даних

BPMN: Business Process Model and Notation – нотація моделювання бізнес-процесів

DTO: Data Transfer Object – об’єкт передавання даних між шарами застосунку

ER: Entity-Relationship – сутність–зв’язок

HTTP: HyperText Transfer Protocol – протокол передавання гіпертексту

JSON: JavaScript Object Notation – текстовий формат обміну даними

ORM: Object-Relational Mapping – об'єктно-реляційне відображення

REST: Representational State Transfer – архітектурний стиль вебсервісів

SQL: Structured Query Language – мова структурованих запитів

SRS: Software Requirements Specification – специфікація програмних вимог

СУБД: Система управління базами даних

UML: Unified Modeling Language – уніфікована мова моделювання

UNIX: Уніфікована операційна система

VSD: Vision and Scope Document – документ бачення і меж проєкту

ВСТУП

Сфера надання флористичних послуг дуже стабільна та має велике різноманіття як клієнтів, так і послуг, за рахунок чого активно зростає та розвивається. Специфіка продажу квітів у кожній із галузей суттєво відрізняється. Загалом її можна розділити на такі напрямки: B2B, B2C, C2C. Для кожної групи покупців потрібно створити умови, які б задовольняли їхні потреби та дозволяли найефективніше взаємодіяти з аудиторією.

За умов високої конкуренції, для кращої залученості клієнтів, а як наслідок – підвищення ефективності роботи бізнесу, підприємці мають забезпечити якісний сервіс та зручні засоби взаємодії з кінцевим споживачем. Однак сфера роздрібного продажу квітів суттєво відстає від інших сегментів у плані розвитку онлайн-сервісів. Це пов'язано передусім із відсутністю фінансових та технічних можливостей у малого бізнесу для розробки індивідуального програмного забезпечення. Додатковими стримуючими факторами виступають складність логістики, нетривалий термін зберігання товару, а також відсутність уніфікованих рішень, які можна було б адаптувати без значних витрат.

У цьому контексті розробка універсального засобу, який реалізує функціонал онлайн-сервісу для продажу квітів – від перегляду товару та оформлення замовлення до керування асортиментом і моніторингу виконання замовлень, – є вкрай актуальною. Така система дозволить автоматизувати основні бізнес-процеси, зменшити витрати на обслуговування клієнтів, розширити охоплення ринку та підвищити конкурентоспроможність підприємств.

Об’єктом дослідження є процес автоматизації діяльності підприємств у сфері роздрібного продажу квіткової продукції за допомогою сучасних інформаційних технологій.

Предметом дослідження є функціональні, архітектурні та технологічні особливості розробки програмного забезпечення для онлайн-сервісу продажу квітів.

Запропоноване рішення має стати ефективним інструментом для бізнесів будь-якого масштабу, надаючи їм змогу інтегруватися у цифрове середовище без значних витрат на розробку. Система, підключення до якої відбуватиметься на основі уніфікованої архітектури, дозволить автоматизувати взаємодію з клієнтами, вести облік товару, керувати замовленнями та контролювати запаси.

Метою роботи є розробка програмного забезпечення для онлайн-сервісу з продажу квітів, яке забезпечить зручну взаємодію між покупцем та продавцем, ефективне керування асортиментом і замовленнями, а також гнучкість і масштабованість для малого бізнесу.

Для досягнення поставленої мети у роботі необхідно вирішити такі **завдання**:

- дослідити специфіку ринку флористичних послуг та потреби цільової аудиторії;
- проаналізувати сучасні технології та засоби розробки вебсервісів;
- охарактеризувати вимоги до архітектури системи та обґрунтувати вибір інструментів;
- встановити ключові функціональні та нефункціональні вимоги до розроблюваного ПЗ;
- визначити особливості реалізації клієнтської та серверної частини;
- з’ясувати шляхи забезпечення масштабованості, безпеки та стабільності роботи системи;
- розробити REST API, реалізувати користувацький інтерфейс і забезпечити керування контентом;
- висвітлити роль тестування та забезпечення якості програмного забезпечення.

Практичне значення роботи полягає у створенні програмного забезпечення, яке можна впроваджувати у діяльність малих підприємств для автоматизації процесу продажу квітів, що дозволить підвищити якість обслуговування, зменшити навантаження на персонал і збільшити обсяги продажу.

Теоретичне значення полягає у дослідженні підходів до архітектурного проєктування веборієнтованих інформаційних систем, що можуть бути

використані у подальших наукових і прикладних дослідженнях з програмної інженерії.

Система є актуальною з таких причин:

1. Покращення якості взаємодії між клієнтом та надавачем послуг за рахунок часткової автоматизації процесів;
2. Зменшення часу, необхідного для оформлення замовлення;
3. Наглядність та доступність асортименту, що спрощує вибір товару;
4. Покращення користувацького досвіду за рахунок зручного інтерфейсу;
5. Підвищення ефективності управління бізнесом завдяки систематизації замовлень та автоматизації процесів.

Розробка такої системи вимагає дотримання вимог до стабільності, масштабованості, безпеки даних та привабливості інтерфейсу. Для цього необхідно застосовувати сучасні підходи та технології індустрії розробки програмного забезпечення.

РОЗДІЛ 1
АНАЛІЗ ПРОГРАМНОЇ ОБЛАСТІ

1.1 Постановка задачі

Під час розробки концепції програмного забезпечення “Онлайн-сервіс салону квітів” для підприємців, було встановлено перелік цілей, функцій та вимог до системи.

Основні цілі програми:

- Покращити взаємодію кінцевого покупця та закладу, що надає послуги
- Організувати єдиний простір для контролю та менеджменту асортименту товарів
- Створити систему, яка буде простою в дистрибуції та забезпечувати усі потрібні функції

Система “Онлайн сервіс продажу квітів” призначена для покращення користувацького досвіду взаємодії із замовленням та підвищення ефективності та оптимізації процесу купівлі квітів. Система являє собою REST API для обробки HTTP запитів, та клієнтську частину у вигляді веб сайту, за допомогою якої клієнт буде взаємодіяти з API. За допомогою цього ресурсу клієнт зможе переглядати каталог доступних квітів, сформувати з них букет, чи обрати уже зібраний та замовити доставку.

Серед інших функцій буде: можливість переглянути місцезнаходження магазинів, переглянути і додати відгуки, корисну інформацію, зв’язатись з магазином.

Основні ролі користувачів:

Адміністратор: роль, в переліках прав якої є можливість керувати наповненням магазину, редагувати, створювати, видаляти товари. Роль “Адміністратор” в контексті магазину репрезентує менеджера.

Користувач: роль в системі, яка репрезентує покупця, має базовий набір дозволів, які дозволяють користуватись функціоналом перегляду товарі, їх покупкою, але при цьому забороняють їх зміну чи видалення.

Система спроєктована за принципом “client/server”. Покупець буде заходити на веб сайт, користуватись елементами навігації вибору товару та формування замовлення, після чого надсилати запит до API, яке буде опрацьовувати замовлення, формувати відповідні ресурси в базі даних та усіх супровідних сервісах

1. Функціональні вимоги
    a. Функціональні вимоги роль “Користувач”
        i. Авторизація
        ii. Формування замовлення
        iii. Оплата замовлення
        iv. Відслідковування замовлення
    b. Функціональні вимоги роль “Адміністратор”
        i. Авторизація
        ii. Моніторинг замовлень
        iii. Створення товарів
        iv. Видалення товарів
        v. Редагування товарів
2. Нефункціональні вимоги
    a. Зручний та зрозумілий інтерфейс, який буде відповідати сучасним вимогам до дизайну
    b. Стабільність системи
    c. Стресостійкість
    d. Здатність до розширення

У результаті виконання роботи очікується отримати програмний комплекс, який забезпечить можливість перегляду товарів та формування замовлення для користувача і контролювати процес продажів, інвентаризацію та стан складу

адміністратору з іншої сторони. Система забезпечить автономний, незалежний, стабільний потік замовлень в каталогізованому сепарованому середовищі, що підвищить зручність адміністрування бізнесу та збільшисть загальні показники, як: KPI, СПІ, середній чек

1.2 Моделювання предметної області

Для побудови об’ємної системи, яка буде стабільно працювати, задовільняти користувача по усім аспектам, виконувати певний перелік функції прогнозовано із передбаченою обробкою так званих “корнер-кейсів”, мати способи та засоби взаємодії для кожного запланованого сценарію потрібно провести грунтовний аналіз того, що буде розроблятись.

Планування повинно охоплювати не тільки системі чи функціональні вимоги, але й нефункціональні, з урахуванням графічної складової системи, якщо така передбачена. Загалом, із розвитком галузі інформаційних технологій така робота перейшла від розробників безпосередньо до окремої професії, яка має назву “бізнес-аналітик”.

Такі спеціалісти мають власні методології, інструменти, практики для того, щоб провести аналіз вимог до системи, опрацювати потреби, цільову аудиторію, створити опис основних бізнес процесів та супутньої документації для покращення етапу планування, більш детального та конкретного сприйняття програмного забезпечення подальними спеціалістами, які будуть розроблювати та впроваджувати систему.

Не дивлячись на те, що така робота частова займає значну частину бюджету проєкту, потребує значого часу для її виконання, а іноді навіть спеціаліалістів вузького направлення, які мають досвід в аналізі конкретних областей – відмова від цього етапу може понести за собою значні збитки у подальшій експлуатації системи. Тому часто ціна такого аналізу може коштувати стільки ж часових та фінансових затрат, ніж безпосередня розробка та впровадження цієї системи в життя, а іноді навіть і більше. Процес аналізу запобігає тому, що розробники не до кінця розуміють процес, який створюють у

кодовій реалізації, не розуміють функціональних вимог, допускають рішення, які можуть повпливати на подальше розширення чи удосконалення системи. За рахунок цього, якщо враховувати вартість у грошовому та часовому еквіваленті процесу аналізу системи у співвідношенню тих самих показників для розробки системи без планування на протязі певного часу вигода може становити 100% від вартості усього продукту. Також, за рахунок аналізу системи та розумінню результатів аналізу можна детально спланувати та розділити процес розробки системи, що у свою чергу дозволить встановити час на створення певного переліку функцій, кожної функції окремо, чи усього продукти. Таке розуміння цінне для бізнесу, бо дозволяє планувати та адаптуватись під реалії.

Серед інструментів, якими користуються спеціалісти з аналізу систем є різноманітні системи планування, які дозволяють контролювати ресурси проєкту та команди, співвідносити можливості та потреби, проводити аналіз виробничої потужності структурних підрозділів та команди загалом. Серед таких Jira, Monday.com, Trello та багато інших системних комплексів, які надають подібні можливості та функціонал.

Також існують різні методології оцінки як проєктної роботи, так і систем в цілому. Такі методології дозволяють на ранніх етапах провести планування, поки команда не занурена у дрібну операційну роботу, але при цьому вони суттєво спрощують подальшу роботу із менеджментом та плануванням роботи. Такі методології поділяються та каскадні, гнучкі, змішані. Залежно від обраної методології підходи до роботи над проєктом коригуються і мають певну специфіку.

Незалежно від обраної методології розробки чи засобу контролю розробки кожен проєктний аналіз супроводжується великою кількістю проєктної документації. Серед таких документів є: VSD (Vision & Scope document), Use case, Use stories, різноманітні види документації Business process model (BPMN, UML), SRS (Software requirements specifications). Кожен із них описує роботу під певним кутом погляду. Наприклад, документ VSD описує межі розробки ПЗ, цільову аудиторію, ключові функції, ціль продукту. Документ Business process

model має на меті візуалізувати процеси, наочно зобразити логіку системи загалом, або певних цільових аспектів, та обов’язково включає створення візуалізацій у вигляді UML діаграм.

Як було вказано вище, документ Business process model обов’язково містить візуалізацію у вигляді UML діаграм. UML - інструмент для візуалізації певних аспектів планування або опису системного комплексу. Існують певні правила побудови таких документів(діаграм), завдяки чому забезпечується уніфікація цього виду нотації. Ці правила були встановлені через те, що такі документи є дуже цінними як для планування, так і для подальшого процесу створення і підтримання продукту. Перш за все варто зазначити, що існують певні види таких документі: Class diagram, Activity diagram, Package diagram, кожна з яких висвітлює різні репрезентації системи. Для розроблюваної системи було побудовано ряд таких репрезентацій, які представлені далі в роботі.

Для кращого уявлення про загальну архітектуру модулів системи було побудовано діаграму пакетів, яка ілюструє логічний поділ застосунку на основні компоненти, їх взаємозв'язки та залежності між ними (рис. 1.1).

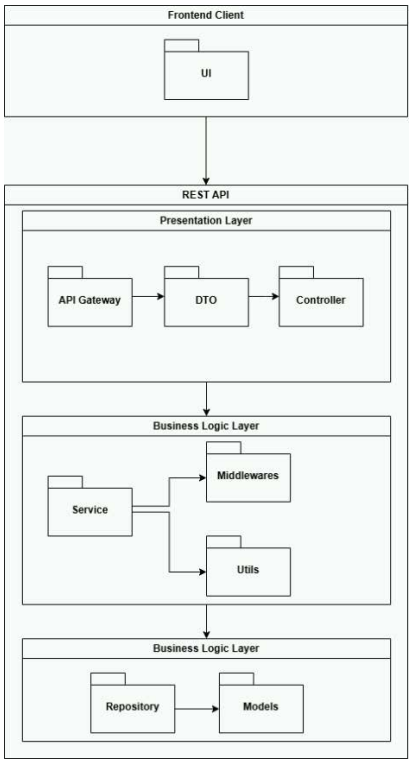


Рис. 1.1. Діаграма пакетів, що демонструє логічну структуру застосунку та взаємозв’язки між основними компонентами

1.3 Діаграма прецедентів

Діаграма прецедентів це вид документу Business process model в нотації UML, який є дуже важливим на початковому етапі планування системи. Цей документ наочно зображає комунікацію та взаємодію зовнішніх користувачів системи (акторів) і системи безпосередньо.

Основними задачами цього виду документу є:

1. Виявлення меж відповідальності системи
2. Зображення методів взаємодії системи та користувачів
3. Зображення переліку функцій, які система повинна мати

Як було вказано вище, діаграма прецедентів будується за певними правилами та нотацією. Нотація хакартеризується певними графічнимим елементами, такими як: зв’язки та їх види, система, актори та їх види, прецеденти. Кожен елемент такої нотації схематично зображує взаємодію однієї абстрації з іншою/іншими, за рахунок чого такі діаграми є зручними для сприйняття прецедентів. Прецедент це певний обмежений проміжок дій та активностей, який формує собою процес. В описі однієї діаграми може бути як один так декілька прецедентів, або навіть процесів.

Діаграми, де описується одразу декілька процесів являють собою комплексні процеси у повному проміжку їх відповідальності. При роботі з діаграмою варто виділити декілька нюансів, які допомогожуть краще розуміти її та дозволять отримувати грунтовніший та більш осяжний спектр інформації. Наприклад, серед таких нюансів є зв’язки між актором та прецедентном. Кожен вид зв’язку символізую різний вид існування прецедента в контексті системи. Існує такий перелік зв’язків:

- Включення
    - Застосовується у тому випадку, якщо викликається прецедент наслідувач, коли викликається попередній прецедент. Наприклад, процес реєстрації включає в себе підтвердження користувачем адреси електронної пошти шляхом надання одноразового коду, який надсилається на указану адресу. Таким чином реєстрація не може

бути виконана без прецеденту “Підтвердження електронної пошти”, отже зв’язок між двома прецедентами має бути “Включення”

- Розширення
    - Застосовується у випадку, коли потрібно відобразити необов’язкові прециденти під час виклику попереднього прецидента, але доступні користувачеві. Наприклад, такі дії як збереження авторизаційних даних користувача на клієнтській стороні протягом якогось часу, або отримання маркетингової інформації від системи не є обов’язковими в процесі реєстрації, але доступні користувачеві, тому вони будуть пов’язані з прецидентом “Реєстрація” зв’язком “Розширення”
- Асоціація
    - Показує обов’язкові зв’язки між попереднім та наступним прецидентом та диктує правило того, що актор повинен виконати дії, що передбачає попередній прецидент перед переходом до наступного прецидента. Такі дії є обов’язковими до виконання, а отже не можуть бути невиконані. Прикладом такого зв’язку є процес авторизації користувача в системі. Для того, щоб користувач обов’язково потрапив до системи та отримав усі права взаємодії йому потрібно авторизуватись надавши логін та пароль для перевірки. Отже прециденти “Авторизація” та “Взаємодія з системою” повинні мати зв’язок “Асоціація”
- Узагальнення
    - Використовується у випадку, коли для користувача надається вибір того, по якому сценарію буде виконуватись програму із кінечного набору опцій. Такі зв’зяки часто використовують там, де не критично важлива безпосередньо дія, а важливіше саме результат виконання дії. Наприклад, процес оплати замовлення може відбуватись за двома сценаріями: оплата готівкою, оплата платіжною картою банку. У контексті виконання замовлення не настільки важливо, яким саме

способом користувач сплатить його, а важливо те, що оплату здійснено і результат досягнуто.

Діаграма прецедентів для системи, що розробляється (рис. 1.2):

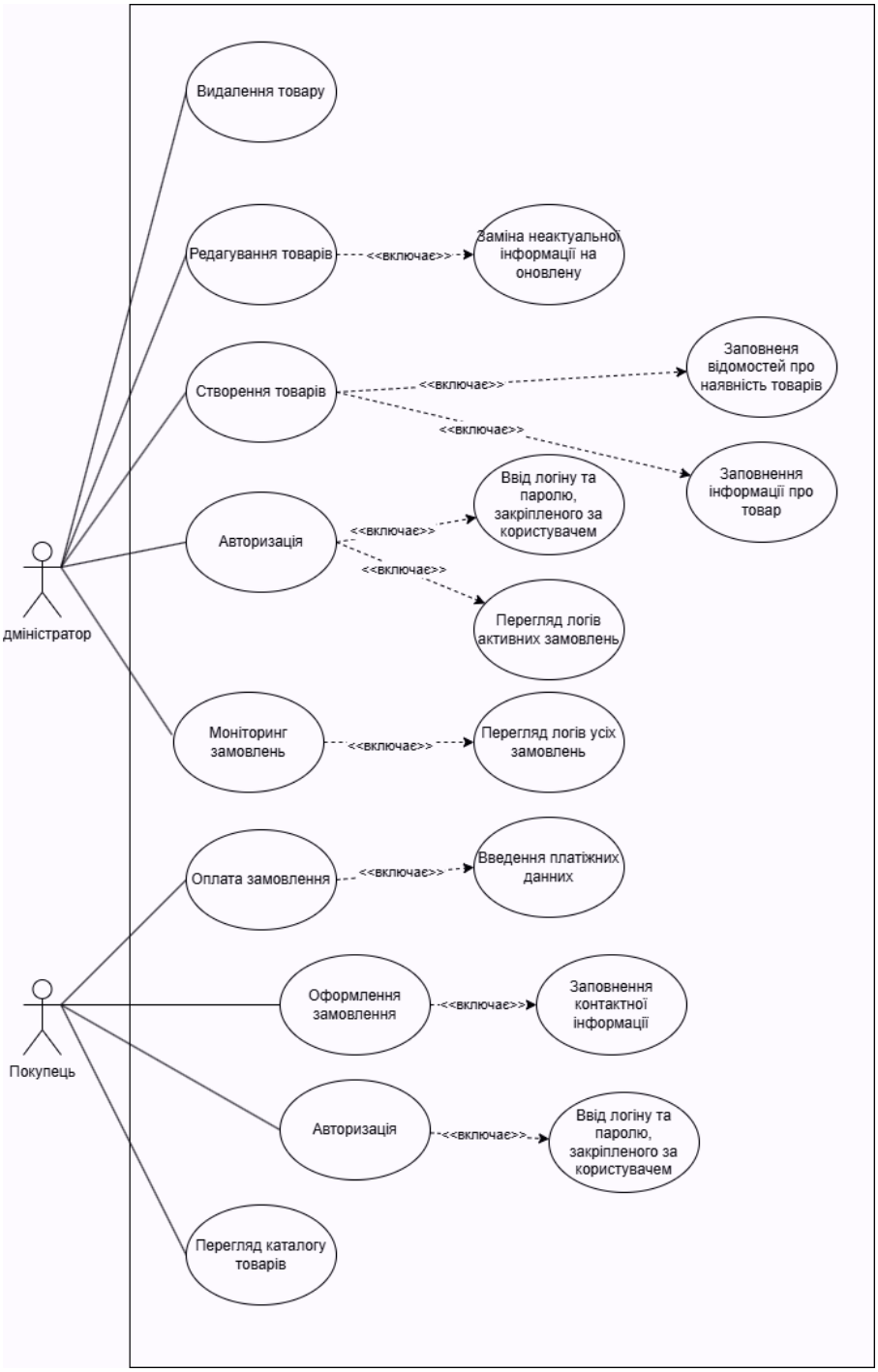


Рис. 1.2. Діаграма прецедентів

У наданій діаграмі (рис. 1.2) було зображено те, як актори можуть взаємодіяти із системою, представлені усі варіанти сценаріїв виконання певних дій. У рамках роботи над системою було визначено 2 ролі акторів: покупець,

адміністратор. Для кожного з акторів було показано перелік дій, які об’єднані у сценарії та занотовано відповідно до правил побудови діаграм.

1.4 Діаграма діяльності

Іншим видом Business process model документу, який з більшим зануренням описує діяльність в рамках певного процесу або більш загального бізнес процесу є діаграма діяльності. Діаграма діяльності також будується за правилами нотації UML, має на меті наочно продемонструвати логіку бізнес-процесів, логіку роботи алгоритмів, різного роду сценаріїв виконання програми. На відміну від діаграми прецедентів, де кожен прецедент являє собою цілу дію та може деталізуватись тільки шляхом назви прецеденту та типом зв’язку з іншими прецидентами – діаграма діяльності має суттєво більше заглиблення в предметну область бізнес процесу, за рахунок чого дозволяє більш детально вивчити його

Такі діаграми будуються при подальшому аналізі системи і через свою складність та зануреність у мікропроцеси не можуть бути побудованими на первинних етапах. Але в свою чергу такі діаграми є більш цінними з точки зору розуміння процесу зсередини, чіткого усвідомлення дій, наслідків, розгалуженості системи.

Через специфіку такі діаграми не особливо підходять для презентації замовника системи, бо є доволі навантаженими і містять переліку інформації, яка може бути не так цікавою замовнику, але є дуже цінною для подальшого технічного супроводу системи та наповненню її функціональністю. Цей вид нотації дозволяє досконало вникнути в алгоритми поведінки системи, що в свою чергу є дуже важливим для адаптації системи під певні потреби. Наприклад, якщо виникла потреба впровадити зміни або додати нові механіки взаємодії в процесі авторизації, розробник може звернутись до такої діаграми та зрозуміти, як саме йому варто впроваджувати ті, чи інші зміни. Розуміння цього в свою чергу дає вагомий результат в якості системи, бо розуміючи загальні принципи

тих, чи інших процесів новий функціонал або зміни у вже існуючому з меншою вірогідність спричинять проблеми у всіх процесах, які відбуваються надалі.

Такі діаграми, як було сказано вище будуються за принципами уніфікованої системи нотації UML, відповідно мають правила та елементи, які диктують те, як саме і чим саме будувати такі елементи. До елементів такої діаграми входять:

- Ініціалізація процесу
    - Точка початку процесу
- Виконувана дія
    - Виконання певної дії в процесі
- Перехід
    - Перехід від однієї виконуваної дії до іншої
- Нода прийняття рішення
    - Місце, де приймається рішення про напрямок виконання процесу в залежності від попередніх умов
- Нода мерджування або нода з’єднання
    - Місце зливання або об’єднання декількох варіацій процесу в один
- Паралельно-виконувані дії
    - Етап виконання, на якому від декількох дій, що виконуються незалежно один від одного очікується завершення
- Кінець процесу
    - Місце виходу з процесу

Набір певних дій, умов, та паралельних виконань об’єднується в один процес, який зображається на діаграмі.

Прикладом нотації процесів у вигляді UML діаграми діяльності є процес в системі, що розробляється є діаграма діяльності користувача (рис. 1.3):

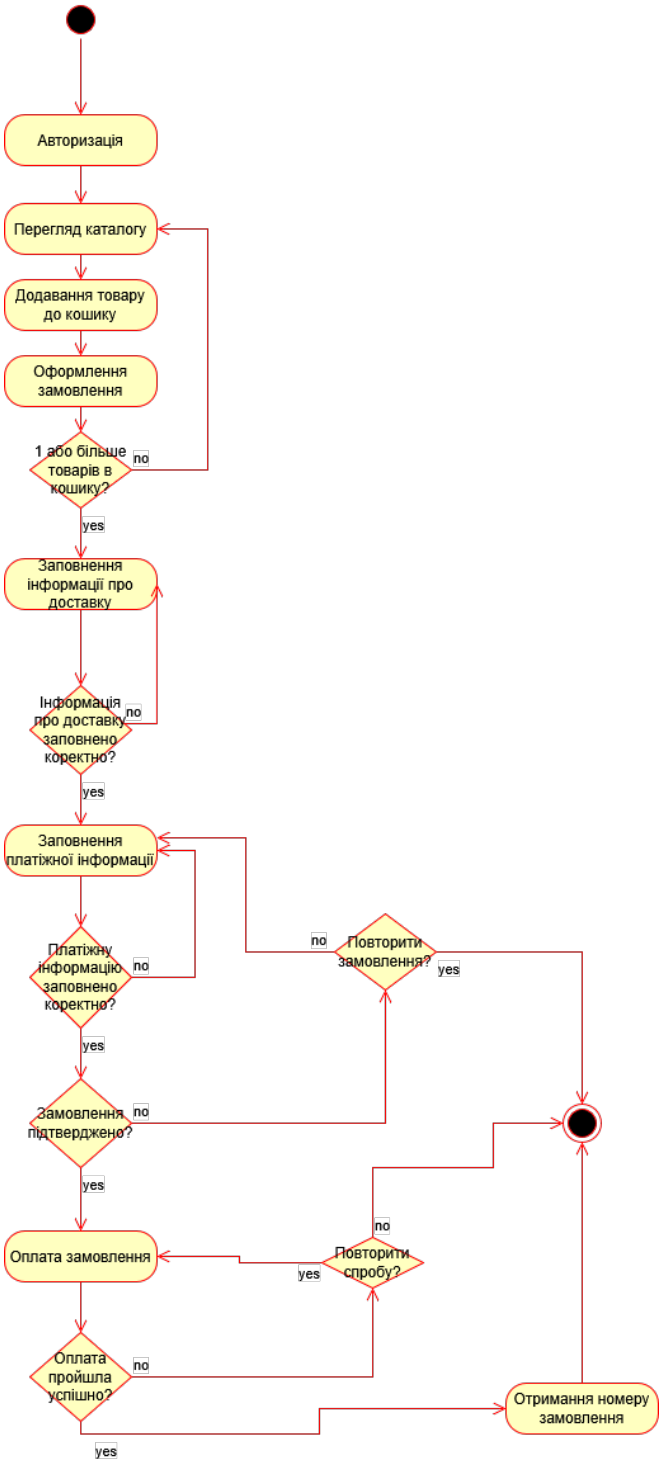


Рис. 1.3. Діаграма діяльності

1.5 Діаграма послідовності

Для кращого розуміння того, в який період часу яка дія відбувається, а також розуміння виконання процесів у хронометражі існує вид діаграми, який має назву “Діаграма послідовності”. Це ще один вид UML діаграми, який пизначений для відображення послідовності виконання дій у часі. За допомогою

такої діаграми можна краще зрозуміти, які ресурси та засоби в який момент часу буде задіювати система. Такий вид діаграм мінімізує ситуацію, в якій буде потреба у використанні одного і того ж ресурсу в один і той же період часу.

Мінімізація таких ситуацій зменшує кількість часу, яка потрібна спеціалісту для узгодження таких конфліктів інтересів між різними фрагментами програми. Не дивлячись на те, що сучасні технології дозволяють реалізувати доступ до одного ресурсу одночасно із двох точок інтересу, такі рішення потребують додаткових затрат часу і фінансів відповідно, але на практиці гостра необхідність цього доволі рідкісна. Тому для того, щоб не вирішувати проблеми, якої можна не створювати було запропоновано використовувати діаграму послідовності.

Як і будь яка інша діаграма UML, ця діаграма будується на основі певних принципів та має певні елементи, які узгоджені для кращої її читаємості. Елементами такої діаграми є:

- Сутність
    - Об’єкт, який існує в систему та “живе” певний проміжок часу
- Лінія життя
    - Проміжок часу, в який об’єкт є активним та несе корисне навантаження
- Запит
    - Процес інтеракції об’єкта з іншим об’єктом або об’єктами
- Дія
    - Спосіб, в який відбувається інтерація
- Зворотня дія
    - Процес, який передбачає передання результату від попередньої сутності наступній сутності
- Ініціалізація
    - Створення об’єкта в контексті виконання
- Деініціалізація
    - Видалення об’єкта з контексту виконання

Прикладом такої діаграми є діаграма послідовності (рис. 1.4) з розроблюваної системи:

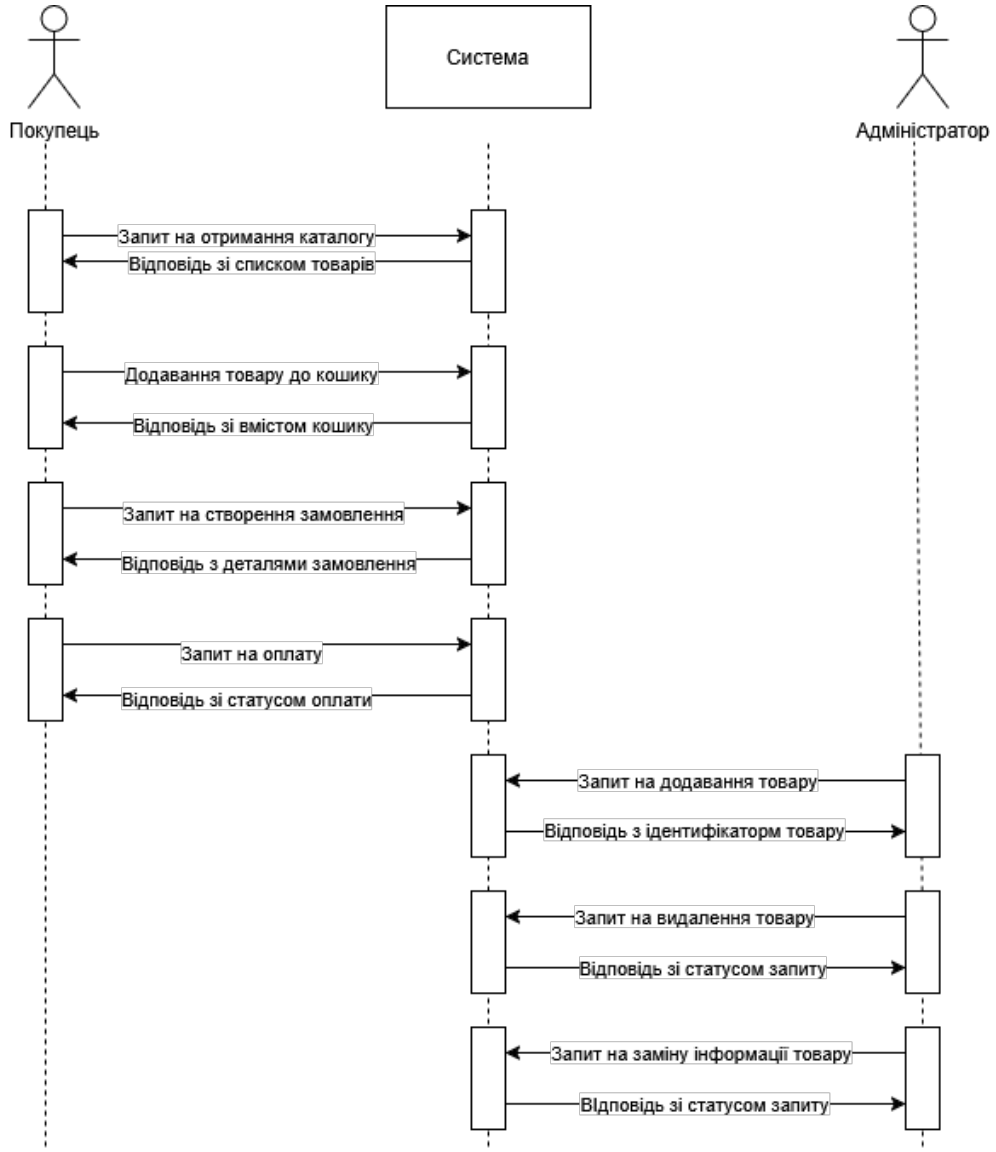


Рис. 1.4. Діаграма послідовності

Проаналізувавши таку діаграму стає чітко зрозуміло, який об’єкт або сутність є активною в певний момент часу, після кого та перед яким вступає в дію та дозволяє проаналізувати потік виконання програми.

РОЗДІЛ 2
ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Загальні відомості про ER-діаграму

Будь-який застосунок оперує певною кількістю даних для виконнання корисної роботи. Деякі дані отримуються програмою ззовні, наприклад від інших сторонніх сервісів, зчитування з файлу, або вводу даних від користувача.

Із збільшенням об’єму даних, якими в різний момент користується система виникла потреба зберігати ці дані у якийсь спосіб. Способів на данний момент існує велика кількість, але основні із них це:

- Оперативна пам’ять
    - Спосіб зберігання даних, який використовує оперативну пам’ять у якості місця для сховища даних. Такий спосіб був передбачений інджeнерами ще від моменту створення перших модулів оперативної пам’яті. Оперативна пам’ять це регістри пам’яті верхнього рівня, які використовуються процесором для зберігання інформації, яка йому потрібна у швидкому доступі. Таким чином оперативна пам’ять знаходить у прямому підпорядкуванні центрального процесора, та є дуже швидким місцем збереження даних. У данному випадку швидкість це головна перевага такого методу. Так як оперативна пам’ять є у прямомум підпорядкуванні процесора процес обміна даних відбувається дуже швидко, а різниця між постійною і оперативною пам’яттю у швидкості отримання даних в часовому еквіваленті може становити на віть 10, або 20 разів. Але це місце для зберігання даних має декілька суттєвих недоліків. Перший, і у деяких моментах критичний – природня властивість оперативної пам’яті скидати усі значення, які були збережені в регістрах пам’яті після перезавантаження. Коли вимикається хост машина – усі дані з оперативної пам’яті очищуються та не можуть бути повернуті в регістри після повторного увімкнення. Окрім того, оперативна

пам’ять, як правило має значно менший об’єм у порівнянні із постійною пам’яттю, що на певному етапі використання програми може бути критичним. На основі такого підходу будуються різноманітні алгоритми покращення швидкості програми, наприклад алгоритми кешування, але на практиці мало хто працює із оперативною пам’яттю безпосередньо, бо для організації таких механізмів існують рішення, як Redis, які дають набагато зручніший інтерфейс

- Постійна пам’ять
    - Зберігання даних програми у постійній пам’яті комп’ютера також є одним із старих підходів для керування данними програми. Постійна пам’ять комп’ютера – це вид пам’яті, який комп’ютер використовує для зберігання файлів, швидкість доступу до яких не настільки важлива у короткому проміжку часу. Наприклад, такими данними можуть бути статичні файли програм: картинши, шрифти, розмітка і так далі. Цей метод зберігання є значно менш ефективним з точки зору часу, ніж наприклад оперативна пам’ять, бо для того, щоб зберегти дані наприклад на жорсткий диск операційна система змушена провести індексацію усього простору пам’яті, виділити місце під конкретні дані, а вже потім записати їх до сховища. Не дивлячись на відносну повільність цього методу зберігання – він має значні переваги, нприклад дані, що записані до такого сховища мають постійне місце зберігання і будуть зберігатись там до поки, їх не буде видалено адресно. Це означає, що для того, щоб видалити значення певних комірок пам’яті комп’ютеру потрібно дати безпосередню команду для цього, а відповідно можна бути впевненим, що дані будуть там надійно збережені увесь час, поки ми будемо їх потребувати. Дивлячись на те, що постійна пам’ять повільна інджeнерами були винайдено різноманітні алгоритми, для підвищення швидкості роботи пам’яті, як програмні, так і апаратні.

Наприклад, для того, щоб така пам’ять швидше працювало створили індексації всього дискового об’єму, а для того, щоб методи взаємодії із постійною пам’яттю у парі “програма/програма” були ще ефективніші – побудовано різні програмні рішення для зберігання даних. Наприклад, одним із таких рішень є СУБД, яка по суті являється засобом менеджменту БД. БД в сою чергу є оптимізованим місцем для зберігання інформації. СУБД, як і самі БД є дуже різні за своєю суттю, функціоналом та можливостями, але об’єднує їх те, що вони покликані поліпшити взаємодію програми із великою кількістю даних, які потрібно постійно зберігати.

- Хмарні сервіси
    - Новаторський підхід до роботи із зберіганням даних програми, який заключається у використанні виробничих потужностей хмарного сервісу, включно із дисковим простором для зберігання даних. Такі сервіси почали з’являтись відносно недавно, але вони є гарною заміною вже зчисним рішенням із встановленням та розгортанням СУБД локально. Такі сервіси надають інтерфейс для роботи із СУБД, при цьому розгортаючи усю необхідну архітектуру на своїй стороні. Таким чином система стає агностичною до того, де саме вона буде використовуватись, бо доступ до неї реалізується через інтернет. Таким чином єдиним фактором, який може повпливати на роботу із такою СУБД є швидкість з’єднання, яка як правило на серверних машинах є великою і стабільною.

Враховуючи велику варіативність методів зберігання, велике різноманіття керуючих систем базами даних, потенційну велику складність та заплутаність безпосередньо баз даних без належного опису того, які БД має властивості, можливо сутності та зв’язки між сутностями – базами даних стало складно керувати, змінювати та розширювати. Для того, щоб охопити усі види БД, описувати усі зв’язки, сутності, залежності, функції бази даних потрібно було створити універсальний метод нотації таких спеціалізованих документів. Для

цього було створено ER-діаграму, яка має правила та елементи, за допомогою чого є загально-прийнятою для опису такого роду аспектів програми.

ER-діаграма – це високорівнева діаграма опису сутностей та відношення між ними у базі даних. Повна назва такої діаграми “entity relations diagram”. Виходячи із назви, у такій діаграмі існує дві ключових абстракції: сутність та зв’язок. Сутність репрезентує певну операційну модель в системі, яка має певний набір атрибутів. Так як у великих системах сутності не можуть існувати окремо – між ними створюються певні зв’язки. Зв’язки існують різні, наприклад “частина/ціле”, де одна сутність є частиною іншої сутності, і коли батьківська сутність знищується – знищуються і її частини. Окрім того зв’язки також можуть репрезентувати залежність полів одного від іншого. Наприклад, структура foreignkey, яка пов’язує певний атрибут однієї сутності із іншим атрибутом іншої сутності, таким чином встановлюючи певний вид зв’язку між ними, який, як уже раніше було вказано може мати різний характер і відповідно вплив.

2.2 Побудова ER-діаграми

Для того, аби побудувати ER-діаграму, потрібно досконало розуміти те, які в контексті системи існують сутності, які кожна із цих сутностей має властивості, зв’язки та залежності. Без розуміння цих аспектів неможливо створити діаграму, яка буде відповідати вигляду, інформативності, та структурі БД.

Немає чіткого декларування того, на якому саме етапі варто будувати таку діаграму. Кожен індивідуальний розробник або команда розробників вирішує це безпосередньо за потреби та на основі свого досвіду. При цьому, існує дві концепції того, як створювати таку діаграму: на основі вже існуючої бази даних або на перед фактичним створенням бази даних. Розберемо обидва підходи:

- Створення на основі вже існуючої бази даних
    - Сучасні інструмент дозволяють дуже ефективно виконувати задачу створення ER-діаграми автоматизовано. Такі інструменти візуалізації існують майже для кожної СУБД: MSSQL, MySQL, PostgreSQL, та багатьох інших. Надавши файл або запит для

створення таблиць написаний на мові запитів SQL, або навіть моделі із сучасних ORM такі системи створюють візуалізацію із урахуванням усіх аспектів: зв’язків між сутностями, поля сутностей, залежності і так далі. Це дуже зручно, при цьому використання таких інструменті не потребує значної затрати часу чи сил. Також, це можливість відносно просто виправити недопрацювання на попердніх етапах розробки продукту, що також є чудовою можливістю для покращення якості продутку

- Створення діаграми перед фактичною реалізацією БД
    - Чудовим підходом є створення подібного роду документації перед фактичною реалізацією БД безпосередньо. Для того, щоб створити таку модель потрібно витратити істотний час на аналіз того, як система буде працювати із данними, визначити типи даних, які будуть існувати в контексті системи, яким чином вони будуть обмежуватись (наприклад можливість встановлення значення NULL для певних полів, встановлення значення за замовчуванням, установка констрейнта unique на певне значення сутності і так далі). Потрібно пропрацювати також аспект того, як дані будуть поєднуватись між собою, тобто які види зв’язків будуть існувати в системі, яким чинох їх краще організувати, а також побудувати ієрархічну систему, яка буде належним чином обмежувати кожен шар ієрархії

Усвідомивши різницю того, на яком етапі будується діаграма можна зрозуміти принципову різницю і принципово різні потенційні наслідки кожного із підходів. В рамках роботи над системою, що розробляється було обрано перший підхід. База даних було розпланована та занотована у довільній формі, на основі чого потіс створювалась реалізацію даних у вигляді моделей ORM. Маючи фактичну реалізацію абстракції моделей даних було створено базу даних, яка унаслідувала усі моделі, властивості із їх залежностями та зв’язками. Такий алгоритм свторення та розгортання є дещо оптимізованим.

Шляхом того, що база даних була описана у довільній формі – було присутнє розуміння того, які саме дані повинні міститись у кожній відповідній сутності, було продумано правила зберігання цих даних, зв’язки між ними. На основі цього опису за допомогою ORM(Object-Relation Mapper) для мови програмуваннння високого рівня Python на основі фреймворку SQLAlchemy було створено відповідні моделі за правилами нотації моделей даних для конкретної технології. Таким чином, процес створення таблиць спрощується, бо ORM сама створює запити до БД на основі тих моделей, які були описані, сама займається валідацією даних як під час створення БД, так і під час роботи з нею, та має набагато зручніший вигляд та є простішим до розуміння. Також, на відміну від написання SQL запиту для створення таблиць самотушки, використання ORM має перевагу у вигляді того, що більшість сучасних рішень уже мають вбудовані інтерфейсти та інтерпритатори для різних СУБД, за рахунок чого немає потреби заглиблюватись в різні діаглекти чи особливості створення БД в контексті різних СУБД.

На рисунку 2.1 зображено ER-діаграму, що демонструє структуру бази даних із усіма сутностями та зв’язками між ними.

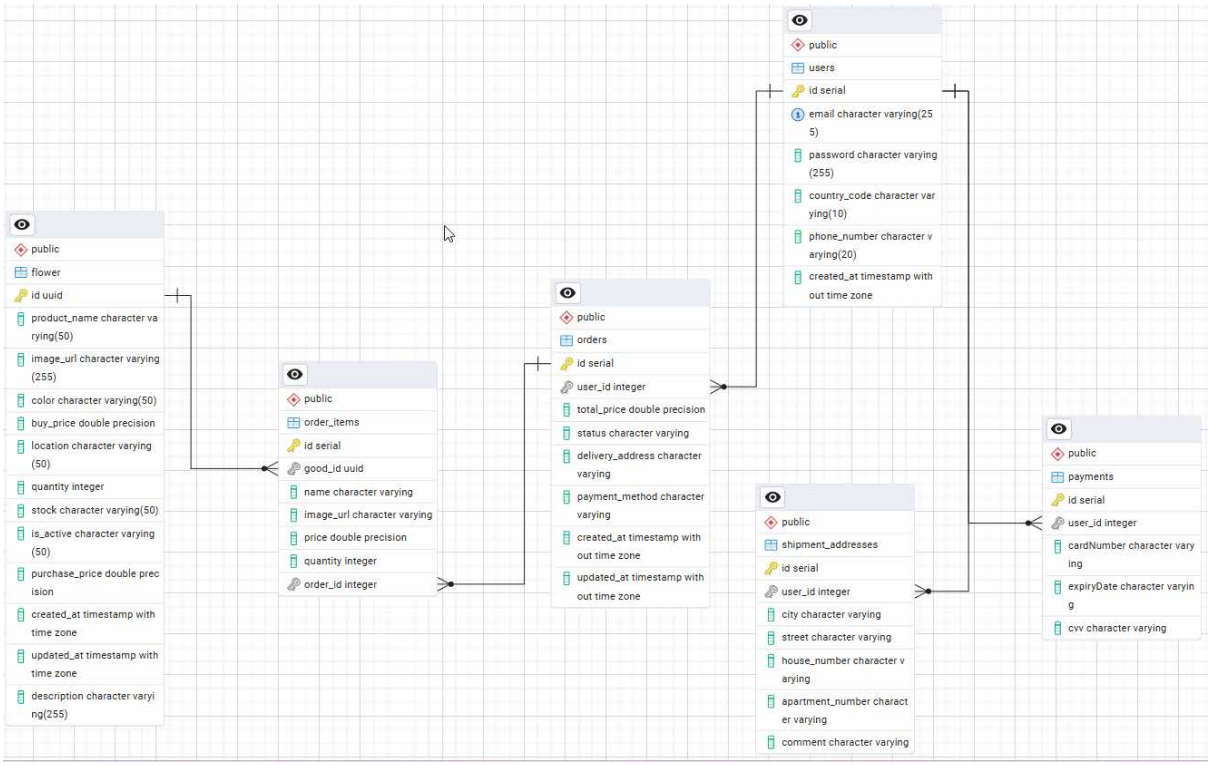


Рис. 2.1. Діаграма зв’язків сутностей (ER-діаграма) бази даних

2.3 Вибір та обгрунтування СУБД

Вибір системи управління базою даних є ключовим в контексті роботи з данними. Неправильний вибір на початку може спровокувати багато проблем у подальшому використанні та розробці системи. Категорично важливо знайти найкраще співвідношення стабільності, функціональності, зручності СУБД до потреб проєкту.

Більшість сучасних СУБД дуже схожі між собою, але при цьому все ще мають невеликі відмінності. Наприклад, деякі СУБД дозволяють визначати свої типи даних, як наприклад SQLite, а деякі ні, як наприклад PostgreSQL не дозволяє визначити свій тип даних напряму. При цьому, не означає, що варто обрати SQLite, бо можливо такої потреби взагалі не буде в рамках проєкту. До вибору СУБД потрібно підходити із розумінням повного переліку того, що від неї потрібно.

Не варто також оператись виключно на простоту використання та розгортання. Часто складність в базовому налаштуванні системи відлякує потенційних користувачів. Навіть враховуючи, що деякі СУБД потребують деякого часу та знань для їх первинного налаштування – у переважній більшості випадків ці складності компенсуються подальними перевагами, такими як: забезпечення унікальних вимог, придатність до змін, адаптивність до умов середовища.

Відповідно до принципу, який був описаний вище для вибору СУБД потрібно встановити вимоги до неї. Так як у рамках системи буде відбуватись робота із чутливою інформацією – потрібне щось надійне з точки зору безпеки, із можливість задавати авторизаційні дані, можливісттю створювати авторизаційні ролі. Друга вимога базується на специфіці сфери, а саме відсутність консистентного потоку даних, що у свою чергу не дає права на втрату даних. Якщо говорити простіше, то у випадку, якщо наприклад СУБД буде перевантажена, зайнята іншим ресурсом, припинить відповідати – ми втратимо замовлення, але окрім цього невдалий запит може пошкодити частину

інформації, яка вже зберігалась. Відповідно до цього критерію для системи потрібне щось, що буде мати захист від таких випадків.

Для вибору СУБД перш за все потрібно встановити формат, у якому вона буде працювати. Проаналізувавши усі сутності присутні у нашій системі можна виявити, що всі вони є структурованими, а отже нам потрібно використовувати відповідно структуровану модель збереження даних. Для такого випадку існують SQL бази даних. Специфіка таких баз даних у тому, що дані всередині БД представлені у вигляді таблиці із певним переліком рядків та колонок. Частіше за все, такі бази даних є реляційними. Реляційність це здатність встановлювати зв’язки всередині бази даних між різними таблиця, відповідно до цього і між сутностями, бо кожна таблиця відображає певну сутність, або несе виключно утилітарний характер.

Відповідно до нашого аналізу було встановлено, що нам потрібна СУБД, яка є реляційною, також вона повинна бути призначена для того, щоб зберігати структуровану інформацію. Чому так важливо саме те, що БД повинна бути призначена для зберігання структурованої інформації? Перш за все треба зазначити, що існують системи управління базами даних, які можуть зберігати і неструктуровані дані, і навіть дані у векторному вигляді.

Теоретично, майже на будь якому типі СУБД можна зберегти будь які дані. На практиці існує велика вірогідність того, що таке рішення викличе чимало проблем. Наприклад, БД, яка призначена для зберігання неструктурованої інформації, тобто тієї, у формі якої не можна бути впевненим працюють набагато повільніше, ніж співставна структурована БД. Ми можемо зберегти сутність “Користувач” із нашої системи наприклад у вигляді JSON нотації в NoSQL СУБД MongoDB. Але якщо уявити ситуацію, в якій потрібно буде отримати конкретного користувача із великого переліку користувачів, то наприклад MongoDB повинна буде відкрити кожен файл із коллекції, отримати звідти значення певного ключа, після чого перевірити його на співпадіння вимогам та якщо він не підходить – пропустити файл і йти до наступного, і так до тих пір, поки відповідний файл не буде знайдено, або не знайдено і повернуо помилку.

Натомість СУБД зі структурованим виглядом зберігання даних, наприклад MySQL має усі значення індексовані, і для того, щоб виконати аналогічну задачу СУБД звернеться до конкретної таблиці бази даних, перевірить усі значення конкретного ствобця таблиці, який відповідає умовам, після чого знайде у ньому відповідний і поверне його.

Для того щоб обрати якусь конкретну СУБД, яка б підійшла під наші вимоги було створено перелік СУБД, які потенційно могли б задовільнити наші вимоги: MSSQL, MySQL, PostgreSQL. Це три найпопулярніших СУБД, які найчастіше використовуються якщо є потреба у збережені структурованої інформації. Загалом кожна із них підійшла б для збереження, але в проєкті де дані грають ключову роль потрібно обрати серед варіантів ту, яка буде найкраще підходити під усі вимоги. Тому для розуміння цього аспекту потрібно порівняти усі СУБД.

- MSSQL
    - Є дуже популярною СУБД. Входить до числа найбільш використовуваних. Ця СУБД є дуже популярною для систем, які передбачають велику кількість запитів, бо здатна витримувати велику кількість запитів, легко маштабується горизонтально (шляхом збільшення кількості серверів, які приймають участь в обробці запитів) та має велику спільному, яка стрімко розвивається. Сама по собі СУБД хоча і закрита, тобто не має відкритого коду, але є доволі придатною для її кастомізування. Це означає, що переважну кількість специфічних потреб вона може закрити. Також, що є не менш важливим – вона має надзвичайно велику кількість вбудованих інструментів для ефективного менеджмента ресурсів. Серед таких, наприклад:
        - Інструменти окрестрації серверів MSSQL
        - Оптимізація внутрішньої логіки роботи СУБД, за рахунок чого є швидкою

- Інструменти для виконання глибоких пошуків, наприклад інструмент Full Find Search
- Має вбудований інструмент для налаштування точки входу, іншими словами налаштування гейтвея, який дозволяє для багатьох серверів обробників створити єдину адресу, на яку будуть поступати запити, після чого їхня адресація буде відбуватись уже на стороні безпосередньо СУБД, що є дуже зручним для систем, де є велика кількість серверів із MSSQL
- Здатність системи генерувати звіти по кількості запитів, по вмісту таблиць, по кількості використаних ресурсів

У випадку нашої системи ця СУБД не буде найкращим рішенням із декількох причин:

- Сервер, який обробляє запити є доволі великим, що у свою чергу потребує значної кількості серверного дискового простору, а також є доволі ресурсоємним, що на практиці означає потребу у значних серверних ресурсах. Так як для розповсюдження системи було обрано за методом дистрибуції, тобто для кожного клієнта(підприємця) буде створюватись окремий екземпляр системи – в результаті сервер, на якому буде розгортатись система буде дорожче в утриманні, що буде відчутно на певному проміжку часу використання системи
- СУБД не використовує транзакції одразу. Не дивлячись на те, що такий механізм існує в рамках цієї системи контролю, при стандартній конфігурації він не буде використовуватись. Відповідно, його потрібно розробити окремо і кожен раз, коли буде виконуватись запит його потрібно буде застосовувати окремо. Це не є критичним недоліком, бо загалом така можливість присутня, але є рішення, де це контролювати не потрібно, а відповідно у цьому аспекті MSSQL не виглядає як краще рішення. У нас є гостра потреба у тому, щоб запити виконувались як транзакції через те, що нам важливо бути впевненими в успішному виконанні запиту.

- Для цільового використання доведеться витратити певну кількість часу на те, щоб налаштувати сервер, який буде здатний обробляти запити у такий спосіб, як потрібно для системи, відповідно це буде коштувати дорожче, ніж використати рішення, яке одразу пропонує усі необхідні налаштування, щоб оптимізувати витрати та складність налаштування системи при дистрибуції
- MySQL
    - Система управління базами даних, яка є не менш популярною. Широко використовується для комерційних цілей. Отримала свою популярність за ряд технічних рішень та семантичних рішень, які були впроваджені розробниками. Це СУБД, яка характеризується простотою у використанні, але при цьому дещо меншим функціоналом, ніж наприклад MSSQL, хоча і в порівнянні із попередньою має ряд суттєвих переваг, які її якісно вирізняють. Перш за все через наявність відкритої комерційної ліцензії MySQL не потрібно купувати навіть при умови, що вона буде використовуватись комерційно, а це позитивно впливає на економічну складову кожного проєкту. Також, вона має відкритий код. Не дивлячись на те, що усі вище перераховані дистрибютори програмного забезпечення є доброчесними – наявність можливості перевірити код та зануритись у тонкості його побудови щоб краще розуміти його поведінку залежно від сценарію є відчутним плюсом для продуктів, які мають якусь чутливу, або цінну інформацію. Не дивлячись на те, що у порівнянні із MSSQL наприклад вона має трохи менший перелік вбудованих функцій, їх достатньо для реалізації більшості вимог. Окрім того, за рахунок своєї відкритості MySQL стала платформою для великої кількості сторонніх гілок розвитку СУБД. Розробники або цілі команди розробників активно вносять свій вклад у розвиток системи, створюючи доповнення до оригінальної СУБД, або навіть створюючи окремі гілки розвитку, як

це сталось у випадку MariaDB, або InnoDB. Це несумнівно є великою перевагою. Система також підтримує транзакції, процедури, функції. У порівняння із MSSQL це рішення більше задовільняє наші вимоги, але при цьому не є найкращим у нашому випадку. Є нюанси, які не грають на руку цій СУБД, наприклад таж відсутність транзакцій при звичайних запитах, натомість їх потрібно налаштовувати окремо. Але перевагою цієї СУБД є те, що вона доволі легка для системи, не займає так багато ресурсів та місця, як MSSQL.

- PostgreSQL
    - Сучасна система курвання базами даних, яка найкраще підходить під наші потреби. Вона включає в себе усе найкраще із попередніх двох конкурентів. Наприклад, вона є дуже добре адаптованою під кросс-платформеність, відповідно однаково добре працює на UNIX та Windows системах. Так як у нашому випадку буде використовуватись операційна система UNIX, а точніше Linux через те, що для дистрибуції він зручніший, краще підходить для адміністрування серверів, є менш затратним з точки зору ресурсів серверу – це є суттєво перевагою перед рішеннями з відсутністю адаптації, або частковою адаптацією для систем на UNIX. СУБД також має відкритий код, за рахунок чого має переваги, які має будь яка open-source рішення – довіра, можливість зануритись в поведінку системи, гарантії безпеки. Через це також отримала велику частину ринку інформаційних технологій, які активно працюють з данними. Має вбудований алгоритм транзакцій, завдяки чому є дуже надійною. Важливо зазначити, що базово працює алгоритм, який забезпечує збереження даних до отримання відомості про збереження даних. На практиці, якщо користувач такої СУБД отримує інформацію про те, що відбувався запит і виконався успішно, то він може бути впевненим у тому, що всі операції над данними провелись успішно. Також, у системі, що розробляється

використовується JSON як основний спосіб нотації даних. У свою чергу представлена СУБД має чудово адаптований парсер для роботи із цим типом нотації. Є нативна підтримка процедур із використанням усіх популярних мов програмування високого рівня, за рахунок чого підходить для більшої кількості розробників. Використовує велику кількість засобів оптимізації роботи, що відображається на швидкості та якості роботи СУБД

Опираючись на проведене порівняння СУБД, провівши аналіз вимог було прийнято рішення про використання PostgreSQL як основної структури для збереження даних системи, а також проведення різного роду маніпуляцій над данними

2.4 Створення СУБД

Провівши аналіз доступних до використання СУБД та обравши основну для проєкту було виконано наступний крок в побудові системи, а саме розробка сутностей в системі для створення бази даних. Це процес, який потребує глибокого розуміння процесу та усіх даних, які потрібні для виконання процесу.

У системі, судячи із її опису буде дві основні сутності: покупець та адміністратор. Ці дві головні сутності, які можуть виконувати певні дії в системі. Розділення користувацьких прав по ролям має велику кількість переваг: можливість обмежувати доступ відповідно до рівня допуску ролі, вибудова ієрархічно правильної моделі взаємодії кожної ролі із системою та іншими її учасниками, надання певних привілегій та багато інших функціональних можливостей надає такий технологічний підхід. Реалізація ролей може бути виконана різними методами, наприклад шляхом надання інформації про роль в авторизаційному токені користувача, або розділення роутів між різними ролями для різних потреб, або використання спеціальних методів автентифікації, за допомогою яких можна реалізувати такий функціонал.

Покупець має певний перелік доступних опцій та маршрутів, які відмінні від переліку доступних опцій та маршрутів для адміністратора, отже в контексті

системи важливо розділити ці дві ролі таким чином, щоб вони не пересікались між собою. З цією ціллю було створено дві окремі таблиці під кожну із сутностей (рис. 2.2 та рис. 2.3).

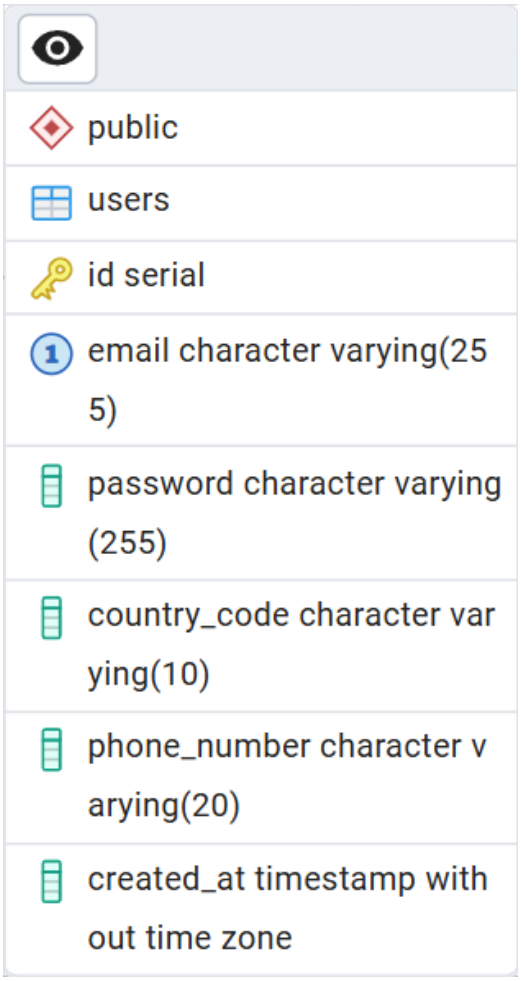


Рис. 2.2. Таблиця, яка репрезентує сутність користувача

Сутність користувача (рис. 2.2) має набір властивостей, які представлені на Рис.4. Серед них так:

- id – цифровий унікальний ідентифікатор, який також є первинним ключем в таблиці
- email – електронна адреса користувача, рядкове значення, для якого також застосовується правило unique, що означає неможливість повторення значення у таблиці. На одну електронну адресу користувач може зареєструвати тільки один обліковий запис
- password – пароль користувача, який зберігається у вигляді рядкового значення. Не має правил, що застосовуються до цього поля

- country_code – код країни оператора мобільного зв’язку, на який користува має намір зареєструвати обліковий запис. Правил не застосовується, відтак користувач може зареєструвати на один і той самий номер телефону декілька акаунтів. Така логіка застосована з міркувань того, що це не є компонентом авторизаційного алгоритму, тому не важливо чи використовується унікальний номер чи ні
- phone_number – безпосередньо номер телефону, який користувач вказав при риєстрації облікового запису. Використовується тільки як канал зв’язку між покупцем та продавцем
- created_at – поле у форматі datetime, яке має запрограмоване стандартне значення на рівні ORM, яке дорівнює datetime.now() із вбудованої бібліотеки datetime. Дозволяє відслідкувати час, коли користувач зареєструвався в системі. Значення цього поля може використовуватись як у маркетингових цілях, наприклад для сповіщення старих користувачів, які вже певний час користуються послугами магазину і так далі

Після створення сутності покупця перейдемо до створення сутності адміністратора. Як було сказано раніше, менеджмент ролей буде виконано за рахунок розділення потоків для кожної ролі. Тобто, у адміністратора буде свій доступний набір маршрутів, якими він може скористатись, серед яких адмін панель, окрема сторінка логування, відсутність сторінки реєстрації у веб інтерфейсі, доступ до маршруту панелі із усіма замовлення. У користувача буде доступний каталог, профіль, корзина. Відповідно до цієї логіки сутності в таблиці також повинні бути відокремлені одна від одної, отже для адміністратора повинна бути створена окрема таблиця (рис. 2.3).

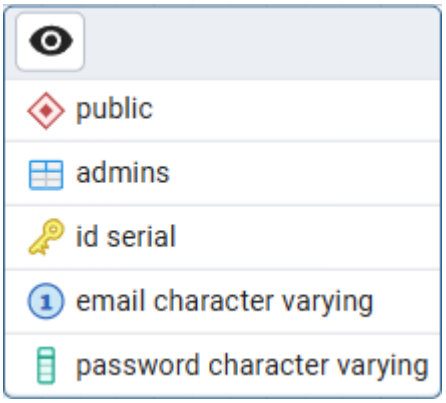


Рис. 2.3. Таблиця сутності адміністратора

Через специфіку створення адміністратора, а саме те, що адміністратора можна створити виключно шляхом активації скрипта utills.createAdmin сутності адміністратор потрібно мати абсолютний мінімум полів.

- id – унікальний цифровий ідентифікатор адміністратора, який також є приватним ключем для таблиці
- emai – lелектронна адреса користувача, рядкове значення, для якого також застосовується правило unique, що означає неможливість повторення значення у таблиці. На одну електронну адресу користувач може зареєструвати тільки один обліковий запис
- password – пароль користувача, який зберігається у вигляді рядкового значення. Не має правил, що застосовуються до цього поля

Тепер, після того як основні сутності було створено можна перейти до тих сутностей, якими вони будуть оперувати адміністратор та користувач. У рамках розробки інтернет магазину ключовим є товар, який представлений на вітрині. У нашому випадку товаром є квіти. Отже, треба визначити набір полів, якими буде володіти сутність товару (рис. 2.4).

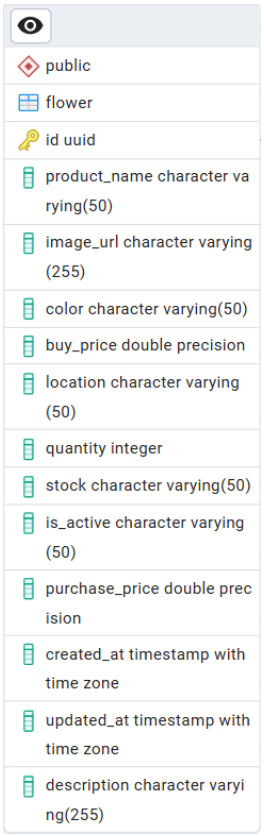


Рис. 2.4. Репрезентація сутності “Товар” в базі даних

У сутності товар є багато допоміжної інформації, яка має тільки системне використання, наприклад поле buy_price, яке відповідає за зберігання інформації про закупочну ціну товару. Перелік полів сутностей:

- id – унікальний цифровий ідентифікатор товару, який також є приватним ключем для таблиці
- product_name – назва відповідного товару, рядкове значення
- image_url – посилання на фото товару. За допомогою цього методу додавання фотографій можна суттєво спростити процес додавання фотографій для товару. Натомість, він має і недостаток у вигляді того, що такий метод є залежним від сторонніх хмарних схофищ, тому варто обирати їх дуже ретельно. Але це додає варіативності в джерелах додавання інформації і економить місце на диску
- color – колір товару, що додається
- buy_price – ціна закупки товару. Не відображається покупцеві, але дозволяє підраховувати аналітику магазину

- location – місце товару, рядкове поле, зберігає інформацію про те, де знаходиться товар
- quantity – кількість доступного товару. Дозволяє відслідковувати складські запаси. Зберігається у вигляді цілого числа та не показується користувачеві
- stock – кількісто товару, яка доступна до продажу. Дозволяє адміністратору магазину відслідковувати кількість товару та залишати буферну кількість за потреби. Не відображається покупцеві
- is_active – булеве значення. На основі цього поля фільтруються товари. Товари, що мають негативне значення не відображаються користувачеві
- purchase_price – ціна, за якою реалізується товар. Семантично повинно бути більшим за значення поля buy_price, але також дозволяється поставити нижче за потреби
- created_at – поле у форматі datetime, яке має запрограмоване стандартне значення на рівні ORM, яке дорівнює datetime.now() із вбудованої бібліотеки datetime. Відображає дату та час, коли товар було створено, не відображається для користувача
- updated_at – поле у форматі datetime, яке має запрограмоване стандартне значення на рівні ORM, яке дорівнює datetime.now() із вбудованої бібліотеки datetime. Відображає дату останньої зміни товару. Дозвлояє аналізувати час, коли останні раз був куплений товар, не відображається для користувача
- description – опис товару. Відображається користувачеві як опис товару

Для того, щоб користувач міг зробити замовлення потрібно створити сутність, яка б сегрегувала в собі усі необхідні поля та відображала повну ємну інформацію про замовлення. В рамках розроблюваної системи такою сутністю є таблиця “orders” (рис. 2.5)

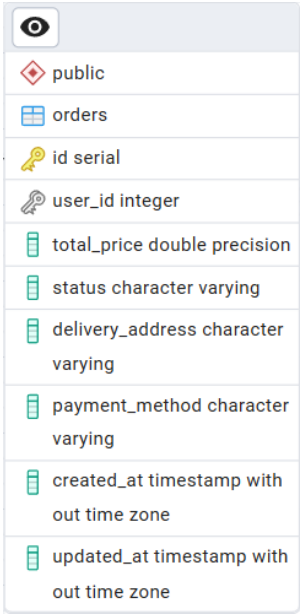


Рис. 2.5. Репрезентація сутності замовлення

Сутність, яка сегрегує в собі усі необхідні дані про замовлення. Опис сутності:

- id – унікальний цифровий ідентифікатор замовлення, який також є приватним ключем для таблиці
- user_id – унікальний цифровий ідентифікатор користувача, який виступає зовнішнім ключем та встановлює залежність між замовленням та користувачем, який зробив замовлення. Завдяки такому зв’язку можна зручно оперувати замовленнями, які користувач виконав без прямої взаємодії із сутністю замовлення, а оперуючи сутністю користувача
- total_price – сума за усе замовлення
- status – статус замовлення, який зберігається у вигляді рядкового значення
- delivery_address – адреса доставки замовлення, яку взав користувач під час замовлення. Зберігається у вигляді рядкового значення
- payment_method – метод оплати, який вказав користувач, зберігається у вигляді рядкового значення
- created_at – поле у форматі datetime, яке має запрограмоване стандартне значення на рівні ORM, яке дорівнює datetime.now() із вбудованої бібліотеки datetime. Відображає дату та час, коли замовлення було створено, не відображається для користувача

- updated_at – поле у форматі datetime, яке має запрограмоване стандартне значення на рівні ORM, яке дорівнює datetime.now() із вбудованої бібліотеки datetime. Відображає дату останньої зміни замовлення. Дозвлояє аналізувати час, коли останні раз був змінений статус замовлення, відображається для користувача

Окрім безпосередньої сутності замовлення в системі існує допоміжна сутність, в якій зберігаються усі необхідні дані про товари, що були замовлені. Таке сепарування сутності замовлення і елементів замовлення було створено для того, щоб зберігати принцип єдиної відповідальності, згідно якого кожна сутність повинна містити тільки ту інформацію, яка відноситься до неї та має конкретне призначення. З цього міркування було створено сутність “order_items” (рис. 2.6)

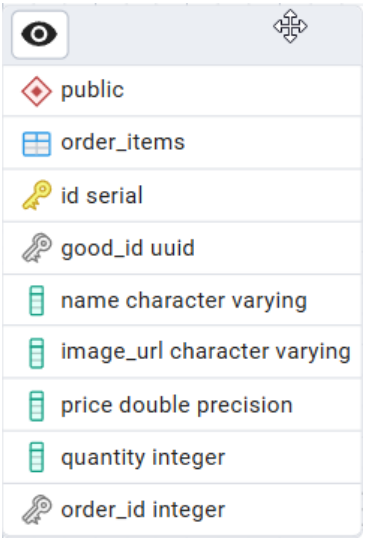


Рис. 2.6. Сутність “order_items”

Описи сутності:

- id – унікальний цифровий ідентифікатор замовлення, який також є приватним ключем для таблиці
- good_id – зовнішній ключ товару, який замовив покупець
- name – назва товару, яки замовив покупець
- image_url – посилання на зображення товару
- price – ціна, яка була вказана на момент покупки
- quantity – кількість, яку товару, яку замовили

РОЗДІЛ 3
ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Вибір мови програмування, на якій буде створюватися застосунок, диктує весь вектор подальшої взаємодії із додатком, простоту його створення, розширення та підтримки. Мов програмування високого рівня існує велика кількість, і їх також потрібно підбирати відповідно до вимог та бюджету команди. Через специфіку кожної мови та її складність, ціна реалізації проєкту може дуже сильно варіюватися. Наприклад, розробка веб-застосунку із застосуванням мови програмування C++ буде коштувати дорожче, ніж розробка аналогічного за функціоналом на Node.js. Таке явище пояснюється тим, що на ринку праці серед програмістів, які працюють з певною мовою, може варіюватися конкуренція, рівень затребуваності, середня кваліфікація тощо.

Окрім вартості розробки, можуть варіюватися і терміни, оскільки на одній мові реалізація певного функціоналу майже не потребує написання коду вручну, а на іншій – такого готового рішення може не бути, і його доведеться створювати з нуля. Відповідно, окрім вищезазначених пунктів, важливу роль відіграє технічна оснащеність мови, кількість бібліотек та фреймворків, створених для неї, адже від цього безпосередньо залежить, чи займатиме розробка більше часу, чи ні.

Не менш важливе значення має також цільове призначення мови. Це важливий фактор при її виборі, адже якщо використовувати технологію не за її прямим призначенням або поза межами її компетенції, то рішення, які будуть створені, не працюватимуть так ефективно, стабільно й надійно, як могли б, незалежно від якості реалізації коду. Наприклад, якщо мова програмування задумувалася як така, що працює з даними, але має певний набір функціональності для створення простих веб-інтерфейсів для візуалізації результатів обробки даних, – це не означає, що гарною ідеєю буде створювати на ній складний веб-додаток із великою кількістю серверної логіки, використанням сторонніх сервісів тощо. Незважаючи на те, що таке рішення може працювати – воно не зможе забезпечити належної надійності, розширюваності та елегантності коду, яку надало б рішення, створене мовою, цільовим призначенням якої є розробка веб-

додатків. Окрім того, неправильний вибір мови програмування створює зайву нішованість для продукту, оскільки стає дедалі складніше знайти спеціаліста, який матиме достатній рівень компетентності в конкретній галузі та знання тієї мови, яка використовується в проєкті.

Існує ще один фактор, який варто враховувати при виборі мови – це її заточеність під певну парадигму програмування. Парадигма програмування – це набір методів та принципів, у рамках яких варто розробляти програмне забезпечення. Існує багато парадигм програмування, наприклад: процедурне – де всі інструкції об’єднані у процедури, кожна з яких є окремою від інших і може викликатися незалежно або разом; ООП – об’єктно-орієнтоване програмування, яке є одним із найбільш розповсюджених у сфері веб- та десктоп-розробки й базується на поняттях наслідування, поліморфізму, абстракції, а також диктує умову, за якої кожен елемент системи є об'єктом, що може бути батьківським або наслідуваним. При цьому існують підпарадигми, які уточнюють загальні правила. Реактивне програмування заточене на роботу, базуючись на відносних величинах, а не конкретних значеннях, та оперує не значеннями змінних як такими, а потоками подій або векторами. Потрібно також орієнтуватися на те, щоб парадигма підтримувалася мовою програмування, яка буде використовуватися. Як уже було зазначено, найпопулярнішою парадигмою програмування в домені веб є ООП. Мова повинна підходити для роботи у рамках певної парадигми. Існують мови програмування, які заточені лише під одну парадигму. Наприклад, C# орієнтований на ООП, а Elm, у свою чергу, використовується виключно для реактивного програмування. Як і з попередніми аспектами – одна мова може бути заточена під певну парадигму, але також мати можливість використовувати іншу, не надаючи при цьому повного обсягу можливостей, як функціональна мова. Існують також мультипарадигменні мови програмування, наприклад, Python, де можуть застосовуватися ООП, реактивне та процедурне програмування.

Оскільки будується веб-додаток, який повинен бути відкритим до розширення, мати готові інструменти для роботи з мережевими запитами,

обробкою авторизації, рішеннями для роботи з СУБД, а також бути відносно простим, лаконічним і підтримувати відповідну парадигму програмування, – вибір мови стає ключовим.

До розроблюваної системи висуваються такі вимоги: вона повинна бути створена на основі парадигми ООП, оскільки ця парадигма є найпоширенішою і добре підходить для реалізації складної логіки серверної частини веб-застосунку. Також потрібен інтерфейс для роботи з мережевими запитами, оскільки додаток передбачає взаємодію з іншими частинами системи за допомогою мережевого протоколу верхнього рівня HTTP. Крім того, необхідна можливість працювати з базами даних та СУБД, причому – з різними, залежно від потреб.

Проаналізувавши всі вимоги, було обрано мову програмування високого рівня Python, яка задовольняє всі потреби цього продукту. Python – це мова програмування високого рівня, яка дозволяє реалізовувати правила парадигми ООП, має велику кількість бібліотек, число й призначення яких постійно зростає. Крім того, спільнота Python активно розвивається, і ця мова має велику кількість висококваліфікованих фахівців. Python має готові рішення для роботи з мережевим трафіком і веб-додатками, що й відповідає цільовому призначенню. Наприклад, бібліотека requests дозволяє виконувати HTTP-запити, обробляти їх, вирішувати конфлікти, обробляти помилки мережевих протоколів. Якщо необхідно використовувати ресурси більш ефективно – існують рішення, які дозволяють частково реалізовувати патерни реактивного програмування для роботи з мережевим трафіком, наприклад: aiohttp, httpx, asynchttp, кожна з яких має свої переваги.

Для створення серверних додатків та API Python також має багато різних рішень – від базових, як aiohttp, де майже все потрібно реалізовувати вручну (засоби авторизації, обробка помилок, конфліктів тощо), до великих фреймворків, які вже мають вбудовані реалізації цих задач, наприклад: Flask, Django, FastAPI, Quart.

Якщо говорити про роботу з БД та СУБД, у Python також є готові рішення для цього – починаючи від базових, як-от psycopg2, де робота відбувається

буквально за рахунок надсилання запитів мовою SQL, і закінчуючи повноцінними ORM та кверібілдерами, як-от SQLAlchemy або Tortoise.

Отже, у нашому випадку Python цілком відповідає всім потребам для того, щоб стати основною мовою серверної логіки застосунку. Хоча він і має певні особливості – зокрема те, що ця мова програмування не має строгої типізації і контроль типів змінних переходить від мови програмування безпосередньо до програміста, який повинен контролювати тип значень змінних і відповідно працювати з кожною з них.

Для вирішення цього було створено механізм, який має назву *type* hinting – він дозволяє вказати передбачений тип даних змінної, однак це не обмежує змінну в прийнятті іншого типу даних, а просто підсвічує передбачений.

Через те, що Python є інтерпретованою, а не компільованою мовою, тобто код високого рівня перетворюється на код нижчого рівня прямо під час виконання програми, помилки типізації часто стають проблемними. Їх потрібно активно відстежувати, адже такі помилки буде виявлено вже під час виконання, а не на етапі компіляції програми.

РОЗДІЛ 4
ВПРОВАДЖЕННЯ СИСТЕМИ

Для розгортання програмного забезпечення, ефективної та стабільної його роботи йому потрібно забезпечити певне середовище. В контексті розгортання системи потрібно розглядати систему як апаратно-програмний комплекс. Таким чином вимоги будуть відноситись і до потужності машини, на якій розгортається система, так і до системи, яка буде встановлена на цій машині. Перелік програмних вимог:

- Через особливість того, як була побудована системи вона є майже агностична до того, в рамках якої операційної системи вона розгортається. Зумовлено це тим, що системи цілком будувалась на технології контейнеризації.
- Контейнеризація – підхід до розробки та розгортання систем, який заключається у тому, що програма розгортається всередині контейнера. Контейнер собою по суті являє віртуальну машину, яка створюється на хост машині (рис. 4.1). Такий підхід дозволяє не залежати від ОС(Windows, MacOS, Linux). Процес побудови такої системи виглядає так:

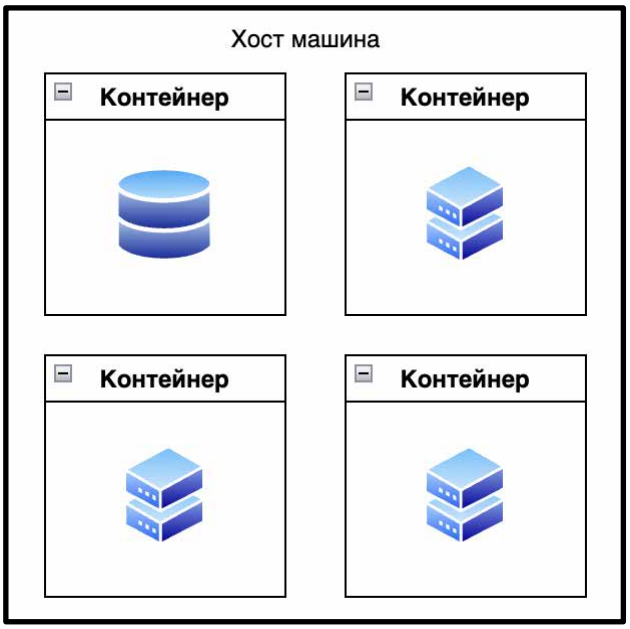


Рис. 4.1. Схематичне зображення розгортання програми

На схемі схематично зображено те, як виглядають контейнери в рамках основної системи. У випадку використання контейнерів можна виділити ряд переваг у порівнянні із розгортанням на локальній машині безпосередньо:

1. Інкапсульованість
    a. Система контейнеризації передбачає відокремлення робочих областей між програмою та локальним середовищем. Для програми створюється окрема підсистема, в рамках якої програма функціонує. Завдяки чому не треба адаптувати програму під кожну систему окремо, що спрощує дистрибуцію; відсутність конфлікту змінних середовища, через те, що системні змінні середовища та контейнерні не пересікаються; можливість налаштування кожного контейнера індивідуально, що надає велику гнучкість
2. Горизонтальна масштабованість
    a. Завдяки тому, що контейнер інкапсульований від зовнішнього середовища та один від одного за звичайних налаштувань – неможлива ситуація, в якій при бажанні запустити декілька екземплярів програми вони будуть пересікатись в рамках процесорних дій, оперативної пам’яті (мутація комірок пам’яті), виникнення “Стан Гонки”, завдяки чому можна запускати одну і ту саму програму в різних контейнерах, завдяки чому можуть бути отримані певні переваги в продуктивності роботи
    b. Окрім можливості створювати багато екземплярів класу присутня можливість налаштувати контейнери так, щоб вони взаємодіяли між собою
3. Безпека
    a. У випадку інфікування операційної системи контейнера, критичного збою в роботі програми, або будь якому іншому збою в системі

контейнера це не потягне за собою наслідків для хост машини, а також решта екземплярів класів

Для зручнішого адміністрування та оркестрації контейнеризованих сервісів у проекті використано поділ на фронтенд та бекенд частини, кожна з яких працює у відповідній системі контейнерів. Схема побудована з урахуванням розділення відповідальності та комунікацій між компонентами. На рис. 4.2 представлено діаграму розгортання системи, яка відображає взаємодію між компонентами програмного забезпечення, контейнерами та протоколами обміну даними.

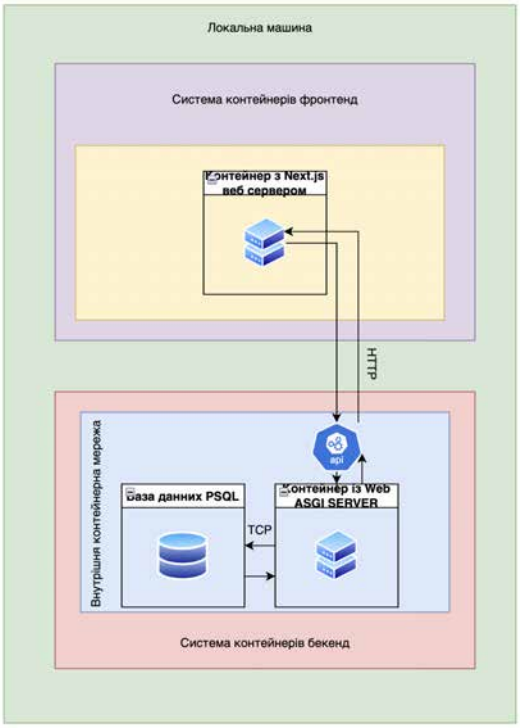


Рис. 4.2. Діаграма розгортання програмного забезпечення онлайн-сервісу

Завдяки такому підходу отримуємо ефективну систему, яка гарно підходить для розповсюдження, розширення, модифікації, через що вона буде стабільною та прогнозованою.

РОЗДІЛ 5
ТЕСТУВАННЯ СИСТЕМИ

Тестування системи здійснювалося на рівні взаємодії кінцевого користувача з вебінтерфейсом, у реальному середовищі за участі тестових даних.

Основною метою є перевірка відповідності функціоналу розробленого ПЗ заявленим вимогам, стабільності його роботи, коректності переходів між станами інтерфейсу, зручності користування та адаптивності. Особлива увага приділялася сценаріям, які проходять через усі основні компоненти системи: авторизацію, каталог, оформлення замовлення, керування профілем і панель адміністратора.

5.1 Адаптивність інтерфейсу користувача

Взаємодіяти з системою користувач може за допомогою веб переглядача із будь якого пристрою. Перевагою веб додатків є їхня нативна кросплатформеність. Через свою природу, веб додатки працюють всередині переглядачів які заздалегідь адаптовані для різного роду контенту. Для ще кращого сприйня інтерфейсу користувачами, які користуються мобільними платформами або мають нестандартні розширення екрану графічний інтерфейс було адаптовано до мобільних пристроїв (рис. 5.1):

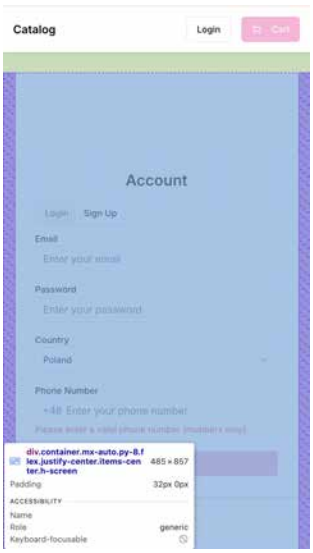


Рис. 5.1. Адаптивна сторінка реєстрації для мобільного пристрою

Граничні показники адаптації по співвідношенню сторін 16:9 є 485x860 пікселів, завдяки чому інтерфейс буде читаємим та зручним навіть на доволі застарілих девайсах. Також, завдяки технології, яка використовувалась для створення фронтенда, а саме фреймворк для CSS Tailwind, який має здатність примітизуватись залежно від версій переглядачів – старі версії веб браузерів не будуть перешкоджати відображенню контенту, бо технологія сама буде спрощувати елементи верстки, якщо це буде потрібно.

5.2 Мовна локалізація та відображення каталогу

На рисунках 5.2 і 5.3 показано, що вебзастосунок підтримує дві мови інтерфейсу — українську та англійську. При перемиканні мови автоматично оновлюється текстовий контент, при зміні валюти автоматично змінюються числові значення відповідно до поточного курсу. Локалізація реалізована з урахуванням міжнародних стандартів форматування валют та одиниць виміру.

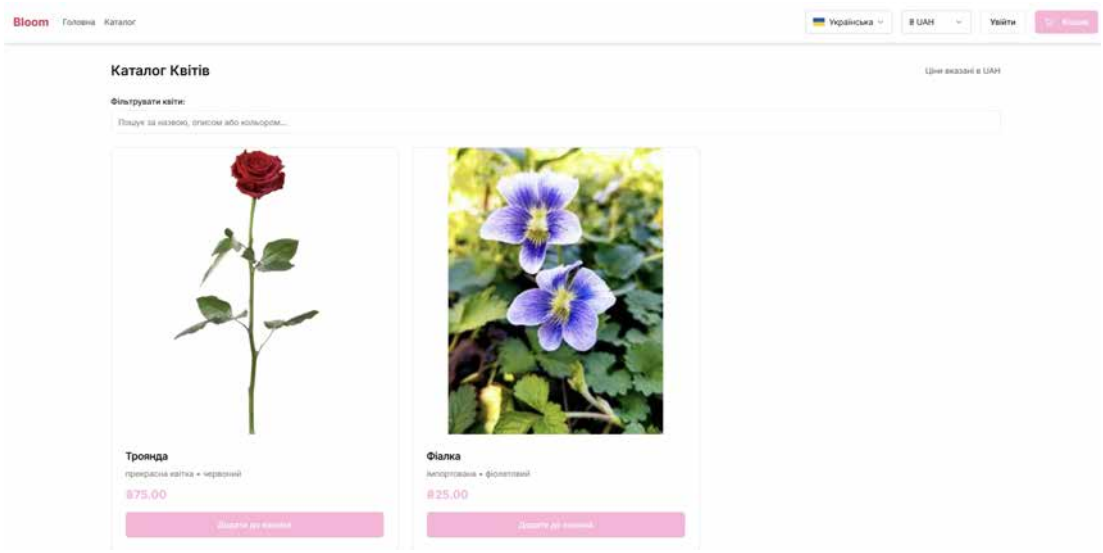


Рис. 5.2. Каталог українською мовою

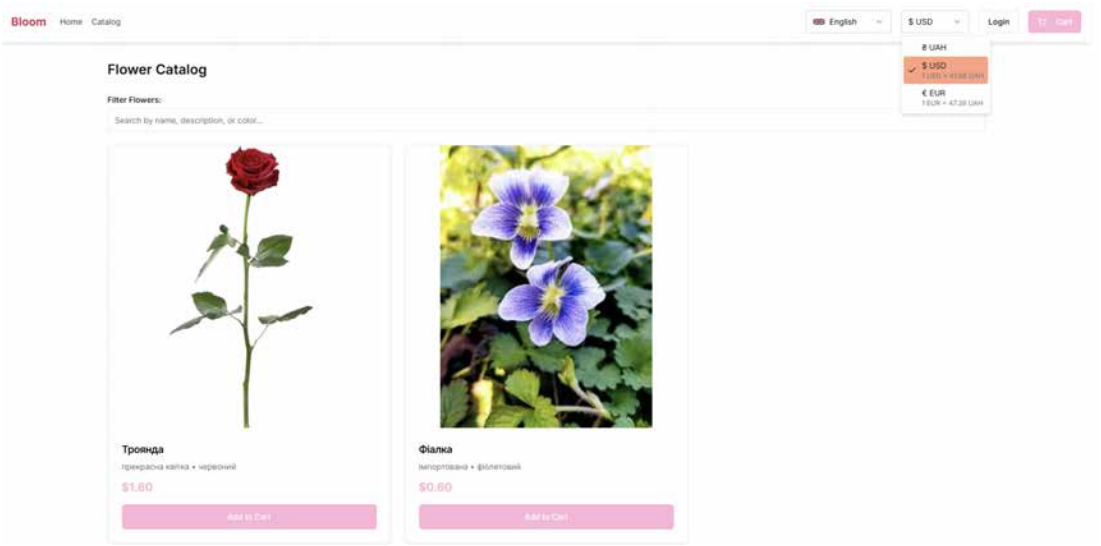


Рис. 5.3. Каталог англійською мовою з цінами в іноземній валюті

5.3 Перевірка доступу до оформлення без авторизації

У випадку, якщо неавторизований користувач намагається перейти до етапу оформлення замовлення, система блокує подальші дії і відображає відповідне повідомлення (рис. 5.4). Це демонструє реалізацію захисту критичних дій за допомогою перевірки стану автентифікації.

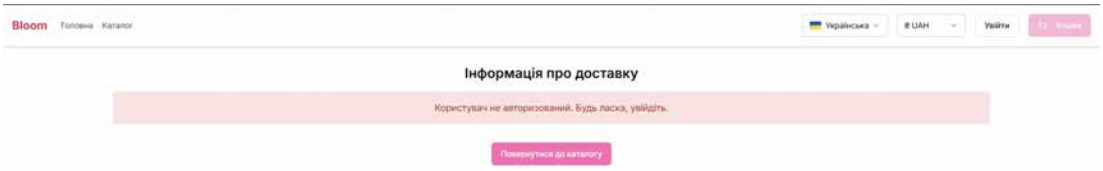


Рис. 5.4. Сторінка з повідомленням про необхідність авторизації

5.4 Авторизація та реєстрація користувачів

Вебзастосунок реалізує два окремі процеси автентифікації:

- Авторизація – перевірка облікових даних із формою для введення email та пароля (рис. 5.5).
- Реєстрація – створення нового облікового запису з перевіркою валідності введених даних (рис. 5.6).

На обох формах передбачено повідомлення про помилки, підказки до введення полів та автоматичну валідацію.

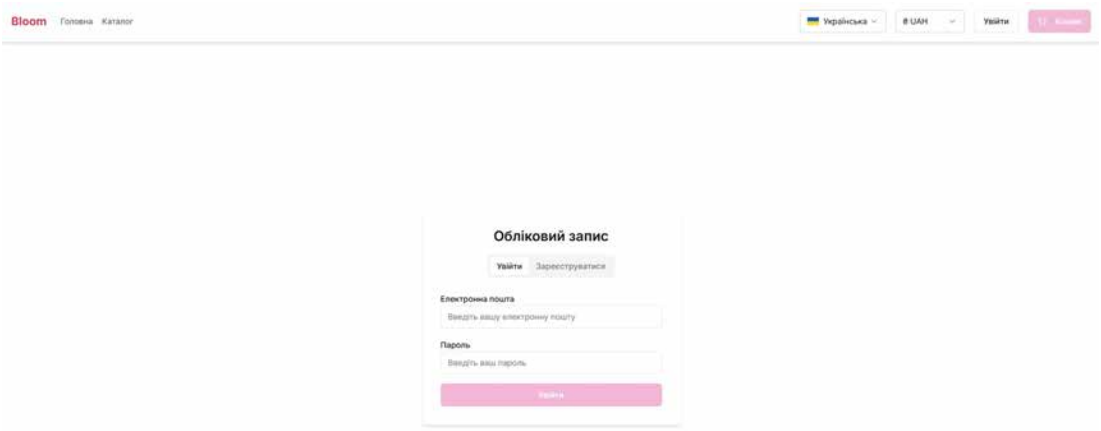


Рис. 5.5. Сторінка авторизації

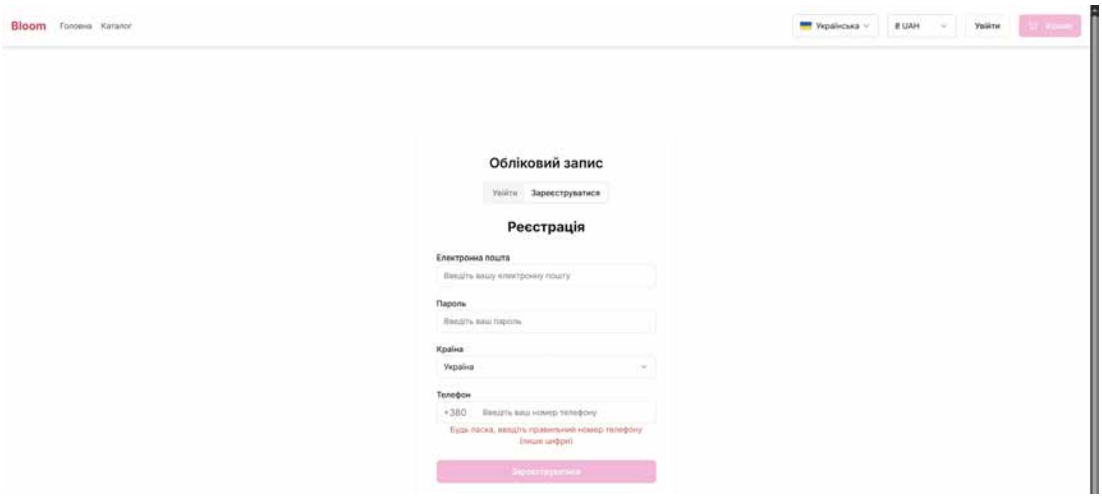


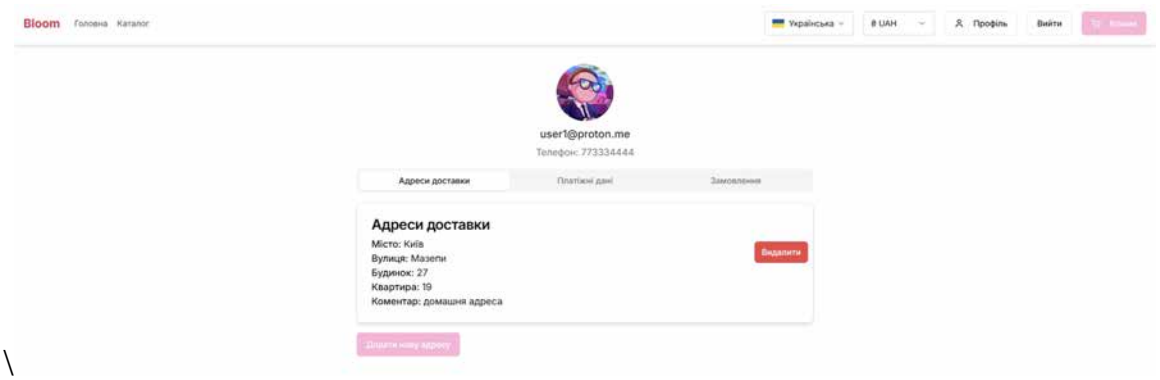
Рис. 5.6. Сторінка реєстрації

5.5 Профіль користуваача

Після входу в акаунт користувач може керувати даними свого профілю:

- Вкладка “Адреса доставки” (рис. 5.7) дозволяє додати, редагувати або видалити адресу.
- Вкладка “Платіжні дані” (рис. 5.8) містить поля для додавання або оновлення картки, які підтримують маску введення.

- Вкладка “Замовлення” (рис. 5.9) демонструє повну історію оформлень із зазначенням статусу кожного.



\

Рис. 5.7. Профіль користувача – вкладка “Адреса доставки”

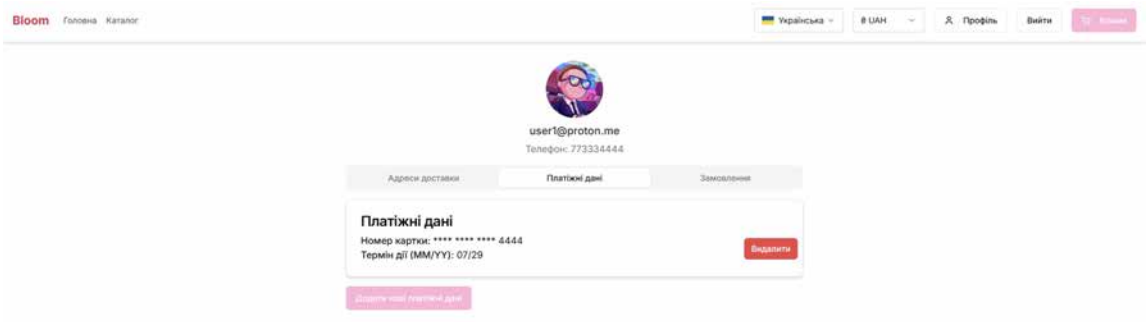


Рис. 5.8. Профіль користувача – вкладка “Платіжні дані”

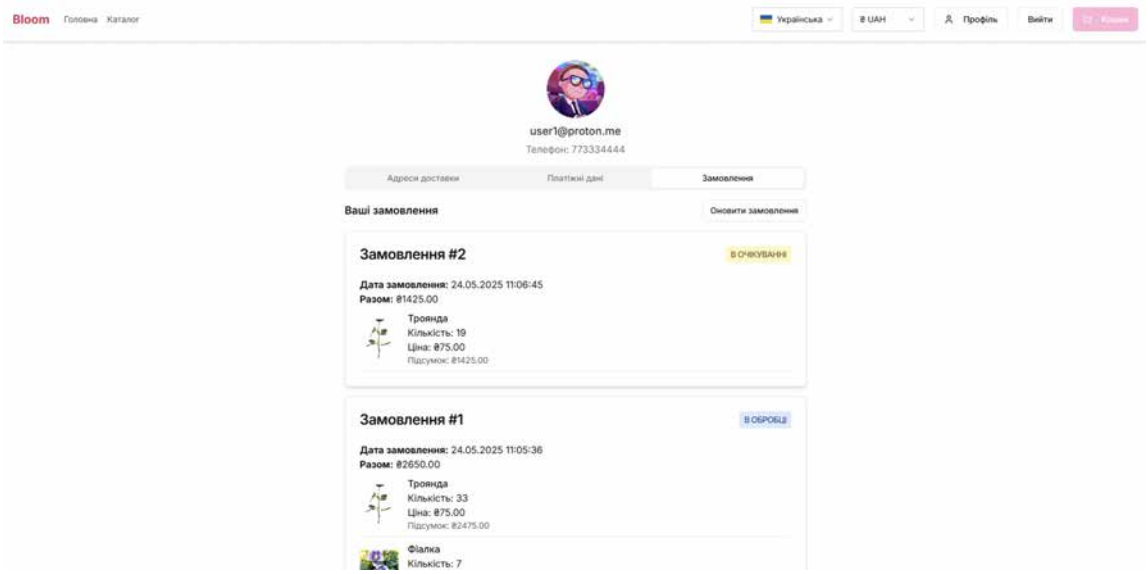


Рис. 5.9. Профіль користувача – вкладка “Замовлення”

5.6 Оформлення замовлення

Процес оформлення включає послідовність дій із перевіркою на кожному етапі:

1. Перегляд вмісту кошика (рис. 5.10): показуються всі додані товари, їх кількість і загальна сума.
2. Заповнення форми доставки (рис. 5.11): користувач вказує ім’я, контактні дані та адресу.
3. Вибір геолокації на карті (рис. 5.12): інтегрована карта Google дозволяє знайти точне місцезнаходження.
4. Введення платіжної інформації (рис. 5.13): поля формату CVV, дати дії картки й номера мають автоперевірку.
5. Підтвердження замовлення (рис. 5.14): система повертає повідомлення про успішне завершення дії.

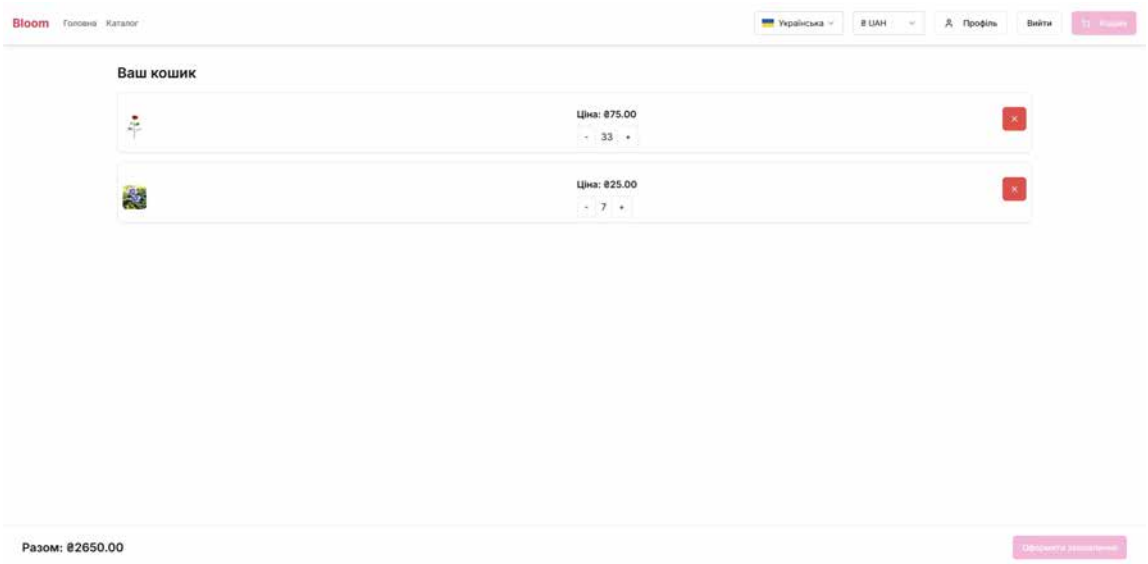


Рис. 5.10. Кошик з доданими товарами

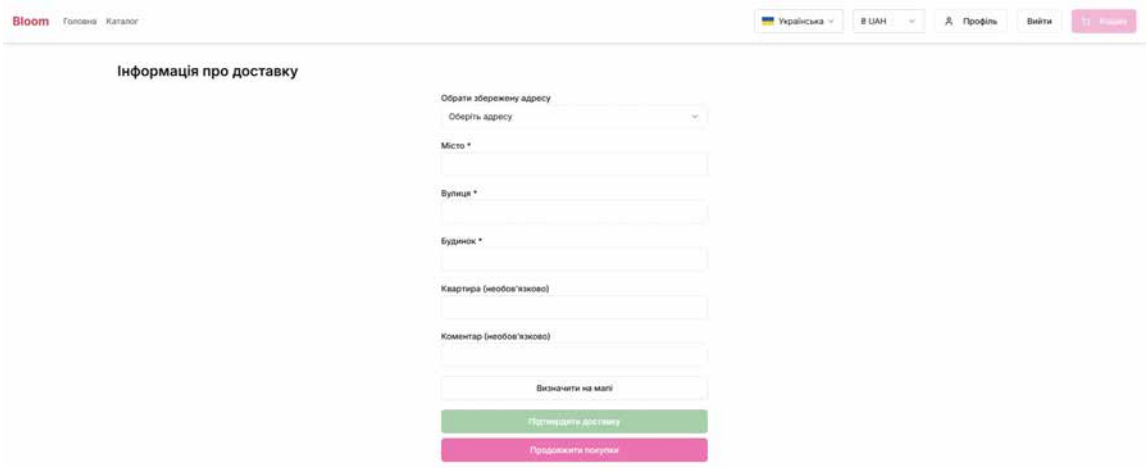


Рис. 5.11. Форма оформлення доставки

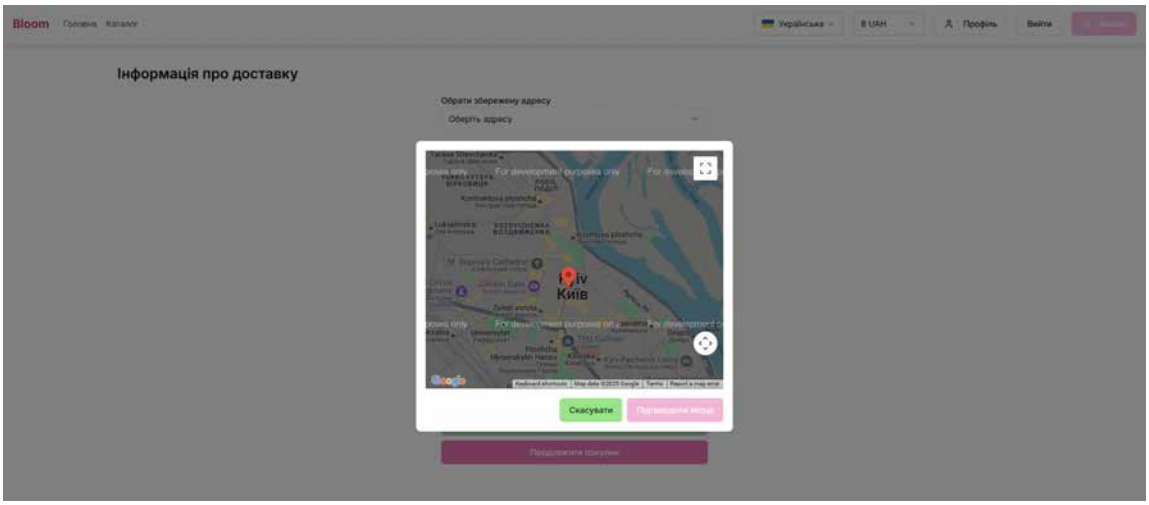


Рис. 5.12. Інтерактивна мапа з визначенням поточного місцезнаходження

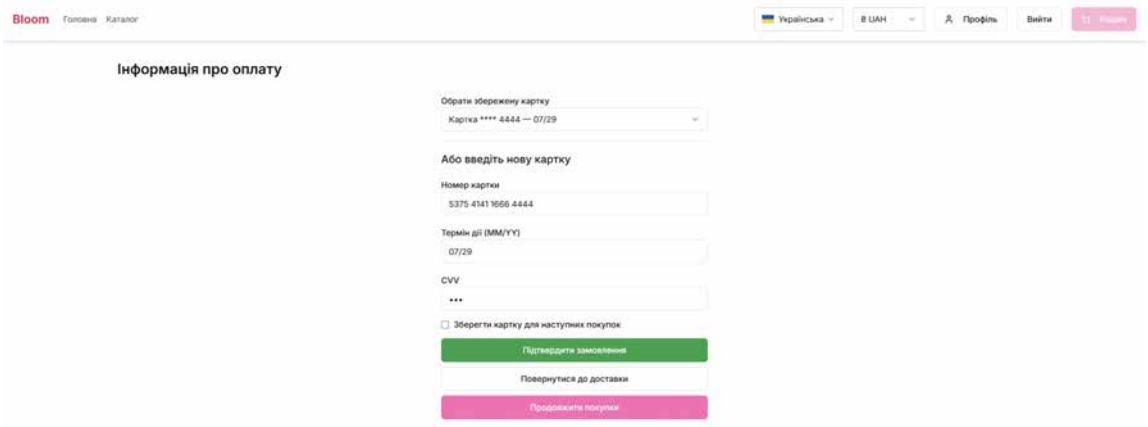


Рис. 5.13. Введення платіжної інформації

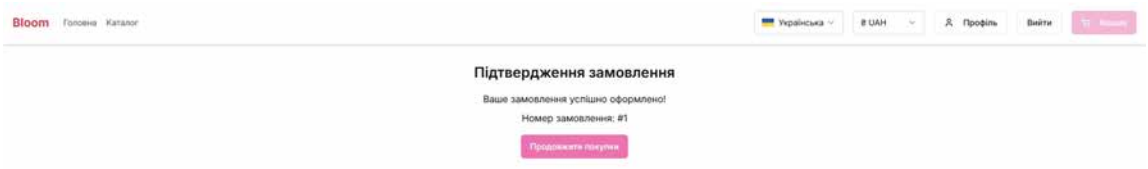


Рис. 5.14. Повідомлення про успішне оформлення замовлення

5.7 Панель адміністратора

Окремий функціонал передбачений для адміністратора, який має доступ до керування товарами та замовленнями:

- **Інвентар** (рис. 5.15): таблиця з усіма позиціями товарів, можливістю редагування, видалення та додавання.
- **Список усіх замовлень** (рис. 5.16): кожне замовлення відображається з деталями та кнопками зміни статусу.
- **Редагування статусу замовлення** (рис. 5.17): доступні значення “В очікуванні”, “В обробці”, “Виконано”, “Скасовано”.

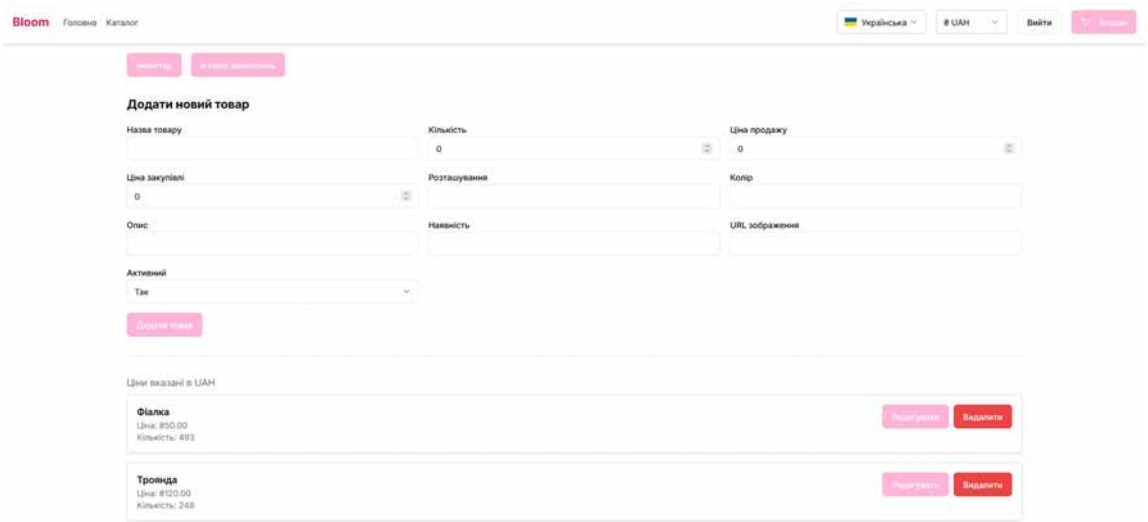


Рис. 5.15. Адмін-панель – вкладка “Інвентар”

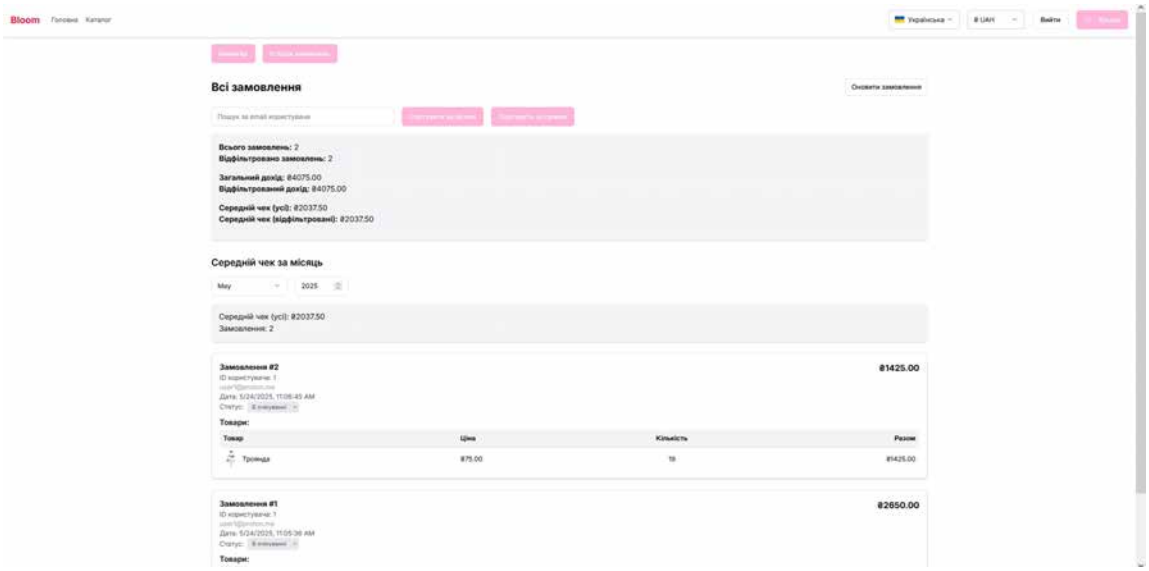


Рис. 5.16. Адмін-панель – вкладка “Всі замовлення”

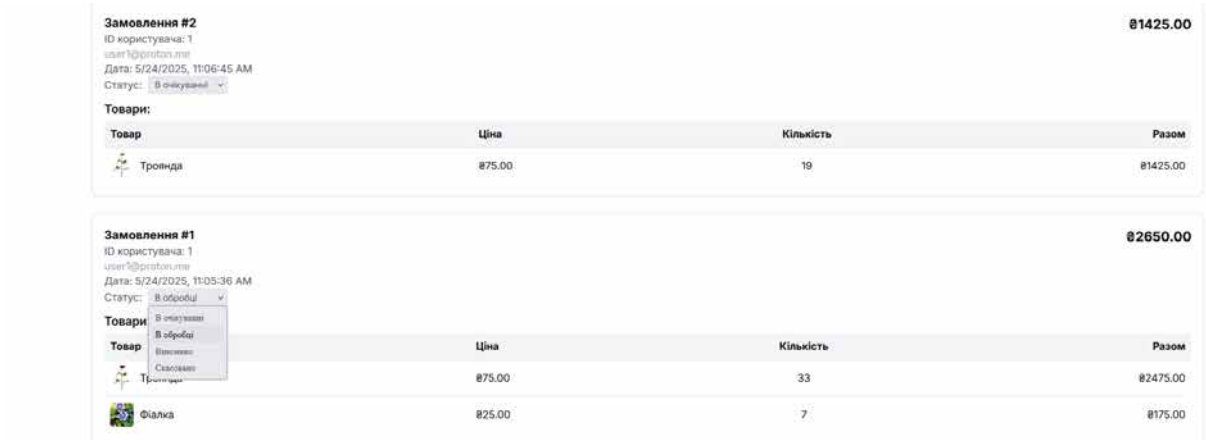


Рис. 5.17. Адмін-панель – зміна статусу замовлення

ВИСНОВКИ

У ході написання бакалаврської кваліфікаційної роботи було досягнуто поставленої мети – розробити інформаційну систему, яка дозволяє автоматизувати процес продажу флористичних товарів з можливістю адміністрування асортименту та оформлення замовлень через веб-інтерфейс. Результати дослідження охоплюють теоретичний, аналітичний, інженерний і прикладний рівні опрацювання завдань.

Виконано наступні дослідницькі завдання:

1. **Досліджено предметну галузь** онлайн-продажу квітів і виявлено ключові потреби бізнесу та користувачів. Проаналізовано схожі рішення та сформульовано функціональні та нефункціональні вимоги до програмного продукту, які забезпечують конкурентоспроможність і зручність системи.
2. **Проаналізовано сучасні інженерні підходи**, що застосовуються при побудові архітектури вебдодатків, зокрема модель REST-клієнт/сервер, контейнеризацію, застосування фреймворків для фронтенду (Tailwind CSS) та бекенду. Результатом аналізу стало формування архітектурного шаблону системи.
3. **Побудовано інформаційну модель системи**. Створено низку UML-діаграм, які охоплюють прецеденти, діяльність і послідовність дій. Ці діаграми дозволили структуровано відобразити ключові сценарії роботи користувачів і процеси взаємодії в системі.
4. **Розроблено ER-модель бази даних**, яка реалізує логічну структуру даних і забезпечує ефективне зберігання, доступ і обробку інформації. База даних спроєктована з урахуванням нормалізації, що дозволяє уникати надмірності даних.
5. **Реалізовано прототип програмного забезпечення**, який містить розроблений користувацький інтерфейс (функціонал реєстрації,

перегляду товарів, оформлення замовлення, оплати тощо) і адміністративну панель для управління товарами. Забезпечено адаптивність дизайну, що дозволяє працювати з системою на різних пристроях.

6. **Здійснено впровадження системи** у віртуалізоване середовище за допомогою технологій контейнеризації, що забезпечує платформонезалежність, гнучкість масштабування та спрощену експлуатацію.
7. **Проведено тестування прототипу**, результати якого свідчать про відповідність функціональних можливостей заявленим вимогам, а також про стабільність та працездатність розробленої системи.

Таким чином, у межах виконаного дослідження розроблено цілісне, гнучке та масштабоване прикладне програмне забезпечення для онлайн-сервісу продажу квітів. Робота продемонструвала важливість системного підходу, глибокого аналізу предметної галузі та грамотного вибору технологій для досягнення високих показників якості ПЗ.

Результати роботи можуть бути використані для подальшої розробки комерційного веб-додатку, а також як основа для створення типових архітектур рішень у галузі електронної комерції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бруяка О. В. *Проєктування інформаційних систем: навчальний посібник*. – Київ: КНЕУ, 2020. – 212 с.
2. Бялік О. І. *Основи баз даних: навчальний посібник*. – Харків: Факт, 2019. – 196 с.
3. Козлов В. В. *UML: Мова моделювання об'єктно-орієнтованих систем*. – Київ: Діалектика, 2021. – 328 с.
4. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт / НУБіП України. – Київ, 2023.
5. Офіційний сайт PostgreSQL. – Режим доступу: postgresql.org
6. Офіційна документація Docker. – Режим доступу: docs.docker.com
7. Офіційна документація Python. – Режим доступу: docs.python.org
8. *API design and management* – TechTarget. – Режим доступу: techtarget.com/searchapparchitecture/resources/API-design-and-management
9. Sommerville, I. *Software Engineering*, 10th Edition. – Pearson, 2015. – 792 p.
10. Yourdon, E. *Modern Structured Analysis*. – Prentice Hall, 1989. – 614 p.
11. Pavlenko, I. *Хмарні обчислення в сучасних IT-системах*. – Вісник НТУУ "КПІ", 2023, №1. – С. 55–62.
12. Білан Н. С. *Сучасні підходи до розробки веб-застосунків*. – Вісник КНУ, Серія: прикладна інформатика, 2022, №4. – С. 88–94.
13. Schmidt, L. *PostgreSQL 15 Cookbook*. – Apress, 2023. – 326 p.
14. Erdoğan, N. FastAPI 101: Develop APIs With Python and FastAPI Framework. – 2022. – Режим доступу: logischermix.com/engineering/fastapi101
15. Сторінка документації FastAPI. – Режим доступу: fastapi.tiangolo.com
16. Офіційна документація Tailwind CSS. – Режим доступу: tailwindcss.com/docs

17. Кравченко О. В. *Застосування мікросервісної архітектури у веб-розробці*. – Інформаційні технології і засоби навчання, 2022. – №6(96). – С. 25–32.
18. Singh, K. *Mastering Full Stack Development with FastAPI and React*. – O'Reilly Media, 2024. – 384 p.
19. Morgan, D. *Node.js Web Development*, 6th Edition. – Packt Publishing, 2023. – 486 p.

ДОДАТКИ

Додаток А – Фрагменти коду frontend-частини

А1. Компонент каталогу товарів для клієнтів

```
"use client";

import { useState, useEffect } from 'react';
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card";
import { Input } from "@/components/ui/input";
import { Label } from "@/components/ui/label";
import { Button } from "@/components/ui/button";
import { useCart } from '@/hooks/use-cart';
import { useToast } from '@/hooks/use-toast';

const Catalog = () => {
  const [filter, setFilter] = useState("");
  const [flowersData, setFlowersData] = useState<any[]>([]); // State to hold
flower data
  const { addItem } = useCart();
  const { toast } = useToast();

  useEffect(() => {
    // Fetch data from the backend
    const fetchFlowers = async () => {
      try {
        const response = await
fetch("http://localhost:8000/user/flower/products/get");
        const data = await response.json();

        if (data && data.data) {
          // Filter active flowers
          const activeFlowers = data.data.filter((flower: any) => flower.is_active
=== "true");
          setFlowersData(activeFlowers);
        }
      } catch (error) {
        console.error("Error fetching flower data:", error);
      }
    };

    fetchFlowers();
  }, []);

  const filteredFlowers = flowersData.filter(flower =>
    flower.product_name.toLowerCase().includes(filter.toLowerCase()) ||
    flower.description.toLowerCase().includes(filter.toLowerCase()) ||
```

```
    flower.color.toLowerCase().includes(filter.toLowerCase())
  );

  const handleAddToCart = (flower: any) => {
    const itemForCart = {
      id: flower.id,
      name: flower.product_name,
      image_url: flower.image_url,
      price: Number(flower.purchase_price) || 0,
    };

    addItem(itemForCart);

    toast({
      title: "Item added to cart",
      description: `${flower.product_name} has been added to your cart.`,
    });
  };

  return (
    <div className="container mx-auto py-8">
      <h2 className="text-2xl font-semibold mb-4">Flower Catalog</h2>

      <div className="mb-4">
        <Label htmlFor="filter">Filter Flowers:</Label>
        <Input
          type="text"
          id="filter"
          placeholder="Search by name, description, or color..."
          value={filter}
          onChange={(e) => setFilter(e.target.value)}
        />
      </div>

      <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-4">
        {filteredFlowers.map(flower => (
          <Card key={flower.id}>
            <CardHeader>
              <CardTitle>{flower.product_name}</CardTitle>
              <CardDescription>{flower.description} -
{flower.color}</CardDescription>
            </CardHeader>
            <CardContent className="flex flex-col">
              <img src={flower.image_url} alt={flower.product_name} className="w-
full rounded-md mb-2" />
              <p className="font-semibold">Price: ${flower.buy_price}</p>
              <Button onClick={() => handleAddToCart(flower)}>Add to Cart</Button>
            </CardContent>
          </Card>
        ))}
```

```
      </div>
    </div>
  );
};

export default Catalog;
```

А2. Фрагменти коду сторінки для введення адреси доставки

```
// ...

const MapComponent = dynamic(() => import('@/components/MapComponent'), { ssr:
false });

// ...

export default function DeliveryPage() {
  const router = useRouter();
  const [city, setCity] = useState("");
  const [street, setStreet] = useState("");
  const [houseNumber, setHouseNumber] = useState("");
  const [mapOpen, setMapOpen] = useState(false);
  const [isFormValid, setIsFormValid] = useState(false);
  const userId = typeof window !== "undefined" ? localStorage.getItem("userId") :
null;

  useEffect(() => {
    setIsFormValid(city && street && houseNumber);
  }, [city, street, houseNumber]);

  // ...

  const handleLocationSelected = async ({ lat, lng }: { lat: number; lng: number })
=> {
    try {
      const res = await
fetch(`https://nominatim.openstreetmap.org/reverse?format=jsonv2&lat=${lat}&lon=${l
ng}`);
      const data = await res.json();
      if (data.address) {
        setCity(data.address.city || data.address.town || "");
        setStreet(data.address.road || "");
        setHouseNumber(data.address.house_number || "");
      }
    } catch {
      console.error("Geolocation error");
    }
    setMapOpen(false);
  };
```

```
  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    if (!isFormValid || !userId) return;

    const payload = { user_id: parseInt(userId), city, street, house_number:
houseNumber };

    try {
      const res = await fetch("http://127.0.0.1:8000/user/shipment/shipment-
addresses", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(payload),
      });

      if (!res.ok) throw new Error();
      const created = await res.json();
      localStorage.setItem("selectedAddressId", created.id.toString());
      router.push("/payment");
    } catch {
      console.error("Save address error");
    }
  };

  // ...

  return (
    <form onSubmit={handleSubmit}>
      <Label>City *</Label>
      <Input value={city} onChange={(e) => setCity(e.target.value)} required />
      <Label>Street *</Label>
      <Input value={street} onChange={(e) => setStreet(e.target.value)} required />
      <Label>House Number *</Label>
      <Input value={houseNumber} onChange={(e) => setHouseNumber(e.target.value)}
required />

      <Button type="button" onClick={() => setMapOpen(true)}>Detect on Map</Button>
      <Button type="submit" disabled={!isFormValid}>Confirm Delivery</Button>

      {mapOpen && (
        <MapComponent
          onLocationSelected={handleLocationSelected}
          onClose={() => setMapOpen(false)}
        />
      )}
    </form>
  );
}
```

Додаток В – Фрагменти коду backend-частини

В1. Код моделі товару Flower

```
import uuid
from sqlalchemy import Column, String, DateTime, func, Integer, Float
from sqlalchemy.dialects.postgresql import UUID

from app.db.pg_base import Base


class Flower(Base):
    __tablename__ = 'flower'

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    product_name = Column(String(50), nullable=False)
    image_url = Column(String(255), nullable=False)
    color = Column(String(50), nullable=False)
    buy_price = Column(Float, nullable=False)
    location = Column(String(50), nullable=False)
    quantity = Column(Integer, nullable=False)
    stock = Column(String(50), nullable=False)
    is_active = Column(String(50), nullable=False, default='true')
    purchase_price = Column(Float, nullable=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())
    description = Column(String(255), nullable=True)

    def __repr__(self):
        return f"""<Flower(id={self.id}, product_name={self.product_name},
        image_url={self.image_url}, color={self.color},
        description={self.description},
        buy_price={self.buy_price}, location={self.location},
        quantity={self.quantity},
        is_active={self.is_active}, purchase_price={self.purchase_price})>"""
```

В2. Контролер для обробки замовлень користувачів

```
from app.dto.orderDto import OrderDto
from app.service.orderService import OrderService
from fastapi import APIRouter


router = APIRouter(tags=["order"])

@router.post("/create")
async def create_order(order: OrderDto):
    order_service = OrderService()
```

```
    order_data = order.model_dump()
    created_order = order_service.create_order(order_data)
    return {"id": created_order.id}

@router.get("/getall/{user_id}")
async def get_user_orders(user_id: int):
    order_service = OrderService()
    user_orders = order_service.get_user_orders(user_id)
    return {"orders": user_orders}
```

В3. Сервіс створення замовлень

```
from app.db.models.orderModel import Order, OrderItem
from app.db.pg_session import PgSession
from app.db.repository.order.orderRepository import OrderRepository
from app.service.orderItemService import OrderItemService
from app.service.flowerService import FlowerService
import logging

class OrderService:
    def __init__(self):
        self.__pg_session = PgSession()
        self.__session = self.__pg_session.get_session()
        self.order_repository = OrderRepository(self.__session)
        self.order_item_service = OrderItemService()
        self.flower_service = FlowerService()

    def create_order(self, order_data):
        """Create a new order with items"""
        user_id = order_data.get("user_id")
        items = order_data.get("items")
        status = order_data.get("status", "completed")
        delivery_address = order_data.get("delivery_address")
        payment_method = order_data.get("payment_method")

        if not items:
            raise ValueError("Invalid order details: no items provided")

        # Calculate total price from items
        total_price = sum(item.get('price', 0) * item.get('quantity', 0) for item
in items)

        # Create order with calculated total price
        order = Order(
            user_id=user_id,
            total_price=total_price,
            status=status,
```

```
            delivery_address=delivery_address,
            payment_method=payment_method
        )

        # Create the order first to get an ID
        created_order = self.order_repository.create_order(order)

        # Now create the order items with the order ID
        for item in items:
            item_data = {
                'good_id': item.get('good_id'),
                'name': item.get('name'),
                'image_url': item.get('image_url'),
                'price': item.get('price'),
                'quantity': item.get('quantity'),
                'order_id': created_order.id
            }

            # Create the order item
            self.order_item_service.create_order_item(item_data)

            # Update flower inventory
            self.flower_service.change_flower_quantity(
                item.get('good_id'),
                item.get('quantity')
            )

        return created_order

    def get_user_orders(self, user_id):
        """Get all orders for a specific user"""
        if not user_id:
            raise ValueError("User ID is required")

        orders = self.order_repository.get_orders_by_user_id(user_id)

        if not orders:
            return []

        user_orders = []

        for order in orders:
            order_items =
self.order_item_service.get_order_items_by_order_id(order.id)

            # Calculate total from items to ensure accuracy
            calculated_total = sum(item.price * item.quantity for item in
order_items)

            order_data = {
```

```
                "order_id": order.id,
                "user_id": order.user_id,
                "user_email": order.user.email if order.user else None,
                "total_price": calculated_total,
                "status": order.status,
                "created_at": order.created_at.strftime("%Y-%m-%d %H:%M:%S"),
                "items": [
                    {
                        "flower_id": item.good_id,
                        "quantity": item.quantity,
                        "price": item.price,
                        "image_url": item.image_url,
                        "name": item.name,
                    }
                    for item in order_items
                ],
            }
            user_orders.append(order_data)

        return user_orders

    def get_all_orders(self):
        """Get all orders in the system for admin view"""
        try:
            orders = self.order_repository.get_all_orders()

            if not orders:
                return []

            all_orders = []

            for order in orders:
                order_items =
self.order_item_service.get_order_items_by_order_id(order.id)

                # Calculate total from items to ensure accuracy
                calculated_total = sum(item.price * item.quantity for item in
order_items)

                order_data = {
                    "order_id": order.id,
                    "user_id": order.user_id,
                    "user_email": order.user.email if order.user else None,
                    "total_price": calculated_total,
                    "status": order.status,
                    "created_at": order.created_at.strftime("%Y-%m-%d %H:%M:%S"),
                    "items": [
                        {
                            "flower_id": item.good_id,
                            "quantity": item.quantity,
```

```
                            "price": item.price,
                            "image_url": item.image_url,
                            "name": item.name,
                        }
                        for item in order_items
                    ],
                }
                all_orders.append(order_data)

            return all_orders
        except Exception as e:
            logging.error(f"Error fetching all orders: {str(e)}")
            raise
```

В4. DTO-модель для створення нового товару

```
from pydantic import BaseModel, Field
from typing import Optional


class FlowerDto(BaseModel):
    name: str = Field(..., title="Name of the flower", max_length=100)
    color: str = Field(..., title="Color of the flower", max_length=50)
    price: float = Field(..., title="Price of the flower", gt=0)
    description: Optional[str] = Field(
        None,
        title="Description of the flower",
        max_length=500)

    class Config:
        schema_extra = {
            "example": {
                "name": "Rose",
                "color": "Red",
                "price": 10.5,
                "description": "A beautiful red rose."
            }
        }


class CreateFlowerRequestDto(BaseModel):
    name: str = Field(..., title="Name of the flower", max_length=100)
    color: str = Field(..., title="Color of the flower", max_length=50)
    price: float = Field(..., title="Price of the flower", gt=0)
    description: Optional[str] = Field(
        None,
        title="Description of the flower",
        max_lengt=500)
    quantity: int = Field(..., title="Quantity of the flower", gt=0)
```

```
    location: str = Field(..., title="Location of the flower", max_length=100)
    stock: str = Field(..., title="Stock status of the flower", max_length=50)
    image_url: str = Field(
        ...,
        title="Image URL of the flower",
        max_length=255)
    is_active: bool = Field(..., title="Is the flower active")
    purchase_price: float = Field(
        ...,
        title="Purchase price of the flower",
        gt=0)

    class Config:
        schema_extra = {
            "example": {
                "name": "Tulip",
                "color": "Yellow",
                "price": 7.5,
                "description": "A bright yellow tulip."
            }
        }
```