Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Системотехніки
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Системне проектування касової системи самообслуговування

(тема)

Виконав: студент 2 курсу, групи СПРм-22-1

Кущ О. В.

(посада, прізвище, ініціали)

Спеціальності: 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми: освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма: Системне проектування
(повна назва освітньої програми)

Керівник проф. Калита Н.І.
(посада, прізвище,ініціали)

Допускається до захисту

Зав. кафедри системотехніки проф. Гребеннік І.В
(підпис) (посада, прізвище, ініціали)

2024 р.

Я, як студент ХНУРЕ, розумію та підтримую політику закладу із академічної доброчесності Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

підпис  Олексій Кущ

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування.

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні.

Попередній захист проведено 10 червня 2024 р.

Керівник кваліфікаційної роботи  проф. каф. СТ Калита Н.І.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Системотехніки

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне проектування

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___»_____20___ р.

# ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові *Кущу Олексію Вікторовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи: Системне проектування касової системи самообслуговування
затверджена наказом по університету від " 01 " квітня 20 24 р. № 259 СТ
2. Термін подання студентом роботи " 10 " червня 20 24 р.
3. Вихідні дані до роботи: *касова система самообслуговування у вигляді клієнт-серверного веб-застосунку.*
4. Перелік питань, що потрібно опрацювати в роботі: *вступ, аналіз предметної області, виявлення вимог до системи та постановка задачі, формування вимог до системи, системне проектування інфраструктури системи, вибір хмарного провайдеру, вибір архітектури веб-застосунку, вибір стеку технологій, функціональне моделювання, реалізація веб-застосунку.*
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій, слайдів (п.5 включається до завдання за рішенням випускової кафедри): *концептуальна діаграма головного бізнес-процесу (IDEF0), декомпозиція 1-го рівня головного бізнес-процесу "Зробити покупку", декомпозиція підпроцесу "Формування покупки", декомпозиція підпроцесу "Облік покупки", діаграма дерева вузлів головного бізнес-процесу, діаграма прецедентів покупця, компонентна діаграма системи (звичайна та хмарна*

*версія), прототипи високої деталізації розробленого користувацького інтерфейсу.*

## КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапів роботи | Терміни виконання етапів | Примітка |
|---|---|---|---|
| 1. | Отримання завдання кваліфікаційної роботи | 01.04.24 | *виконано* |
| 2. | Аналіз завдання, літератури та аналогів з теми | 01.04-18.04 | *виконано* |
| 3. | Вибір засобів для реалізації технічних вимог до програми | 18.04-22.04 | *виконано* |
| 4. | Структурне проектування | 22.04-12.05 | *виконано* |
| 5. | Вибір середовища розробки програми | 12.05-13.05 | *виконано* |
| 6. | Розробка програми | 13.05-27.05 | *виконано* |
| 7. | Оформлення графічної частини та програмної документації | 27.05-02.06 | *виконано* |
| 8. | Оформлення пояснювальної записки | 02.06-10.06 | *виконано* |
| 9. | Представлення кваліфікаційної роботи в ДЕК. | 10.06.24 | *виконано* |

Дата видачі завдання 1 квітня 2024 р.

Студент  Кущ О.В.
(підпис) (посада, прізвище, ініціали)

Керівник роботи  проф. Калита Н.І.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ

Пояснювальна записка до магістерської кваліфікаційної роботи: *106* с., *6* табл., *49* рис., *2* додатки, *45* джерел інформації.

СИСТЕМНЕ ПРОЕКТУВАННЯ, ІНФОРМАЦІЙНА СИСТЕМА, КАСОВА СИСТЕМА САМООБСЛУГОВУВАННЯ, IDEF0, MYSQL, JAVA, SPRING, HIBERNATE, MVC, БАГАТОРІВНЕВА АРХІТЕКТУРА, AWS, МІКРОСЕРВЕРНА АРХІТЕКТУРА

Об'єкт дослідження – касове самообслуговування для середнього та малого бізнесу.

Предмет розробки – інформаційна система касового самообслуговування.

Мета роботи – підвищення ефективності середнього та малого бізнесу шляхом реалізації масштабованої інформаційної системи касового самообслуговування.

Методи розробки – системний підхід та методи структурного системного аналізу, методи моделювання та проектування складних систем, хмарні технології (Amazon Web Services), реляційних баз даних, All Fusion Process Modeler (BPWin) як CASE-засіб функціонального моделювання, методи проектування серверних додатків для Інтернет мереж, а саме, таких як, сервісно-орієнтований архітектурний підхід, трирівнева архітектура та архітектурний патерн MVC.

Створено веб-застосунок на базі тришарової клієнт-серверної архітектури. У розробленій системі впроваджена автоматизація головного бізнес-процесу системи касового самообслуговування.

Сфера застосування – підтримка електронної комерції інформаційної система розрахунково-касового самообслуговування.

# ABSTRACT

Master's Thesis: *106* pages, *6* tables, *49* figures, *2* applications, *45* titles.

SYSTEM DESIGN, INFORMATION SYSTEM, SELF-SERVICE CASH SYSTEM, IDEF0, MYSQL, JAVA, SPRING, HIBERNATE, MVC, MULTILEVEL ARCHITECTURE, AWS, MICROSERVICES ARCHITECTURE

The object of research – self-service checkout for small and medium businesses.

The subject of development is an informational system of self-service checkout.

The purpose of the work is to increase the efficiency of small and medium businesses by implementing a flexible component of a self-service checkout information system.

Development methods include a systematic approach and structural system analysis methods, modeling and design methods for complex systems, cloud technologies (Amazon Web Services) relational databases, All Fusion Process Modeler (BPWin) as a CASE-tool for functional modeling, and design methods for server applications for Internet networks, specifically, such as, service-oriented architectural approach, three-tier architecture and the MVC architectural pattern.

A web application based on a three-tier client-server architecture has been created. The developed system has implemented automation of the main business process of the self-service checkout system.

The area of application is e-commerce support within a self-service checkout information system.

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

БД – база даних;
ІС – інформаційна система;
ритейлери – роздрібні торговці;
СУБД – Система Керування Базами Даних;
чек – квітанція;
штрих-код – баркод;
AJAX – Asynchronous JavaScript And XML;
API – Application Programming Interface;
AWS – Amazon Web Services;
CSS – Cascading Style Sheets;
CASE – Computer-Aided Software Engineering;
DAO – Data Access Object;
DTO – Data Transfer Objects;
DI – Dependency Injection;
EC2 – Elastic Compute Cloud;
GCP – Google Cloud Platform;
HTML – HyperText Markup Language;
HTTP – HyperText Transfer Prorоco;
IoC – Inversion of Control;
JDBC – Java DataBase Connectivity;
JPA – Java Persistence API;
JS – JavaScript;
MVC – Model-View-Controller;
NFC – Near-field communication;
ODBC – Open DataBase Connectivity;
RDS – Relational Database Service;
S3 – Simple Storage Service;
SQL – Structured Query Language;
UX – User Experience;
UI – User Interface (інтерфейс користувача);
VPC – Virtual Private Cloud;

ЗМІСТ

# ВСТУП

Останніми роками касові системи самообслуговування набувають все більшого поширення, особливо в секторі роздрібної торгівлі. Вони являють собою значну еволюцію в обслуговуванні клієнтів, дозволяючи підприємствам підвищити ефективність, зменшити черги і потенційно знизити витрати на робочу силу. З розвитком технологій та зміною очікувань споживачів, актуальність і необхідність цих систем стала ще більш очевидною. У сучасному динамічному суспільстві споживачі цінують швидкість і зручність, що робить касові системи самообслуговування невід'ємною частиною сучасного світу.

З кожним роком актуальність подібних систем зростає. І одним з найголовніших чинників цього є рівень розвитку безготівкової оплати в регіоні, де впроваджується система. Актуальність концепції касового самообслуговування також посилюється завдяки зростанню популярності цифрових оплат та пов'язаних з ними технологій. Розвиток способів безготівкової оплати набирає популярності, а технології, такі як NFC, стають все більш доступними. Хоча і існують системи, які можуть приймати готівку, але вони можуть бути складнішими в обслуговуванні та дорожчими. Тому популярність безготівкових платежів впливає на актуальність впровадження систем самообслуговування [1].

Іншим чинником, що впливає на актуальність впровадження касової системи самообслуговування є зростаюча тенденція робити менші та частіші покупки. За останні роки переважна більшість покупок включає меншу кількість товарів, що зробило системи самообслуговування особливо зручними для користувачів. Це обумовлено тим, що з кожним роком кількість невеликих торгових точок поступово зростає. Це робить бажані товари для клієнта більш доступними, що в свою чергу призводить до того, що споживачу зручніше робити менші покупки, але частіше. Тому, щоб адаптуватися до цієї тенденції, багато магазинів адаптують свої стратегії продаж. Системи самообслуговування працюють надзвичайно добре в таких умовах, забезпечуючи швидку та зручну процедуру оплати [1].

Однак поширення касових систем самообслуговування не позбавлене проблем. Сучасні системи часто стикаються з такими проблемами, як технічні збої, підвищений ризик крадіжки та опір з боку менш обізнаних у техніці

користувачів. Крім того, системам може бути важко обробляти складні транзакції, пов’язані зі зваженням товарів, різними типами скидок чи купонів або продуктами з віковими обмеженнями, таких як алкоголь чи тютюн.

Розробка повноцінної системи подібного класу на глобальному рівні є дуже складною, і це може призвести до припущення різних архітектурних помилок в розробці, а це в свою чергу нерідко призводить до того, що витрачені кошти на створену систему в декілька разів менше, ніж кошти на подальшу підтримку її працездатності. Враховуючи цей фактор та неможливість придбання або переходу до глобальних рішень [2] для невеликих мереж магазинів (торгових точок), що за статистикою перевірених джерел [3] є найбільш актуальним на сьогодні, важливим є проаналізувати та знайти вузькі місця які характеризують складність системи, та якомога більше спростити систему в концептуальному плані не звужуючи відповідних функціональних вимог предметної області.

Застосування методологій системного проектування може суттєво допомогти в розробці касової системи самообслуговування. Системне проектування — це методичний процес проектування того, як компоненти взаємодіють для досягнення бажаного результату або функції. Це передбачає розуміння компонентів системи, їх взаємозв’язки, потреби користувачів та потенційні проблеми. Використання методологій системного проектування дозволяє створити інтуїтивно зрозумілий інтерфейс користувача та забезпечити безперешкодну інтеграцію з існуючими системами. Такий цілісний підхід гарантує, що кожен аспект досвіду самообслуговування, від взаємодії програмного та апаратного забезпечення до дизайну користувацького інтерфейсу, буде враховано та оптимізовано. Забезпечення легкої інтеграції між усіма частинами системи може підвищити ефективність, задоволеність клієнтів і, зрештою, продуктивність бізнесу.

Об'єкт дослідження є касове самообслуговування для середнього та малого бізнесу. Предмет розробки є інформаційна система касового самообслуговування. Метою роботи є підвищення ефективності середнього та малого бізнесу шляхом реалізації масштабованої інформаційної системи касового самообслуговування у вигляді веб-застосунку з використанням сучасних методологій системного проектування та відповідних засобів автоматизації проектування та розробки.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Змістовний опис касових систем самообслуговування

При розробці будь якої системи важливо розуміти до якого типу інформаційних систем відноситься розроблювана. Це може значно спростити розробку в подальшому, так як допоможе обрати правильну архітектуру, засоби розробки та інше на базі існуючих методологій. Тому обираючи тип розроблюваної інформаційної системи на основі проведеного аналізу предметної області (підприємства) було доведено до висновку, що інформаційна система касово-розрахункового обслуговування відноситься до типу систем електронної комерції (E-Commerce) [4]. Це обумовлено тим, що дана інформаційна система включає в себе, як сукупність технічних так і організаційних форм, призначених для ведення комерційної діяльності й здійснення угод з використанням електронних систем в цифровій мережі. Також представляє з себе як засіб взаємодії з партнерами, банком, постачальниками, споживачами товарів і послуг та інше. Зазвичай у системах такого класу присутні всі етапи здійснення угоди, а саме, наприклад: пошук бажаної продукції чи послуги, уточнення деталей, оплата, можливість скасування угоди та інше.

Поміж наведених вище характеристик, що притаманні системам електронної комерції, також існують певні підтипи даного типу системи, а саме: електронний бізнес (E-Business), електронної комерції типу B2C (Business-to-Consumer) та типу B2B (Business-to-Business) [5]. Кожна з них досить близька до електронної комерції, відрізняються лише існуючими компонентами та взаємодіями між ними, наприклад в електронної комерції типу B2C відбувається взаємодія між організацією (Business) (зазвичай в особі інтернет-магазину), та часткам, так званим кінцевим споживачем (Consumer), а наприклад в електронної комерції типу B2B відбувається наприклад взаємодія між двома фірмами чи підприємствами.

Будь-яка із подібних систем складається з наступних ключових компонентів: інтерфейс користувача, сканер, платіжний термінал, принтер чеків, комп'ютер, програмне забезпечення.

Залежно від складності системи, вона також може містити специфічний функціонал, такий як: сигналізація проти крадіжки, ваги, конвеєрна стрічка, зона

пакування, а також автоматична система прийому готівки з обробкою та видачею купюр чи монет.

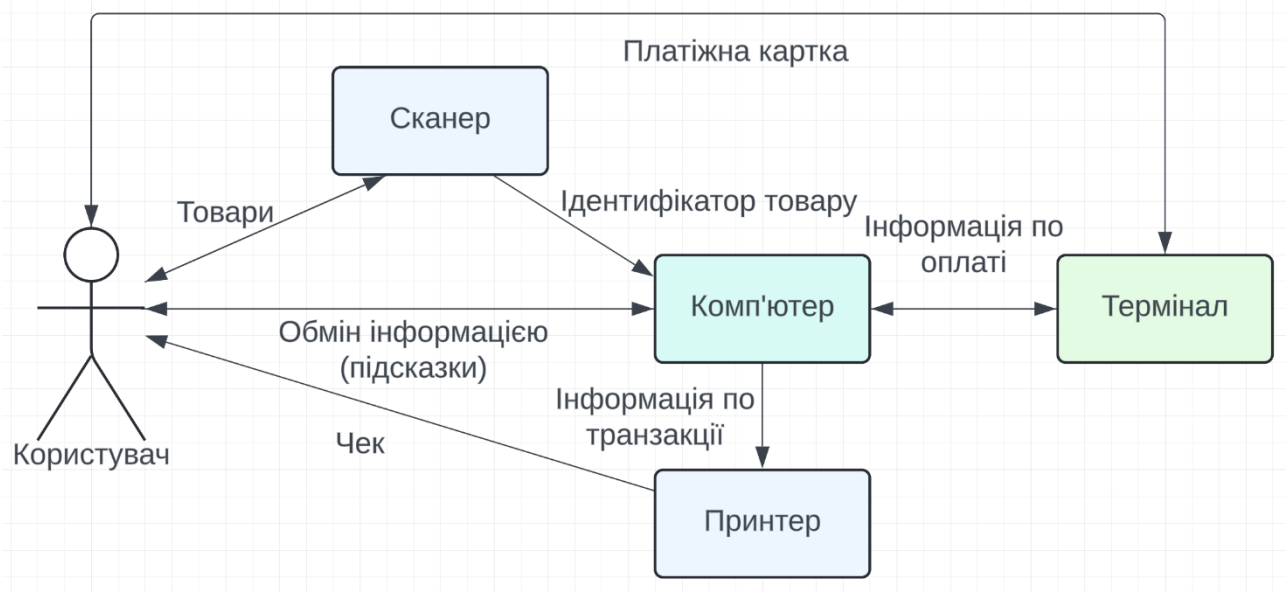



Рисунок 1.1 – Схема взаємодії користувача з розрахунково-касовою системою

### 1.1.1 Інтерфейс користувача

Інтерфейс є ключовим компонентом кожної системи обслуговування, адже він створює основні канали взаємодії між користувачем і технічним обладнанням. В касовій системі самообслуговування, інтерфейс формується не лише програмним забезпеченням, але й взаємодією з технічними компонентами. Зокрема, це включає дисплей, зазвичай із сенсорним екраном, через який користувач отримує деталізовані інструкції щодо відсканованих товарів, їхню та загальну вартість, можливі способи оплати та ін. Таким чином, інтерфейс забезпечує ефективну взаємодію користувача з системою, спрощуючи процес самообслуговування.

Це, можливо, один з найважливіших компонентів даної системи, оскільки дизайн та зручність мають вирішальне значення для забезпечення високого рівня лояльності користувачів. В ідеалі, інтерфейс користувача має бути інтуїтивно зрозумілим та достатньо простим, щоб користувач без попереднього досвіду чи навчання міг ним вільно користуватися. Це критично важливо, бо якщо інтерфейс користувача буде занадто складний або важкий у використанні,

користувачі можуть витратити надто багато часу на освоєння системи, або навіть відмовитися від її використання.

Відповідний колір, якась анімація чи звук можуть використовуються в ефективному дизайні інтерфейсу користувача. Такі складові можуть зробити інтерфейс більш привабливим та допомогти користувачам в процесі. Наприклад, відсканований товар може супроводжуватися звуковою підказкою чи якоюсь анімацією руху товару у віртуальний кошик для покупок.

Також інтерфейс користувача має відповідати певним нормам з урахуванням людей з обмеженими можливостями, враховуючи такі аспекти, як висота екрана, розміри кнопок, контрастність кольорів, розмір шрифту, налаштування сенсора тощо.


Рисунок 1.2 – користувацький інтерфейс на прикладі екрану

### 1.1.2 Комп'ютер та програмне забезпечення

Основна задача програмного забезпечення це обробка платежів, надання зручного інтерфейсу та контроль процесу самообслуговування. В ці задачі входить: керування всіма аспектами транзакцій, підказки користувачу через інтерфейс, керування сканером, обробка платежу, інтеграція з банківським програмним забезпеченням, видача квітанції та багато чого іншого в залежності від складності системи.

Також розповсюдженою задачею програмного забезпечення в касових системах є звітності та аналітики. Відповідна інформація може бути використана для створення різних моделей, що може дати уявлення про структуру продажів, оборотність запасів і поведінку клієнтів, що може бути корисним для бізнес-планування та прийняття рішень. Також цю інформацію можна використати для управління запасами на складі.

## 1.2 Основні переваги кас самообслуговування

Однією з головних переваг касових систем самообслуговування є їхня економічність. Вони досить швидко окупаються, заощаджуючи гроші магазину за рахунок зменшення потреби у певній кількості оплачуваних працівників (касирів).

Іншим, не менш важливим плюсом цих систем є економія місця або більш ефективна організація простору. Каси самообслуговування, як правило, займають менше місця, ніж традиційні каси, що дозволяє виділити більше простору для демонстрації продуктів або ж додати додаткові каси, що може збільшити пропускну спроможність магазину та підвищити кругообіг клієнтів. Це також може пришвидшити процес чекауту з точки зору користувача, так як вірогітність того що одна з кас самообслуговування буде вільною або мати мінімальну чергу буде вищою порівнюючи з аналогічним рішенням через класичні каси.

Також це нові можливості в маркетинговому плані за рахунок інтеграції програм лояльності користувачів. Яскравим прикладом є можливість вводу якогось купону пере оплатою, для отримання скидок, або якихось бонусів. Або це може бути якийсь купон у вигляді штрих-код або QR-код, власник якого може його відсканувати тим же сканером, що і сканує товари, та отримати відповідну вигоду.

## 1.3 Основні недоліки та актуальні проблеми кас самообслуговування

У контексті систем касового самообслуговування, фундаментальним завданням є розробка інтуїтивно зрозумілого користуванні інтерфейсу. Основну

проблему сучасних систем самообслуговування складає отримання відмови в користуванні від потенційних користувачів через складний інтерфейс. Кінцеві користувачі, особливо старшого віку, можуть відчувати незручності у роботі з цими системами, оскільки вони зазвичай вважають ці системи "не для них" через складність їх використання.

Це саме стосується впровадження систем самообслуговування у побутові сфери де потребується частіше взаємодія між людиною та машиною. Люди зазвичай орієнтовані на традиційне обслуговування, яке прийнятніше для них, проте з часом це може змінитися, оскільки ефективність цих систем впроваджується через інноваційні технології які становляться все більш досконалими.

Хоча, не варто залишати осторонь різні технічні проблеми та інші незначні питання, наприклад, можливість крадіжок, труднощі з обробкою певних видів товарів або різних типів знижок. Однак, ці проблеми не є важливими для основної задачі - розробки зручного інтерфейсу. Ці проблеми, безумовно, варто вирішувати, але вони не впливають на основну трудність розробки системи - неінтуїтивний інтерфейс.

Тому, основне увагу при проектуванні системи касового самообслуговування було б варто приділити виключно зручності користування — це є ключовим фактором для успіху такої системи.

# 2 ВИЯВЛЕННЯ ВИМОГ ДО СИСТЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

## 2.1 Формування вимог до інформаційної системи

### 2.1.1 Аналіз цільової аудиторії та модель користувача

Цільова аудиторія користувачів системи касового самообслуговування орієнтована на відвідувачів різного віку, соціального статусу та статі. Було виділено п'ять основних групи відвідувачів за віковою категорією, а саме:

– 7-12 років: для дітей система касового самообслуговування може стати привабливою іграшкою. Вони швидко навчаються новому і можуть використовувати систему під наглядом батьків;

– 13-17 років: підлітки зазвичай активно використовують технології в повсякденному житті, включаючи мобільні пристрої та гаджети, тому вони з легкістю зможуть освоїти систему касового самообслуговування;

– 18-35 років: молодій аудиторії віком від , яка активно використовує техніку і готова до нововведень;

– 35-55 років: відвідувачам середнього віку може бути менш інформованою щодо використання системи, але після короткого інструктажу вона зможе користуватися системою;

– від 55 років: люди цієї вікової категорії мають вищий поріг відмови від системи через зміст та складнощі в навчанні користування новітніми технологіями.

Що до особливостей знань користувача системи, це мінімальні знання в сфері обслуговування.

В результаті проведеного аналізу цільової аудиторії була побудована спрощена описова модель користувача (профіль):

а) Соціально-демографічні характеристики:

– вік: від 7 років;

– стать: будь яка;

– основна мова: державна мова відповідно до локації торгової точки;

– рід занять: будь який;

б) рівень комп'ютерної грамотності: низький;

в) оточення:

– робоче місце: спеціальний стіл, що передбачає такі компоненти, як площадка для розміщення товарів, термінал самообслуговування та пакувальний стіл;

– конфігурація обладнання: локальна мережа; комп’ютер, з мінімальними характеристиками (тактова частота центрального процесора (CPU): 1.5 Ггц; кількість ядер центрального процесора: 4; обсяг оперативної пам'яті комп'ютера (ОЗП): 4 GB; розмір вільного простору на жорсткому диску або SSD: 50 GB;

– операційна система: будь яка, де є доступ до браузера.

### 2.1.2 Аналіз бізнес-процесів та бізнес-функцій

Аналізуючи предметну область було виявлено та уточнено ряд бізнес-процесів та ряд бізнес-функції, що входять до них. На основі поставленого ряду бізнес-процесів та бізнес-функцій та виявлених груп користувачів, розроблені можливі сценарії поведінки користувачів системи касового самообслуговування. В результаті було виявлено, що ключовим профілем системи є користувач.

Прецедент «Зробити покупку» є головним бізнес-процесом системи (основна логіка), що складається з послідовності дій:

– додати товар до кошика. Може бути двома способами:

1) режим сканування штрих-коду, користувач за допомогою спеціального сканера може зчитувати штрих-код та автоматично додавати відповідний товар до кошика;

2) у разі пошкодження штрих-коду та неможливості його відсканувати сканером, обраний товар можна додати до кошика «вручну», знайшовши його за певним параметром (назва товару, його штрих-код, то що). В результаті буде здійснено пошук в каталозі товарів та виведено результат (список відповідних товарів, що задовольняють обраному товару) у відповідному вікні. Також користувач може обрати (ввести) кількість знайденого товару та додати його до кошика. Кошик представляє з себе спеціальний контейнер що зберігає певний товар та його кількість. Після кожного закриття покупки даний контейнер очищується.

– перегляд та можливе редагування кошика: користувач може переглянути список доданих товарів до кошика, та за потребі редагувати його (змінювати кількість товарів чи зовсім видаляти бажаний з них);

– вибір способу оплати: користувач обирає спосіб оплати електронними коштами (карткою). Далі, при закритті покупки підключається інтерфейс сервісу електронної оплати;

– закрити покупку: закриття замовлення (формування чека), транзакція яка включає в себе інтерфейс сервісу електронної оплати;

– перегляд створеного чеку (опціональна бізнес функція): користувач за бажанням може переглянути створений чек після закриття покупки (в практичному плані (сенсі) його потрібно друкувати для покупця).

Детальна Use Case Diagram для користувача, з коментарями що до можливих сценаріїв поведінки наведена на рисунку 2.1. Дана діаграма була побудована використовуючи безкоштовний програмний засіб StarUML [6]. Це чудовий CASE-засіб, що забезпечує не тільки зручне UML моделювання, а й підтримує функціє генерації коду та багато чого іншого.

Однак варто зазначити, що бізнес-процес «Зробити покупку» повної автоматизації не підлягається (без участі робітників підприємства). Оскільки, за поставлених умов, таких як доступні технології на даний момент та вимоги законодавства, це або вкрай важко реалізувати, або навіть неможливо. Проблема полягає в неможливості ідентифікувати покупця, зокрема його вік, у разі придбання ним товарів з якимось віковим обмеженням, таких як алкоголь чи тютюн. Насправді ж, найбільш економічно вигідним рішенням даної проблеми буде залучення робітників відповідного підприємства до даного процесу (контролер кас самообслуговування). Наприклад це магазин, звісно в залежності від магазину, але в більшості випадків, відсоток покупок з такими товарами значно менший ніж покупок з товарами без обмежень, тоді даний процес підтвердження покупки не потребує значної зайнятості контролера каси самообслуговування. Це дозволяє розділити обов’язки одного робітника між декількома касами самообслуговування, або ж залучити робітників які мають посередній рівень зайнятості, наприклад, в супермаркеті це може бути охоронець, чи наприклад, робітник що поповнює товари на поличках. В технічному плані, дана операція підтвердження зазвичай відбувається за рахунок

сканування спеціального штрих-коду, який може бути розміщений на бейджику, та який ідентифікує робітника який це підтвердив.

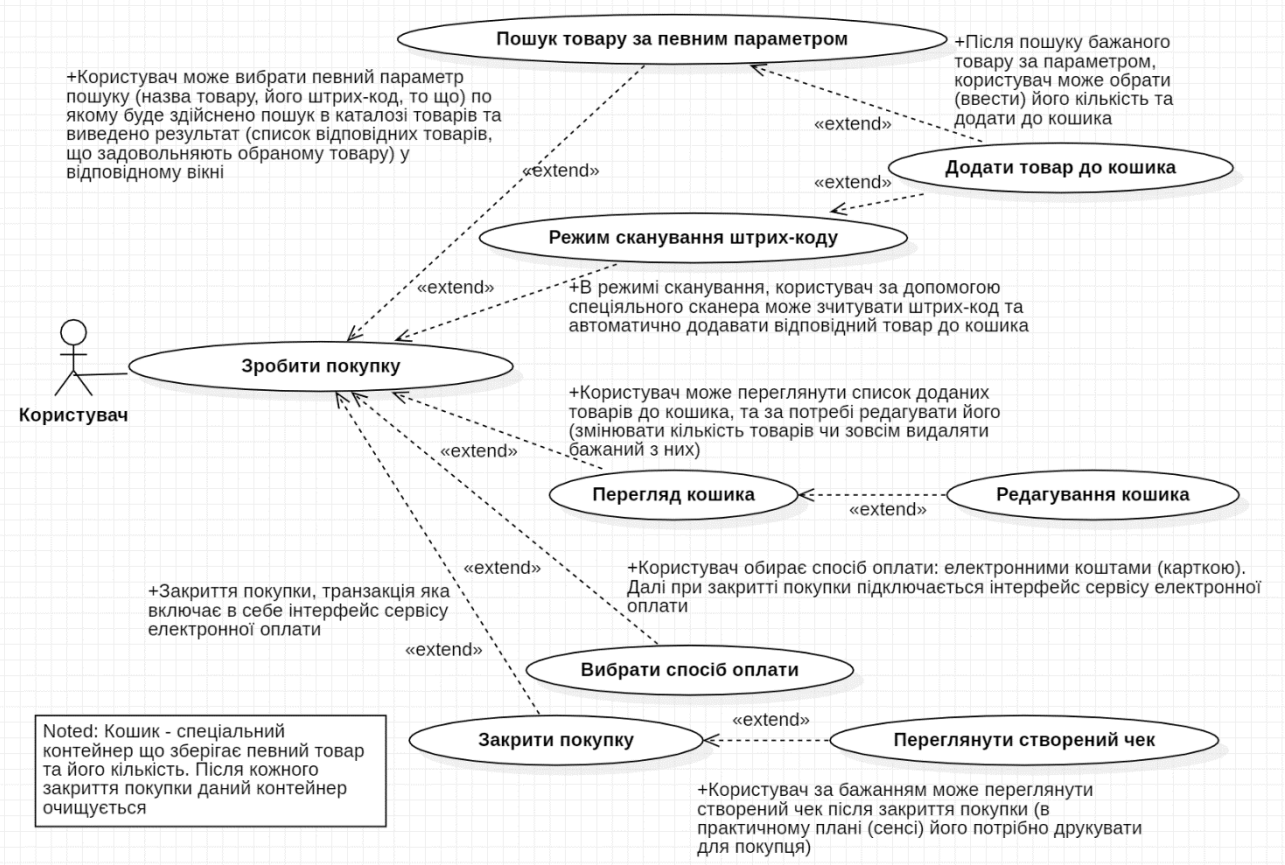



Рисунок 2.1 – Use Case Diagram

### 2.1.3 Функціональне моделювання

На основі отриманих функцій системи, в результаті аналізу предметної області, було проведено функціональне моделювання системи та виділено ключовий бізнес-процес та бізнес-функції, що входять до нього, та підлягають автоматизації. А саме — це бізнес-процес «Зробити покупку». Для відображення бізнес-процесу, його основних компонентів та життєвого циклу, була створена концептуальна діаграма згідно до стандарту методології IDEF0.

Для побудови даної діаграми було використано одних з найвідоміших автоматизованих засобів проектування та розробки, а саме: пакет All Fusion Process Modeler (BPWin) [7]. Даний CASE-засіб дозволяє проводити моделювання з використанням різних стандартів, таких як IDEF0, IDEF3, DFD.

До блока головного бізнес-процесу відносяться визначені входи та виходи, механізми, які на створеній діаграмі представляють користувачів системи, та управління (рисунок 2.2).

Входи:

– товари;

– платіжні реквізити;

– уподобання користувача (на основі яких користувач обирає спосіб оплати).

Виходи:

– сформований чек;

– продажі (звіт).

Механізми:

– користувач (покупець);

– програмне забезпечення.

Управління:

– каталог товарів (зареєстровані товари в системі);

– регламент роботи (як самої торгової точки так і її персоналу, що бере участь у обслуговуванні кас, вчасності це підтвердження покупок з товарами з віковим обмеженням);

– вимоги законодавства.

Одним з найважливіших пунктів управління є законодавчі вимоги. В підприємствах, які безпосередньо взаємодіють хоча б з якимось з підтипів реєстратору розрахункових операцій, часто фігурують наступні нормативні документи (законодавчі та під законодавчі акти) [8], що регламентують його діяльність, а саме в першу чергу — це акти «Про захист прав споживача» [9], «Про захист персональних даних» (конфіденційна інформація) [10].

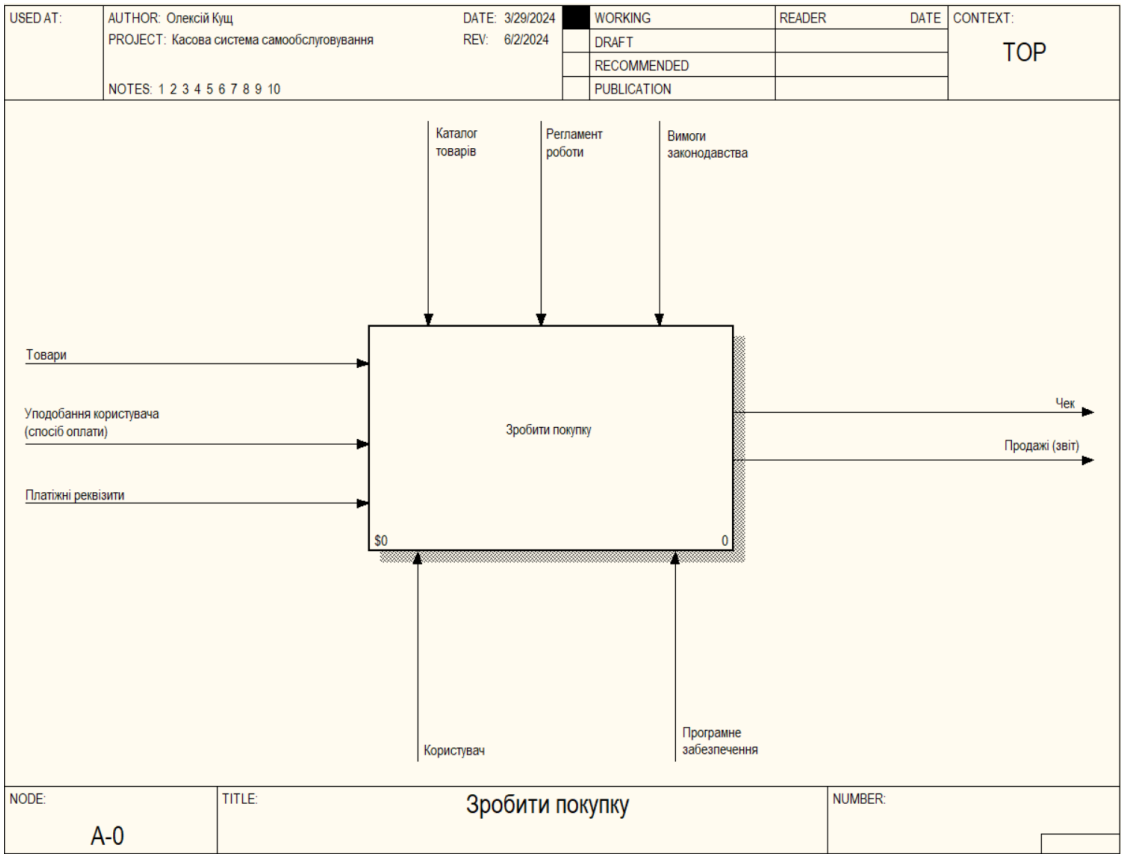



Рисунок 2.2 – Контекстна діаграма IDEF0 бізнес-процесу «Зробити покупку»

Далі, в ході аналізу ключового бізнес-процесу було виділено основні бізнес-функції які забезпечують його протікання. Була проведена декомпозиція ключового бізнес-процесу на бізнес-функції (рисунок 2.3). В результаті, було виділено чотири бізнес-функції, а саме:

– блок A1 «Оформлення покупки»;

– блок A2 «Облік покупки».

Блок A1 являється ключовим, та представляє з себе функцію формування покупки на стороні клієнта, та подальша його обробка на стороні сервера. В це входить, формування кошику, вибір способу оплати, сама оплата, різні етапи валідації та інше.

Блок A2 — це функція фінальної обробки покупки. В першу чергу це формування звіту для покупця (чеку). В якості вхідних даних приймає результат відпрацювання (вихід) блоку A1, а саме, оплачена покупка. Дана бізнес-функція також може включати різні сторонні функції збору даних та інформації, їх аналізу, створення різних звітів, що до популярності деяких товарів, та інше. Це

також може бути пов’язано зі складом торгової точки. На основі отриманих звітів, може бути реалізовано динамічне поповнення запасів на складі.

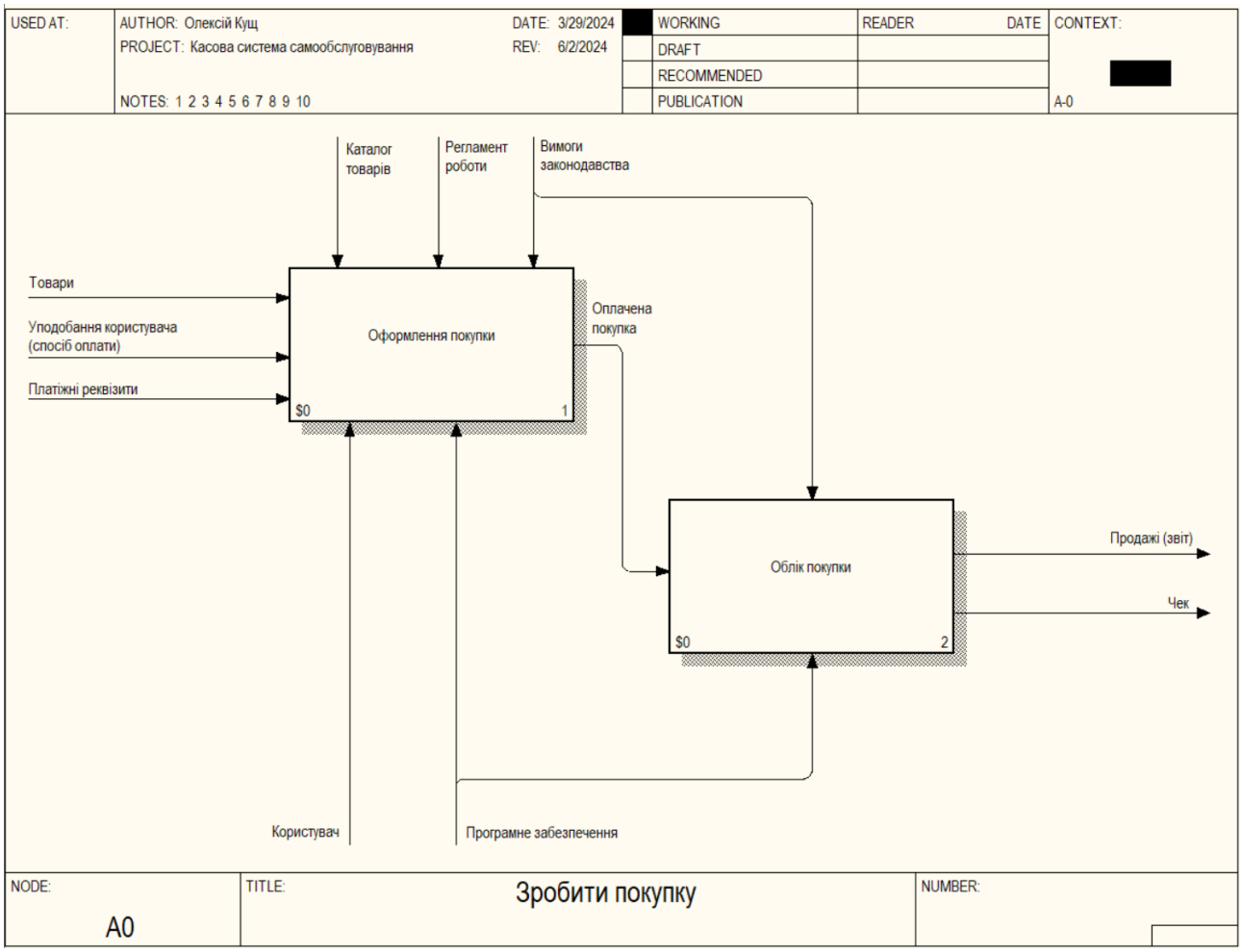



Рисунок 2.3 – Декомпозиція бізнес-процесу «Зробити покупку»

З метою конкретизації інформації стосовно бізнес-функцій та чіткого визначення ролі кожної окремої бізнес-функції в процесі виконання бізнес-процесу було проведено декомпозицію блоків «Оформлення покупки» та «Облік покупки» (рисунки 2.4 – 2.5).

Перший процес, «Оформлення покупки», щільно пов’язаний з користувачем, та кожний із його підпроцесів залежить від нього. В першу чергу, це формування кошика, користувач сканує свої товари та система заносить їх до його віртуального кошику. По завершенню даного етапу, при переході до оплати, товари у кошику піддаються перевірці, в першу чергу, це перевірка на вікове обмеження. Система перевіряє кожний з товарів у кошику, та при виявленні хоча б одного відповідного товару, процес повертається до етапу формування кошику,

чекаючи підтвердження віку покупця. У разі ж проходження перевірки, процес переходить до уточнення способу оплати. Останнім же підпроцесом являється сама оплата, що виконується через платіжний термінал. Успішно сплачений товар, формує завершену (оплачену) покупку, що готове перейти до наступного бізнес-процесу, а саме «Облік покупки». Цікавим є те, що другий процес, «Облік покупки», виконується без участі користувача. Основним в ньому є формування чеку для користувача, та можливе збереження якихось даних для подальшого аналізу. Все це, може бути повністю автоматизованим, та ні як не залежати від користувача.

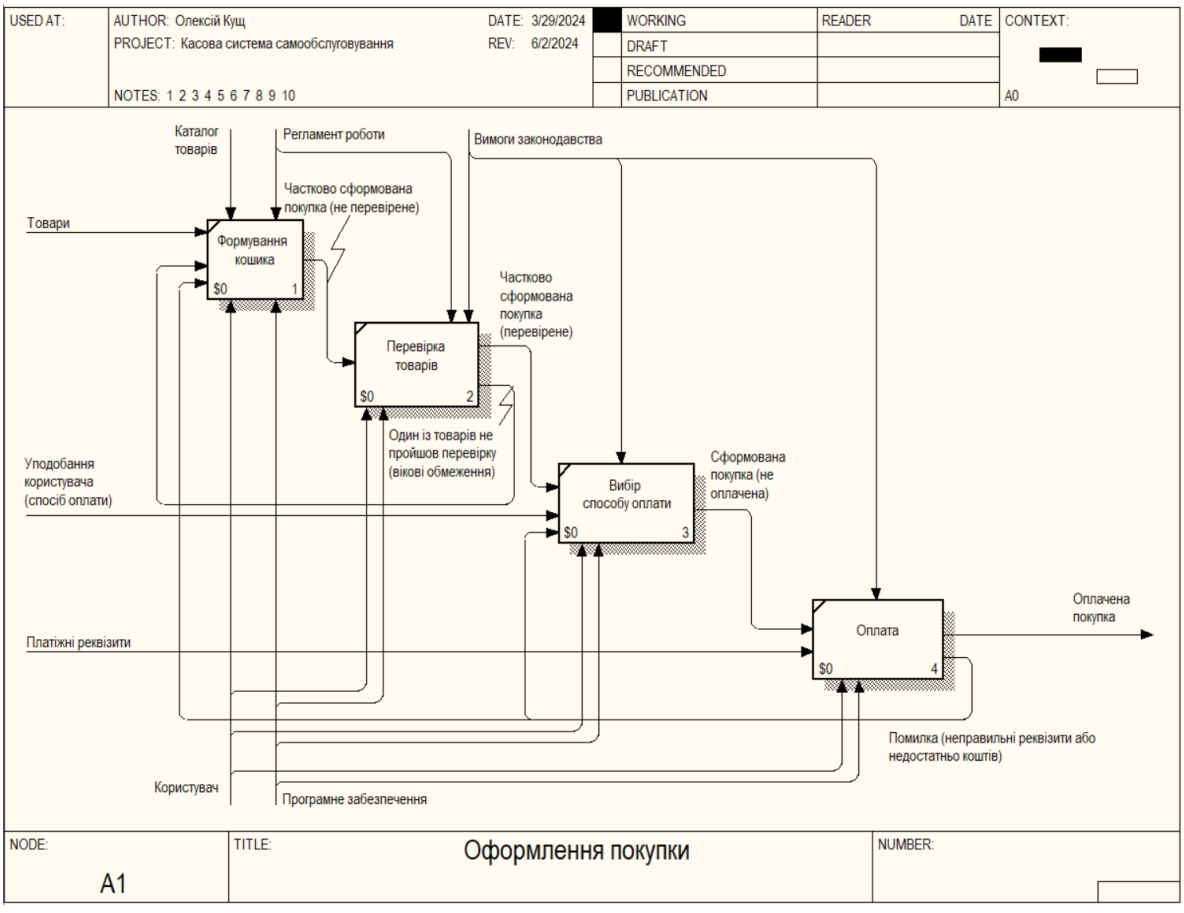



Рисунок 2.4 – Декомпозиція процесу A1 «Оформлення покупки»

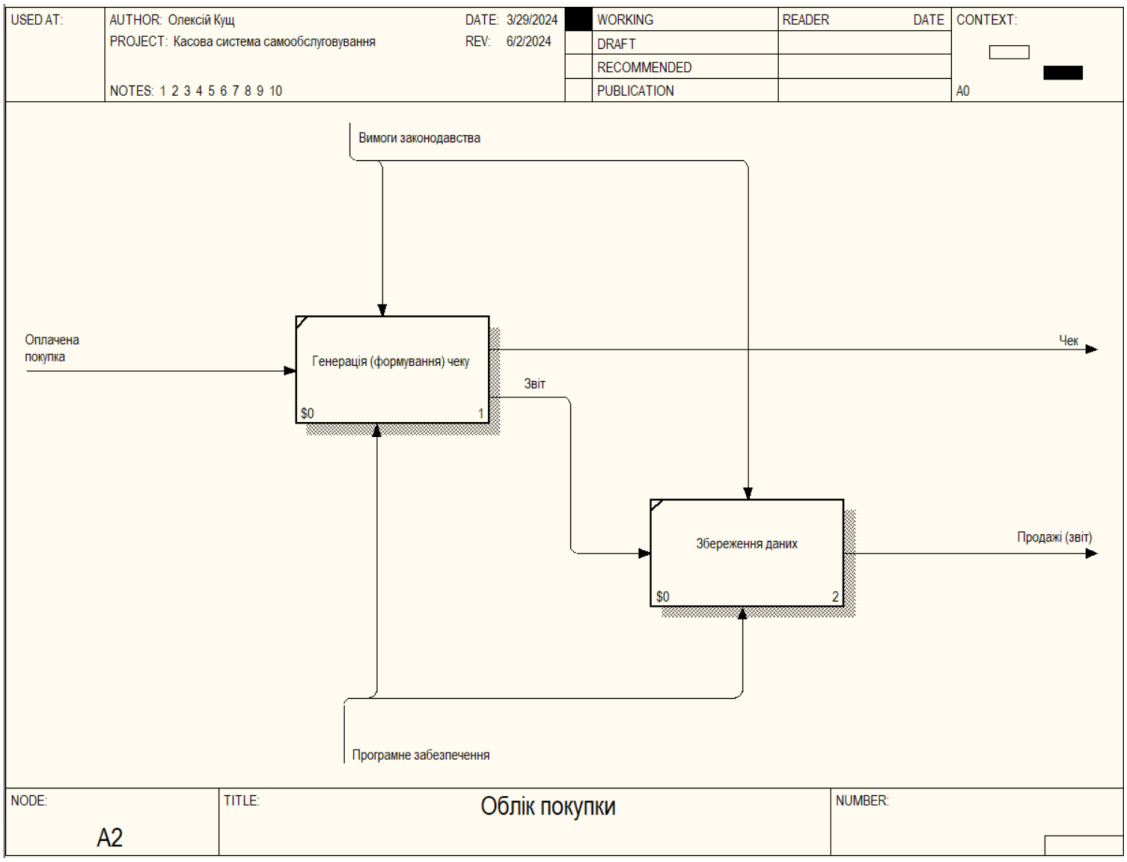



Рисунок 2.5 – Декомпозиція процесу A2 «Облік покупки»

Також, для більшої наглядності, була згенерована діаграма дерева вузлів (процесів та функцій) ключового бізнес-процесу (рисунок 2.6).

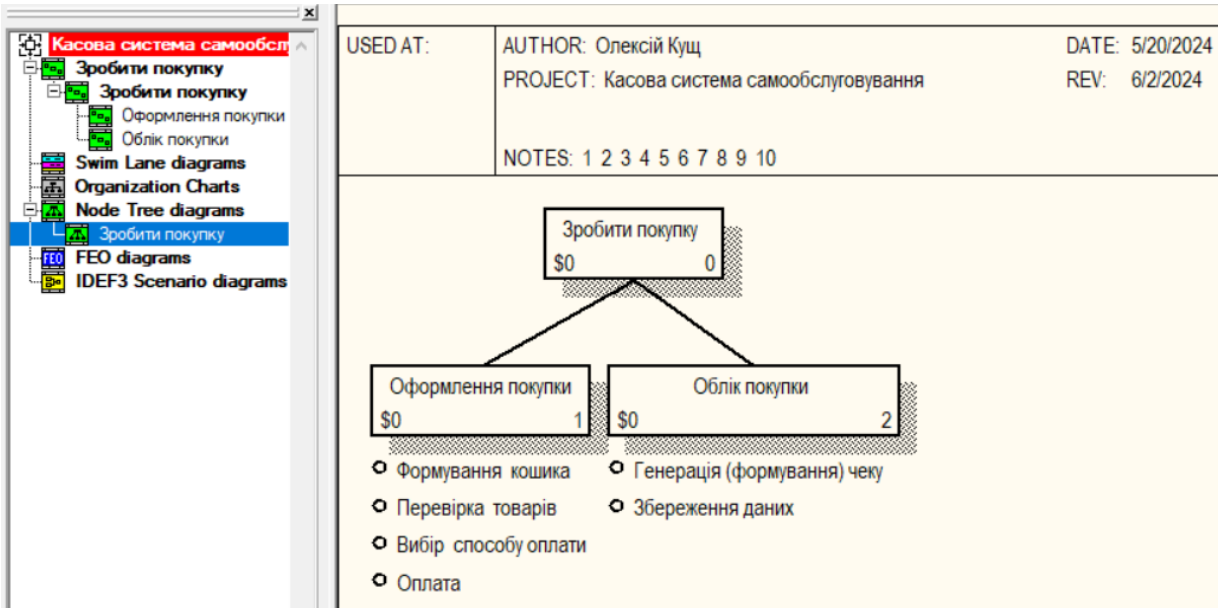



Рисунок 2.6 – Діаграма дерева вузлів бізнес-процесу «Зробити покупку»

## 2.2 Постановка задачі

Завданням роботи є аналіз, системне проектування та розробка компонентів інформаційної системи касового самообслуговування, а саме це реалізація механізму зберігання даних, реалізація функціональних вимог поставлених предметною областю та реалізація користувацького інтерфейсу. Розроблена система має підлягати ряду критеріїв якості, а саме, таких як: масштабованість, продуктивність, безпека, розширюваність, надійність та інше.

Користувачами системи є покупці.

Метою розробки системи є створення веб-сервісу самоконтролю за придбаними товарами чи послугами для малого та середнього бізнесу, зокрема невеликих мережевих магазинів з кількома торговими точками. Вибір даної аудиторії обумовлений тим, що підприємства такого рівня зазвичай стикаються з проблемами в придбанні чи переходу до більшості глобальних рішень [2] через фінансові обмеження. Тому для подолання цієї проблеми важливим є розробити максимально доступне та економічно ефективне рішення, але яке водночас буде покривати мінімально необхідний функціонал будь якої касової системи самообслуговування.

Функціональні вимоги системи:

– вибір локалізації (державна мова та будь яка інша міжнародного рівня: Українська, Англійська);

– зробити покупку;

1) додати товар до кошика. Сканування товару та автоматичне занесення його до віртуального кошика. Додатковий пошук товару «вручну». Пошук повинен бути можливий за декількома параметрами, такими як: назва товару, його виробник та цифровий штрих код товару. Знайдені товари мають відображатися з наступними ідентифікуючими полями, такими як: зображення товару, його назва, виробник, ціна, штрих-код;

2) перегляд кошика (відображення наступних полів товару: зображення, назва, виробник, загальна сума по певному товару, вартість одного товару, кількість товарів у кошику, штрих-код відповідного товару);

3) редагування кошика (зміна кількості товарів у кошику, видалення певного товару з кошика);

4) відмінити покупку (система готова до нової покупки, попередній стан системи був очищений);

– зміна стилю інтерфейсу.

Вимоги до розроблюваного інтерфейсу:

1) Максимально зрозумілий інтерфейс;
2) Навчання (наявність повної документації та системи контекстно-залежної допомоги, простота навчання);
3) Локалізація інтерфейсу для декількох різних мов, та відповідна функція зміни мови інтерфейсу за бажанням користувача;
4) Робота з системою:

– одноманітний інтерфейс (однаковість елементів інтерфейсу, дотримання стилю);

– зрозумілість критичних моментів для клієнта (при «ручному» пошуку товару чи при перегляді кошику, можливість чітко ідентифікувати товар по його зображенню, назві та виробнику товару, та специфічні поля для кошика як: кількість певного товару в кошику, ціну за умовну одиницю, сума по певному товару та загальну ціну кошика);

– обробка помилок, представлення конструктивних коментарів відповідно до викликаної помилки;

– можливість відмінити дію та підтвердження відміни;

– підтвердження усіх безповоротних операцій, таких як «Видалення товару з кошика» чи «Відмінити покупки»;

– перевірка стану покупки для запобігання каскадних помилок (неможливість продовжити оплату покупки у випадку якщо кошик пустий, наприклад, кнопка переходу до оплати має бути не активна);

– налаштованість інтерфейсу, зміна певних елементів інтерфейсу, їх кольору чи стилю;

– кросбраузерність інтерфейсу, коректне відображення в будь-яких браузерах;

– адаптованість інтерфейсу під різні розширення екранів.

5) У пріоритеті мінімалістичний дизайну.

# 3 ПРОЕКТУВАННЯ СИСТЕМИ

## 3.1 Опис обраної архітектури для побудови системи, її компоненти та їхнє призначення

### 3.1.1 Опис інфраструктури системи

Проаналізовавши поставлені вимоги до розроблюваної системи, було обрано класичну для веб додатку архітектуру, а саме — клієнт-серверну. Даний підхід неймовірно розповсюдженій, і на справді на цій архітектурі базується увесь Інтернет [11]. За часту, в даній архітектурі фігурує три основних компоненти: клієнт (зазвичай веб-браузер), сервер (якась віддалена запущена, на якій запущена основна програма — вебсервер) та можливе, але не обов'язкове, якесь сховище для зберігання даних (рисунок 3.1).

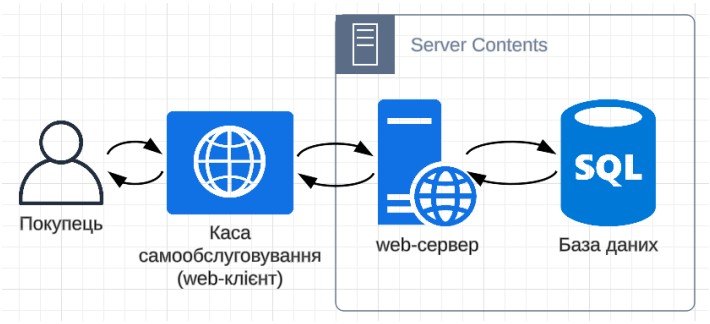



Рисунок 3.1 – Компонентна діаграма системи (базова версія)

Клієнтом, в даній архітектурі, може бути не тільки веб-браузер на комп'ютері, а це може бути і набор різних пристроїв або інтерфейсів. У світі Інтернету речей (IoT) [12] «клієнтом» може бути що завгодно: від смартфона до розумного холодильника, чи навіть машини для виробництва морозива. Це також стосується і пристроїв що нам потрібні, таких як: сканера, принтера, POS-термінал. Але, все ж таки, ця можливість, відправляти запити на пряму безпосередньо до сервера, в даному випадку не дуже зручне рішення. Зручнішим буде, підключити всі ці пристрої, як механізми вводу, до веб-браузера, та як би підключитися до сервера через нього. Це зручно, тому що веб-браузера має чудові механізми реалізації інтерфейсу, на відміну від більшості пристроїв. Що

в свою чергу, відкриває можливості якось візуалізувати цю інтеграцію між пристроями, у разі необхідності. Насправді, багато пристроїв можуть підключатися до сервера через веб-браузер. Наприклад, багато сучасних принтерів та сканерів мають веб-інтерфейси для легкого налаштування та керування. Після підключення до Інтернету ви можете відкрити браузер, перейти на призначену локальну IP-адресу та взаємодіяти з елементами керування пристроєм прямо в браузері, забезпечуючи зручний та інтуїтивно зрозумілий інтерфейс користувача. Веб-браузер же тут буде відігравати роль такого централізованого вузла вводів / виводів.

Взаємодія між браузером та вебсервером відбувається наступним чином. Якщо спрощуючи, то коли до браузеру надходить якась URL-адреса, то він надсилає запит на вказаний сервер, де вже сервер знаходить по запиту, те що потрібно клієнту. Зазвичай сервер формує в якості відповіді повноцінну вебсторінку, в процесі, звертаючись до сховища даних за відповідними до запиту даними, наприклад якісь файли, картинки, текст, тощо. Нарешті, коли браузер отримує цю відповідь від сервера, він її відображає.

Ця відносно проста інфраструктура надзвичайно ефективна для обробки кількох одночасних запитів та розподілу їх між серверами, у разі необхідності. Вона надійна та безпечна, оскільки елементи керування можна застосувати на кожній кінцевій точці, і досить зручно масштабується, оскільки ви можете додавати сервери відповідно до потреб.

Однак, як вже згадувалося раніше, також, вкрай важливим для даної системи, її архітектури, мати високу гнучкість, щоб в майбутньому була можливість максимально безперешкодно інтегрувати додаткові сервіси та сформувати повноцінну систему, з якою можна буде вийте на відносно глобальний рівень.

Одним з можливих рішень даної проблемі є використання мікросервісної архітектури [13]. Це архітектурний підхід, який структурує програму як набір невеликих, слабко пов’язаних служб. На відміну від традиційної монолітної архітектури, яка будує програму як єдину неподільну одиницю, мікросервіси поділяють програму на її основні функції. Потім кожна з цих функцій розробляється та розгортається як незалежні служби, які працюють у власних процесах і спілкуються одна з одною, як правило, через:

– HTTP/REST через JSON або інші формати даних [14];

– через спеціальні брокери повідомлень, такі як Apache Kafka чи RabbitMQ [15];

– двійкові буфери протоколи, такі як gRPC [16].

Основна перевага мікросервісної архітектури полягає в її модульності. Кожен сервіс розділений за бізнес-можливостями, і його можна створювати з використанням різних технологій, в тому числі мов програмування, що дає змогу розподіляти відповідальність на різні команди. Ця модульність зменшує ризики відмови всієї системи, так як компоненти це незалежні програми, якщо один компонент стикається з проблемою, інші продовжують працювати. Також модульність надає зручність в інтеграції нових модулів (мікросервісів).

Однак, по при всі переваги, дизайн з використання мікросервісної архітектури, може сильно ускладнювати систему. В першу чергу через реалізацію зв’язку між сервісами, реалізацію розподілених транзакцій та їх тестування, узгодженість даних системи та інше. Також, не менших проблем може спричинити розгортання та подальше обслуговування подібних систем. На щастя, дані проблеми можуть покривати такі технології, як Docker та Kubernetes [17]. Вони полегшують інтеграція, тестування системи та забезпечують автоматичне розгортання за рахунок контейнеризації — технології що забезпечує ідеальне обчислювальне середовище для розгортання програм.

### 3.1.2 Опис хмарної версії інфраструктури системи

Скоріш за все, для реалізації подібної системи, з такою архітектурою, найбільш вигідним рішенням буде скористатися послугами якогось хмарного провайдера, наприклад як: Microsoft Azure [18], Google Cloud Platform (GCP) [19] чи Amazon Web Services (AWS) [20]. Тим паче, технології хмарних обчислень за останні роки сильно розвинулися, та з кожним роком стаються все більш популярнішими та доступнішими.

Одна зі значних переваг використання постачальників хмарних послуг полягає в тому, що вони пропонують безліч готових архітектурних рішень, які дозволяють швидко та ефективно розгортати програми. Ці рішення абстрагують налаштування складної інфраструктури та забезпечують інтегровані спрощені інтерфейси для роботи.

### 3.1.2.1 Вибір провайдера хмарних технологій

Було проведено порівняння між трьома найпопулярнішими на даний момент постачальниками хмарних технологій, а саме: Amazon Web Services (AWS), Google Cloud Platform (GCP) та Microsoft Azure.

Порівнювалися сервіси, що надають одне й те саме рішення від різних провайдерів (таблиця 3.1). Також були розглянуть тільки ті сервери, що потенційно можуть бути використані для реалізації інфраструктури розроблюваної системи, а саме:

– API Gateway, для реалізації єдиної точки входу до системи;

– Virtual Servers, для запуску вебсервера;

– Object Storage, для зберігання інформації про товари;

– Relational Database Service, для зберігання даних системи;

– Virtual Private Cloud, для ізоляції та забезпечення безпеки системи в Інтернет середовищі.

Таблиця 3.1 – Порівняння схожих сервісів між різними провайдерами хмарних технологій

| Рішення | Amazon Web Services (AWS) | Google Cloud Platform (GCP) | Microsoft Azure | Порівняння |
|---|---|---|---|---|
| API Gateway |  Amazon API Gateway |  Cloud Endpoints |  Azure API Management | **Функціонал**: Кожний з провайдерів надає базовий функціонал API Gateway, однак і кожний з сервісів має унікальні сильні сторони. Amazon API Gateway пропонує окрім чудової інтеграції зі своїми внутрішніми сервісами, вчасності AWS Lambda, що є дуже перспективною, та може бути використаної в реалізації розроблюваної системи, також надає інтеграцію з RESTful та WebSocket сервісами. Однак слід зазначити, що деякі користувачі вважають його повідомлення про помилки неінформативними, а налаштування його автентифікації може бути складним. Однією з переваг GCP є те, що він підтримую протоколи gRPC для високопродуктивних API. Однак навряд |

| | | | | |
|---|---|---|---|---|
| | | | | чи розроблювана система на даному етапі потребує подібну технологію, хоча це не виключено в майбутньому, при переході до більш глобального рішення. Також Google Cloud Endpoints відомий здатністю автоматично генерувати клієнтські бібліотеки API, заощаджуючи розробникам значну кількість часу.<br>В свою чергу, Azure API Management, пропонує вражаючі функції, такі як автоматичне створення пакета SDK, генерація ключів API, комплексна аналітика та надає високий рівень настроюваності, що дозволяє точніше контролювати ваші політики API. Однак з іншого боку його відносно складно встановити та налаштувати. |
| Virtual Servers |  <br>EC2 (Elastic Compute Cloud) |  <br>Compute Engine |  <br>Azure Virtual Machines | **Ціноутворення**: AWS EC2 має складну структуру ціноутворення та може стати дорогим рішенням без належного управління та розуміння моделі ціноутворення. Тоді як віртуальні машини Azure схиляються до простіших моделей. Найкраще в цьому плані у GCP Compute Engine , де постійне використання екземплярів віртуальних машин може призвести до автоматичних знижок.<br>**Типи екземплярів**: EC2 відомий широким вибором, тоді як GCP зосереджується на високопродуктивних варіантах. В свою чергу, віртуальні машини Azure, можуть похвалитися гарною інтеграцією в середовище Windows.<br>**Надійність та масштабованість**: В AWS досить часто трапляється, що віртуальні машини стають недоступними без попередження [21]. Частіше за все, це відбувається під час міграції та менеджменту машин на стороні провайдера (під час технічного обслуговування). Схожа ситуація і з Azure, це тому, що ці платформи були розроблені з пріоритетом на гнучкість та еластичність, швидкість розгортання та масштабованість, а не на надійність запущених машини. В свою чергу Google пропонує функцію живої міграції, що дозволяє переміщувати запущені екземпляри на інші хости під час технічного обслуговування. Це унікальна функція, яку не пропонують AWS чи Azure, та яка надає перевагу GCP в аспекті надійності. |

| | | | | |
|---|---|---|---|---|
| Object Storage |  S3 (Simple Storage Service) |  Cloud Storage |  Azure Blob Storage | **Ціноутворення**: Google Cloud Storage цінується за постійні ціни та знижки на тривале використання. Де в AWS S3 в цьому можуть бути проблеми, в часності в ціноутворенні на частоту та кількість запитів до сховища.<br>**Різноманітні типи**: Всі три постачальника хмарних технологій пропонують широкий вибір різних типів та класів сховищ на основі комбінації різних характеристик. Вчасності, це частота звертання до даних. Хоча слід зазначити, що AWS в даній категорії де що випередив свої конкурентів.<br>**Довговічність**: Хоча всі три забезпечують високу довговічність, S3 за замовчуванням копіює дані лише в межах регіону. GCP Cloud Storage і Azure Blob Storage можуть реплікувати дані між регіонами для додаткової надлишковості. Якщо потрібна дуже висока надійність із мультирегіональною реплікацією, GCP або Azure можуть бути кращими. Хоча слід зазначити, що Amazon S3 відомий своїм беззаперечним показником довговічності а саме 99,999999999% (одинадцять дев'яток) та підтримує шифрування даних у стані спокою та під час передачі.<br>**Доступність та масштабованість**: кожній з провайдерів може похвалитися практично нескінченною масштабованістю та дуже високим рівнем доступності. Хоча слід зазначити, що Amazon S3 тут також відомий своєю масштабованістю, високою доступністю та низькою затримкою. |
| Relational Database Service |  RDS (Relational Database Service) |  Cloud SQL |  Azure SQL Database | **Підтримувані бази даних**:<br>RDS пропонує найширший вибір, включаючи MySQL, PostgreSQL, MariaDB, Oracle, Aurora (сумісні з MySQL і PostgreSQL), SQL Server і Amazon Redshift.<br>Cloud SQL підтримує MySQL, PostgreSQL і SQL Server.<br>База даних SQL Azure підтримує різні випуски Microsoft SQL Server, а також MySQL, PostgreSQL і MariaDB.<br>**Варіанти розгортання**:<br>RDS пропонує два варіанти розгортання: ініціалізований (сервери, якими керує користувач) і без сервера (автоматичне масштабування).<br>Cloud SQL і Azure SQL Database в основному зосереджені на безсерверній |

| | | | | |
|---|---|---|---|---|
| | | | | моделі з автоматичним масштабуванням. Але всеж таки, якщо вам потрыбне гнучке розгортання та більший контроль над конфігурацією сервера, слід звернути увагу на RDS.<br>**Pricing**: Ціни RDS залежать від механізму бази даних, типу екземпляра, використовуваного сховища та операцій вводу/виводу.<br>Cloud SQL і база даних SQL Azure зазвичай використовують модель оплати за використання на основі використаних обчислювальних ресурсів і сховища. |
| Virtual Private Cloud |  <br>Amazon VPC |  <br>VPC Network |  <br>Azure Virtual Network | **Функціонал**: Кожний з провайдерів надає базовий функціонал Virtual Private Cloud, однак GCP де що виділяється, пропонуючи можливість охоплювати всі регіони без додаткових витрат, глобального охоплення (global span). Це унікальна функція, яку не надають AWS або Azure. |

Підсумовую, можна сказати, що на даний момент перевагою Google Cloud Platform (GCP) є чудово структуроване ціноутворення та в цілому дещо привабливіші, зокрема для Object Storage, що скоріш за все буде використано в розроблюваній системі. В свою чергу AWS виділяється своєю гнучкістю та еластичністю, на додаток зі своєю швидкістю розгортання та масштабованістю, що насправді дуже важливо для розроблюваної системи. Microsoft Azure де що суміжне між GCP та AWS, однак слід зазначити, що він виділяється високим рівнем налаштування для своїх сервісів, однак на жаль це не завжди потрібно.

Як можна було здогадатися заздалегідь, як такого «кращого» варіанту тут немає. Вибір повністю залежить від конкретних потреб проекту, без врахування яких неможливо буде зробити правильний вибір. Хоча, і «неправильним», цей вибір також буде важко назвати, так як дані провайдери вже дуже дано на ринку, та протягом усього цього часу завжди займали передові позиції в даному напрямку, що свідчить про їх самостійність, та здатність покрити майже будь які рішення.

Тому вибір провайдера, в даному випадку не дуже принциповий, та навряд чи сильно вплине на якість та ціну кінцевого продукту. Однак, як і було вже зазначено, все ж таки вони мають певні характеристики, які їх виділяють між собою, і які можуть вплинути на результат. Тому взявши до уваги це, та

проаналізувавши вимоги до розроблюваної системи, було обрано Amazon Web Services.

#### 3.1.2.2 Детальний опис обраних сервісів та проектування інфраструктури

Для подальшого аналізу, було більш детально розглянуто основні характеристики та можливі способи застосування обраних сервісів (таблиця 3.2).

Таблиця 3.2 – Детальний опис обраних AWS сервісів

| Сервіс | Детальний опис |
|---|---|
| API Gateway [22] | Одним з найрозповсюдженіших архітектурних патернів в розподілених системах є API Gateway (API Шлюз), і майже всі хмарні провайдери надають свою реалізацію.<br>Даний патерн описує певний програмній засіб, який діє як посередник між запитами клієнтів, та внутрішніми компонентами розподіленої системи. API Gateway — це надійна та безпечна точка входу до вашої системи, яка приймає абсолютно всі запити від клієнтів направляє їх до правильних служб, та в подальшому збирає результати для відповіді клієнту.<br>Це потужний патерн, який може включати в себе багатий функціонал та купу інших архітектурних рішень які можуть значно спрощувати завдання розгортання, захисту, керування та масштабування API, дозволяючи розробникам зосередитися на фактичній логіці системи та її компонентів. Ось декілька основних з функцій API Gateway:<br>– маршрутизація (routing). Мабуть одна з найголовніших функцій API Gateway. Даний функціонал полягає в обробці вхідних клієнтських запитів та їх маршрутизації (направлення) до правильних серверних служб на основі вказаних правил маршрутизації, таких як: URL-шлях, HTTP метод та інші дані. Наприклад в залежності від того який клієнт робить цей запит, чи це каса самообслуговування, чи це звичайна каса, чи це запит від якогось працівника підприємства, там адміністратор чи менеджер по продуктам або складу, API Gateway аналізує цей запит, та вирішує на яку із внутрішніх служб відправляти цей запит на обробку;<br>– кешування відповідей. API Шлюзи можуть кешувати відповіді від серверних служб, щоб покращити час відповіді та зменшити навантаження на серверні служби;<br>– моніторинг і логування (monitoring, logging). API Шлюзи можуть логувати запити та відповіді, надаючи цінні дані про те, як клієнти використовують API;<br>– балансування навантаження (load balancing). Архітектурний патерн який описує програмний засіб, що сприяє рівномірному розподілу вхідного мережевого трафіку між декількома серверними службами, гарантуючи, що сервери не перевантажуються, а продуктивність додатків залишається оптимальною;<br>– обмеження швидкості та регулювання (rate limiting, throttling). Відстежуючи кількість запитів, які клієнт робить протягом визначеного періоду часу, API Шлюзи можуть допомогти запобігти надмірному використанню або зловживанню API;<br>– автентифікація та авторизація. API шлюзи можна налаштувати для автентифікації та авторизації кожного клієнтського запиту до того, як він досягне серверних служб, додаючи додатковий рівень безпеки. |

| | |
|---|---|
| EC2 [23] | Для запуску віртуального серверу, в AWS використовується Amazon EC2 (Elastic Compute Cloud).<br>Даний сервіс дозволяє швидко піднімати віддалені машини для різних хмарних обчислювань, в тому числі і можливістю запуску власних програм, вчасності вебсерверу. Це універсальне рішення, що має неймовірні можливості в налаштування. Відповідно до поставлених задач, можна конфігурувати такі фактори, як: операційна система (Windows або Linux), пам'ять, ЦП, система зберігання даних, мережеві конфігурації, та багато чого іншого. Також Amazon EC2 надає широкий вибір різних типів вже готових екземплярів, оптимізованих для різних випадків використання. Вони згруповані за категоріями на основі таких характеристик як: пам'яті, ЦП, об'єму пам'яті та об'єму мережі. Одним з найголовніших переваг використання Amazon EC2 є його миттєва масштабованість. EC2 дозволяє динамічно збільшувати або зменшувати потужність хмарних обчислювань за лічені хвилини, задовольняючи потреби вашої програми. Можна змінювати я характеристики певної машини, наприклад збільшуючи обсяг пам'яті (вертикальне масштабування), так і вводити в експлуатацію один або навіть тисячі екземплярів сервера одночасно (горизонтальне масштабування). |
| S3 [24] | Постачальники хмарних технологій пропонують різноманітні механізми зберігання даних, адаптовані до різних типів і розмірів даних і різноманітних випадків використання. Їх можна загалом розділити на сховище об'єктів, сховище блоків і сховище файлів. Кожен із цих типів сховища використовується для різних програм на основі таких факторів, як розмір даних, потреба в доступі в реальному часі, вартість тощо.<br>Наприклад, Amazon S3 (Simple Storage Service) — це служба зберігання об'єктів. Він призначений для зберігання будь-яких типів об'єктів або файлів і є хорошим вибором для зберігання зображень, файлів резервних копій, архівів даних або журналів, а також розповсюдження вмісту. Він може похвалитися практично необмеженою масштабованістю та довговічністю. S3 може інтегруватися з різними службами AWS і надає такі функції, як керування життєвим циклом, версії, шифрування та безпека. |
| RDS [25] | Також, популярним AWS сервісом пов'язаним зі зберіганням даних — є Amazon RDS (Relational Database Service). Це сервіс, що поставляє інтеграцію з реляційними базами даних у середовищі AWS. Він підтримує шість популярних баз даних: Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle і SQL Server. RDS поставляє все що необхідно для виконання звичайних завдань пов'язаних з базами даних, такі як: надання віртуального обчислюваного середовища для запуску СУБД, різні налаштування, програмне оновлення, масштабування, автоматичне резервне копіювання, знімки, виявлення збоїв, відновлення (ремонт) та багато чого іншого.<br>В подібній архітектурі централізована база даних це рідкісне явище. Зазвичай кожна з внутрішніх служб системи має свою базу даних, і якщо потрібні якісь дані з іншої служби, то їх отримують безпосередньо через цю службу а не звертаються напряму до бази даних. Це робиться, в першу чергу, для підтримки незалежності компонентів, що дуже важливо в мікросервісній архітектурі. Це забезпечує ту гнучкість, заради якої ми використовуємо цю архітектуру, та спрощує тестування окремих сервісів. |
| DynamoDB [26] | Ще одним популярним механізмом зберігання даних в AWS, є DynamoDB. Це NoSQL база даних для створення високопродуктивних масштабованих програм, які потребують низької затримки доступу до даних. На відміну від реляційних баз даних, DynamoDB зберігає дані в парах ключ-значення, що робить його ефективним для отримання конкретних елементів на основі унікальних ключів, однак і не виключаються потужні агреговані запити.<br>Також DynamoDB пропонує привабливе ціноутворення. Можливе пакетне придбання послуг даного сервісу, до трьох років за раз, по дуже привабливій ціні.<br>Основним недоліком DynamoDB є можливі проблеми у випадках коли необхідна чітка схема зі складними зв'язками між даними або коли потрібно суворо |

| | |
|---|---|
| | відповідає вимогам ACID (атомність, узгодженість, ізоляція, довговічність). В такому випадку, можливо кращим рішенням буде використати RDS.<br>Але у випадках коли потрібна висока пропускна здатність з швидким доступом до даних, DynamoDB можливо ідеальне рішення. |
| VPC [27] | Також важливою складовою будь якої подібної хмарної інфраструктури є безпека. На щастя більшість хмарних провайдерів пропонують досить багато хороших механізмів для реалізації цієї безпеки. AWS не э виключенням, та надає послуги свого сервісу Amazon VPC (Virtual Private Cloud).<br>Це сервіс, який дозволяє запускати AWS ресурси у логічно ізольованій віртуальній мережі, яку можна визначати та налаштовувати самостійно. Він забезпечує безпечне середовище, де ви можете контролювати мережеві аспекти, визначаючи власний діапазон IP-адрес, створюючи підмережі та налаштовуючи таблиці маршрутів і мережеві шлюзи.<br>Ключовою проблемою, яку вирішує Amazon VPC, є необхідність безпечної, налаштовуємої та масштабованої мережевої інфраструктури в хмарі. Це ідеальний механізм, що дозволяє поєднати переваги хмарних обчислень із безпекою та контролем.<br>Отже, одна з головних функцій даного сервісу — це безпека. Ви можете створити групу безпеки (security group), яка фактично діє як віртуальний брандмауер, щоб контролювати вхідний і вихідний трафік на рівні екземпляра, такого як EC2. Інший спосіб для надання можливісті контролювати мережевий доступ як до своїх екземплярів, так і до інших підмереж (subnets), це використання списків контролю доступу до мережі (NACL (Network Access Control Lists)). |

В результаті була створена хмарна версія компонентної діаграми інфраструктури системи в AWS середовищі (рисунок 3.2). Без прозорості виділено те, що стосується виключно сервісу обробки покупок на касах самообслуговування. Компоненти з прозорістю, це те що може бути інтегровано в систему в майбутньому.

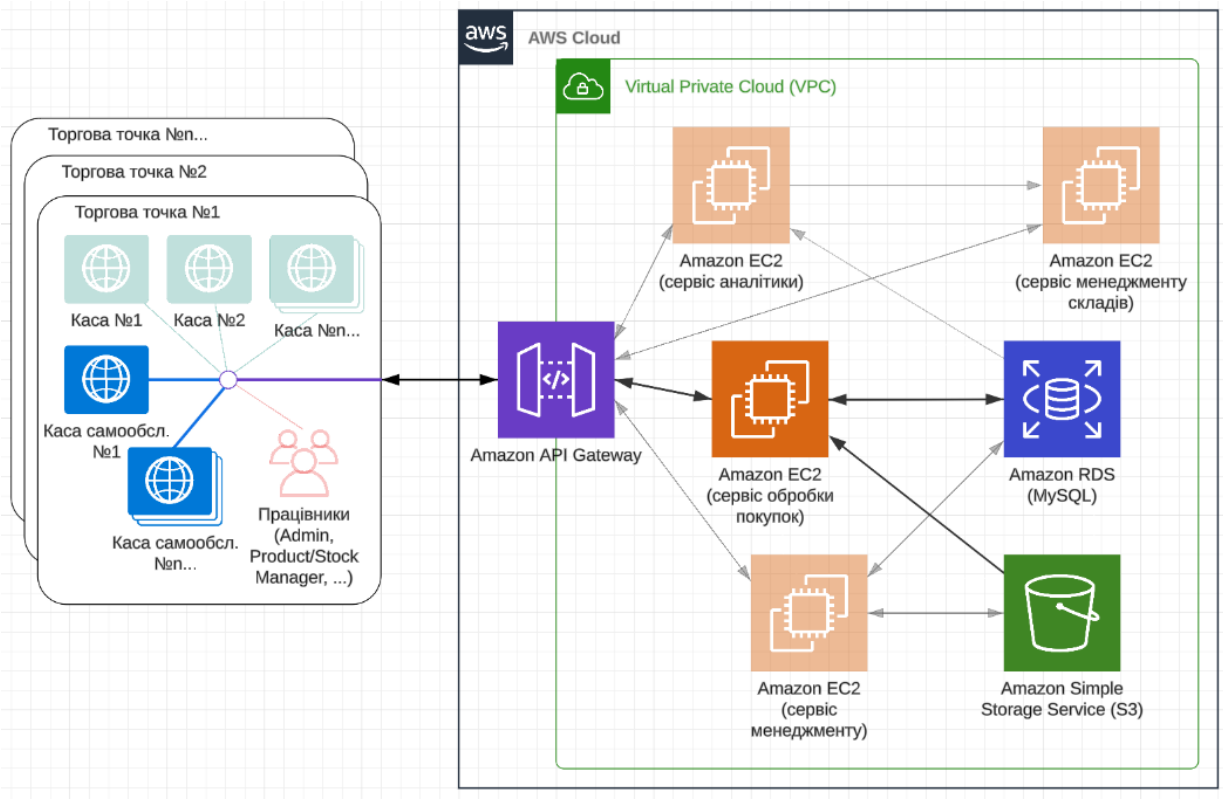



Рисунок 3.2 – Компонентна діаграма системи (хмарна версія (AWS))

### 3.1.2.3 Аналіз та оптимізація інфраструктури

Звісно використання подібних сервісів не безкоштовне, і можливо навіть на початкових етапах використання власних серверів є більш економічно вигідним. Однак з розвитком системи, додаванням нових сервісів, збільшення кількості самих торгових точок, та в загалом потужності системи, це все може призвести до значних витрат, тим паче на обслуговування цієї інфраструктури. Це може бути обумовлено такими факторами, як: регулярне технічне обслуговування, придбання ліцензій, усунення несправностей, заміна застарілого та закупка нового обладнання. Все це, в свою чергу, потребує наймання кваліфікованого персоналу, вчасності DevOps інженерів. Робота таких фахівців за часту відносно високооплачувана. Також, як відомо, даний підхід обмежений в масштабуванні. Зазвичай це займає багато часу, та може призводити до непередбачуваних витрат. В загалом використання власних серверів може бути економічно вигіднішим, однак це за ідеальних умов, насправді ж, через велику кількість факторів, даний підхід має багато різних ризиків.

Натомість, використання хмарних технологій, де рівень відповідальності дещо зсуваємо в сторону їх постачальника, робить дане рішення більш простішим, надійнішим та зручнішим. Тому використання хмарних технологій, в потенціалі є більш привабливим, та економічно вигідним рішенням.

Також слід зазначити, що зазвичай магазини не працюють вночі, тому для заощадження коштів, бажано відключати сервери на ніч. Якщо у випадку з власним сервером, тут можуть бути додаткові налаштування та можливі проблеми з цим. У випадку з використанням хмарних технологій, все знову ж таки набагато легше та зручніше. Плюс багато постачальників хмарних технологій, в тому числі і AWS, дотримуються політики «pay for what you use», що означає, що ти платиш лише за те що ти використовуєш. І наприклад коли ви вимикаєте ваш сервер, за те що він начеб то простоює, ви нічого не платите.

На додаток, різні хмарні провайдери поставляють дуже багато механізмів, за рахунок яких можна оптимізувати інфраструктуру в цілому, та мінімізувати витрати. Наприклад, сервіс аналізу даних чи сервіс менеджменту складів, не обов’язково повинні бути повноцінними працюючий сервісами (AWS EC2). Так як їхнє використання може бути обмеженим, та наприклад, якісь підрахунки можуть відбуватися раз на день, в такому випадку доцільніше було б використати якийсь більш підходящий хмарний веб сервіс. Наприклад у випадку AWS, це може бути CloudWatch чи навіть сукупність якихось AWS Lambda функцій.

Тому важливо розуміти, що хоча ці хмарні технології і можуть забезпечити високу гнучкість, масштабованість, економію коштів та інші переваги, вони також вимагають певного рівня знань та розуміння для ефективного керування.

Було проведено порівняння між використання різних AWS сервісів для реалізації механізму зберігання інформації про товари та надання цієї інформації користувачам у разі необхідності. Було розглянуто наступні рішення:

– лише S3;

– RDS;

– та DynamoDB з S3.

Для проведення оцінки витрат, важливо розуміти такі показники, як:

– кількість унікальних типів товарів (який потрібен об’єм пам’яті для зберігання такої кількості товарів);

– кількість покупок на місяць (для оцінки кількості запитів до сервісу на місяць).

Було представлено наступні умови:

– умовне місто з населенням 1 мільйон осіб;

– мережа продуктових магазинів, з кількістю 10 торгових точок;

– 15000 різних типів товарів.

Було підраховано, що в середньому для зберігання інформації про один товар, потрібно приблизно 5 КБ даних (200 Б для зберігання текстової інформації про товар, та інше на зображення продукту). З цього виходить, що для зберігання 15000 товарів, потрібно 75 МБ пам’яті (0.075 ГБ).

Наступним кроком було підрахування кількості запитів до сервісу за місяць. Припустимо, що щодня одну торгову точку відвідує 1000 осіб. Виходить що це приблизно 1000 окремих покупок за добу. Однак на даний момент нас цікавить тільки каси самообслуговування. Припустимо, що це 50% від всіх покупок. Навіть якщо звичайних кас буде більше, це не значить що кількість закритих ними покупок буде більше. Слід враховувати що пропускна здатність кас самообслуговування може бути більшою, що значить, що за одиницю часу буде закрито більше покупок ніж у випадку зі звичайними касами. Однак також слід враховувати що і подібні покупки зазвичай менші за об’ємом. Отже 500 покупок за добу. Враховуючи що в умові 10 торгових точок, та в середньому в місяці 30 днів, отримуємо приблизно 150000 покупок на місяць.

Однак, скільки одна покупка потребує запитів до сервісу, та який об’єм даних має повернути сервіс за одну покупку? Припустимо, що середня кількість товарів в покупці на касі самообслуговування складає 7. Це означає що 7 запитів до сервісу. При успішному скануванні товару, повертаємий об’єм пам’яті складає 5 КВ, за покупку виходить 35 КВ. Однак слід враховувати, що іноді покупець може шукати товар за параметрами, що може призводити до повертання додаткових 200 КВ (припустимо, що при вводі покупцем не зовсім конкретних параметрів, максимально відображаємий об’єм товарів у вікні пошуку зіставляє 20 штук). Припустимо що пошук відбувається один раз на 100 покупок. В результаті отримуємо, що 150000 * 7 + 150000 / 100 = 1051500 запитів на місяць, та 150000 * 35 + 150000 / 100 * 200 = 5550000 КВ (5.55 ГБ).

Також треба підрахувати кількість відсканованих даних. Це могло б значити об’єм даних на кількість запитів, однак слід враховувати що зображення,

що займають більшу кількість пам’яті, зберігаються окремо, і сканується лише умовно якийсь JSON файл з інформацією о товарах. Тому це приблизно 0.2 КБ * 15000 = 3000 КБ (0.003 ГБ), тобто 1051500 * 0.003  = 3154.5 ГБ.

Операції додавання нових товарів до системи нас не цікавлять, так як ми розглядаємо лише сервіс обробки процесу покупки, тому можемо нічого не встановлювати.

Тип S3 було обрано як «Standart». Це доволі гнучкий варіант який здатен витримувати високе навантаження та в свою чергу є порівняно недорогим.

Для оцінки вартості використаних сервісів було використано один з сервісів AWS, а саме AWS Pricing Calculator [28]. Він відображає реальну ієрархію ціноутворення платформи в залежності від введених параметрів для обраних сервісів.

В результаті очікувана вартість використання сервісу S3 на місяць складає 7.56 USD (приблизно 300 гривень за поточним курсом. За рік це виходить 7.56 * 12 = 90,72 USD (приблизно 3630 гривень за поточним курсом). На рисунку 3.3 наведено розрахунок вартості використання сервісу S3 Standart:

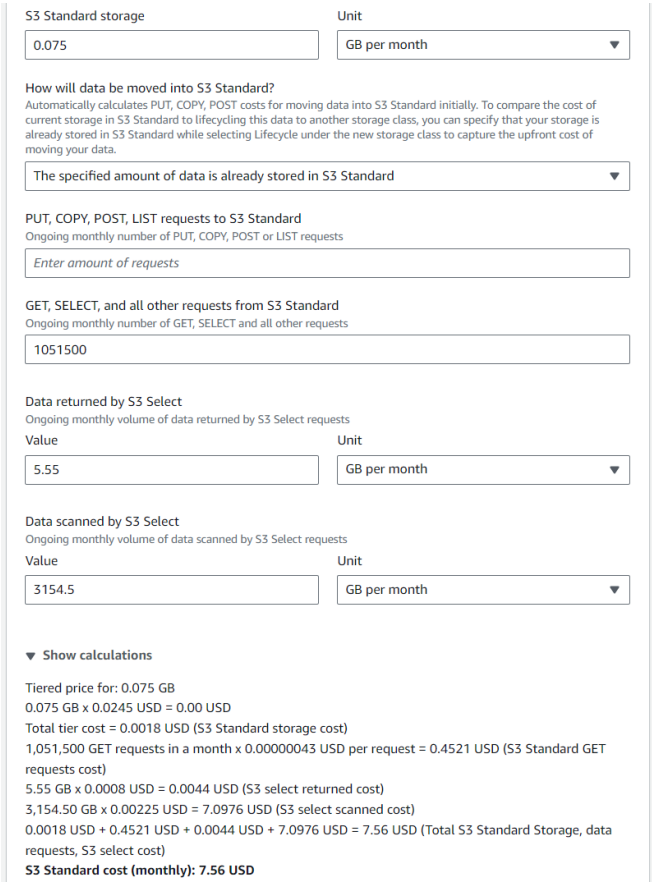



Рисунок 3.3 – Розрахунок вартості використання S3 за місяць

Також слід враховувати, що вартість сервісу варіюється в залежності від обраного регіону. Різниця може досягати до 10% від загальної суми. В розрахунках було обрано центральний європейський регіон, а саме: Europe (Frankfurt).

Також були проведені схожі підрахунки для RDS сервісу. Припустимо робочий день нашого магазину складає 14 годин на добу. Враховуючи, що умовна мережа магазинів має 10 торгових точок та в середньому 30 робочих днів магазину на місяць, отримаємо 14 * 10 * 30 = 4200 робочих годин на місяць. Враховуючи, що очікувана кількість запитів на місяць складає 1051500, отримаємо приблизно 1051500 / 4200 = 250 запитів на годину, або приблизно 4 запити на хвилину. Для подібного навантаження в принципі може вистачити і RDS сервісу з мінімальними налаштуваннями, однак слід враховувати, що це середній показник, та в пікових моментах дане значення може бути в десятки разів більше. Припустимо що в кожній з торгових точок у нас по 6 кас самообслуговування, виходячи з того, що в умові 10 торгових точок, пікове навантаження малоймовірно що може бути вищім за 60 запитів за секунду. У випадку з використанням S3, це не було проблемою, так як даний сервіс може витримувати і значно більші навантаження.

Насправді в загалом, це не проблема для AWS, дана платформа відома своєю гнучкістю та швидкістю розгортання. Можна прогнозувати пікові дні або навіть пікові години на день, та налаштувати, щоб автоматично піднімалася додаткова база даних, або ж просто використовувалася більш потужна БД. Але це потребує додаткових витрат, та може бути не просто налаштувати. Однак це не єдине рішення від AWS, далі буде розглянуто ще одне, що в даному випадку є ще більш гнучкішим та вигіднішим.

Для підрахунку була обрана одна з найпростіших віртуальна машина для запуску БД, а саме «db.t3.micro», що має 1 ГБ оперативна пам'ять. Кількість екземплярів даної машини було встановлено на 1. Нажаль кожна з машин, що включає базу даних, має мати як мінімум 20 ГБ вільної пам’яті для зберігання даних, що набагато більше ніж нам потрібно (75 МБ). Тип диску для цього був обрав відносно типовий, а саме: «General Purpose SSD (gp2)» (один з найдешевших варіантів). Також було обрано тип розгортання «Single-AZ», що означає що база даних не буде реплікуватися на інші зони доступності (Availability Zone). В загалом, якщо навіть не враховувати той фактор, що дуже

рідко і не на довго буває коли якась Availability Zone недоступна, для розроблюваного сервісу це не дуже принципово, так як у разі такого, просто можуть бути задіяні звичайні каси.

Для того, щоб база даних працювала не постійно, а могла автоматично вимикатися, наприклад вночі, коли магазини не працюють. Слід встановити моделі ціноутворення як «OnDemand», що робить можливим платити за обчислювальні ресурси, які використовує ваш екземпляр бази даних за годину. В результаті було встановлено яку кількість годин на день має бути доступна БД (14 годин).

В результаті очікувана вартість використання сервісу RDS на місяць складає 14.84 USD (приблизно 600 гривень за поточним курсом). За рік це виходить 14.84 * 12 = 178,08 USD (приблизно 7123 гривні за поточним курсом). На рисунку 3.4 наведено розрахунок вартості використання сервісу RDS:

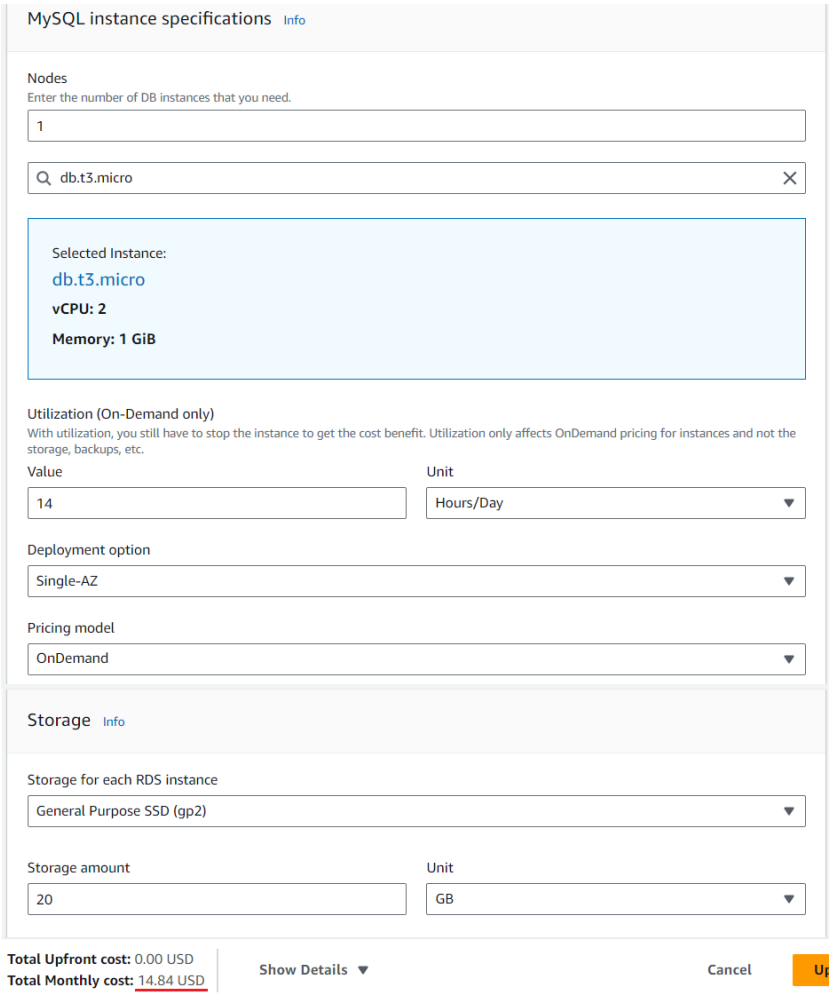



Рисунок 3.4 – Розрахунок вартості використання RDS за місяць

Також було розглянуто третє рішення реалізації механізму зберігання даних в інфраструктурі AWS на базі DynamoDB для зберігання основної інформації про товари та додатково S3 для зберігання зображень товарів.

Як вже було встановлено до цього, середній розмір про товар (текстові дані) складає 200 байт, що і було встановлено в калькулятор. Також було встановлено загальний об’єм пам’яті для зберігання всіх товарі. Так як зображення ми зберігаємо в S3, для зберігання текстової інформації про товар, потрібно: 15000 * 0.2 = 3000 Б (3 КБ). Нажаль при використанні DynamoDB мінімальний об’єм пам’яті ви платите складає 1 ГБ, в той час як при поставлених умов потрібно значно менше (0.000003 ГБ). Це схожа ситуація, як і у випадку з RDS, де було потрібно 75 МБ, а використано 20 ГБ , але виглядає ще більш критично. Насправді, це має незначний характер, і при таких маленьких об’ємах пам’яті різниця в їх вартості незначна. Модель ціноутворення для подібних сервісів в AWS зазвичай така: чим більший об’єм пам’яті ти використовуєш тим більше ти платиш. Тому і набагато вигіднішим в подібних сервісах при великих об’ємах даних є горизонтальне масштабування, а не вертикальне.

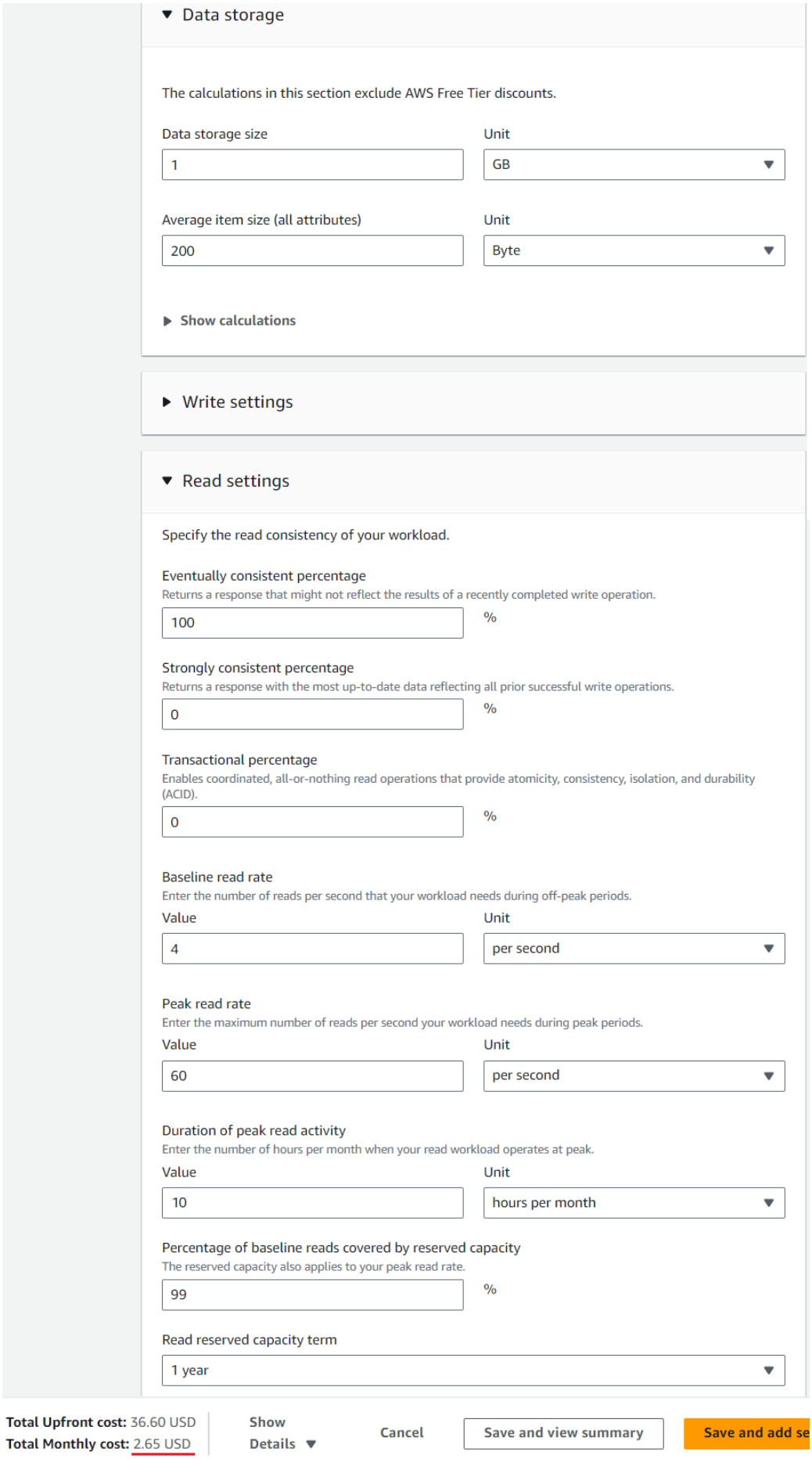
Знову ж таки, налаштування запису нас не цікавлять, так як розглянутий сервіс тільки читає дані.

Так як дані в базі не змінюються, їх консистентність не може порушитися. Тому ми і встановлюємо відсоток звичайних операцій на 100, та на 0 сильно консистентні запити та транзакції.

Як вже було встановлено до цього, середня кількість запитів на хвилину складає 4. А можливий піковий показник не може перевищити 60 запитів на секунду. Тривалість пікового навантаження була встановлено 10 годин на місяць.

Відсоток запитів, що відповідають вище встановленим нормам було поставлено як 99%, так як на кожні 100 запитів відбувається запит с пошуком, який має дещо інший характер. Так як він зустрічається доволі рідко, їм можна знехтувати, тому калькулятор не буде його враховувати. В будь якому випадку результат є приблизний та має певну похибку.

В результаті очікувана вартість використання сервісу DynamoDB на місяць та на рік, що складає 2.25 USD (приблизно 18 гривень за поточним курсом) на місяць, та 36.60 USD (приблизно 1465 гривень за поточним курсом) на рік. На рисунку 3.5 наведено розрахунок вартості використання сервісу DynamoDB:



▼ Data storage

The calculations in this section exclude AWS Free Tier discounts.

Data storage size: 1 — Unit: GB

Average item size (all attributes): 200 — Unit: Byte

▶ Show calculations

▶ Write settings

▼ Read settings

Specify the read consistency of your workload.

Eventually consistent percentage
Returns a response that might not reflect the results of a recently completed write operation.
100 %

Strongly consistent percentage
Returns a response with the most up-to-date data reflecting all prior successful write operations.
0 %

Transactional percentage
Enables coordinated, all-or-nothing read operations that provide atomicity, consistency, isolation, and durability (ACID).
0 %

Baseline read rate
Enter the number of reads per second that your workload needs during off-peak periods.
Value: 4 — Unit: per second

Peak read rate
Enter the maximum number of reads per second your workload needs during peak periods.
Value: 60 — Unit: per second

Duration of peak read activity
Enter the number of hours per month when your read workload operates at peak.
Value: 10 — Unit: hours per month

Percentage of baseline reads covered by reserved capacity
The reserved capacity also applies to your peak read rate.
99 %

Read reserved capacity term
1 year

**Total Upfront cost:** 36.60 USD
**Total Monthly cost:** 2.65 USD

Show Details ▼ | Cancel | Save and view summary | Save and add se

Рисунок 3.5 – Розрахунок вартості використання DynamoDB за місяць

DynamoDB відомий своєю гарною моделлю ціноутворення. Та дозволяє резервувати потужність читання на певний термін, отримуючи відповідні скидки в залежності від терміну. Максимальний можливий термін 3 роки, в прикладі було встановлено 1 рік.

Далі було підраховано скільки буде коштувати зберігати зображення товарів в S3. Кожне зображення в середньому займає 4.8 КБ, отже необхідний об’єм пам’яті для зберігання зображень для 15000 товарів складає: 15000 * 4.8 = 72000 КБ (0.072 ГБ). А об’єм пам’яті витягнутий зі сховища складає: 150000 * 4.8 * 7 + 150000 / 100 * 200 = 5340000 КВ (5.34 ГБ). Відсканований об’єм даних складає 0, тому що потрібні зображення миттєво знаходяться но їх унікальній назві, що не потребує сканування всіх даних.

В результаті очікувана вартість використання сервісу S3 на місяць для зберігання тільки зображень продуктів, складає 0.46 USD (приблизно 18 гривень за поточним курсом). На рисунку 3.6 наведено розрахунок вартості використання сервісу S3 Standart для зберігання тільки зображень продуктів:

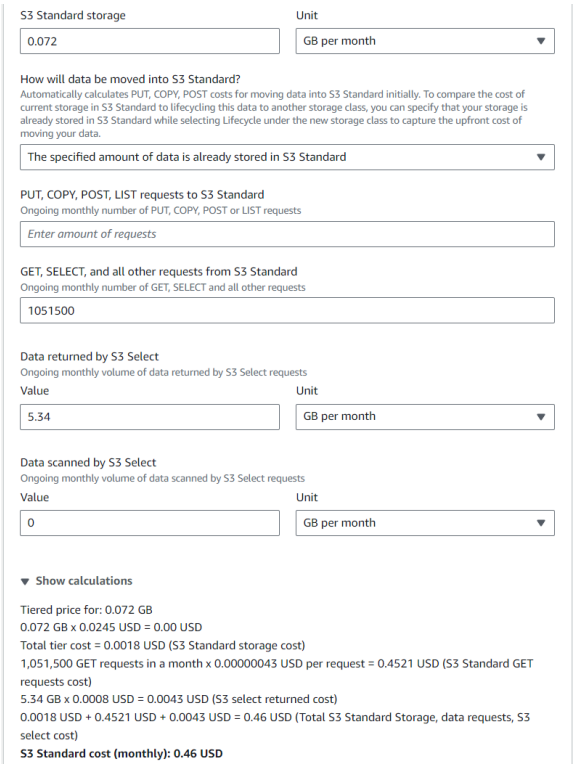



Рисунок 3.6 – Розрахунок вартості використання S3 за місяць для зберігання тільки зображень продуктів

В результаті, очікувана вартість використання сервісів DynamoDB та S3 на рік складає 73.92 USD (приблизно 2957 гривень за поточним курсом), що можна побачити на рисунку 3.7.

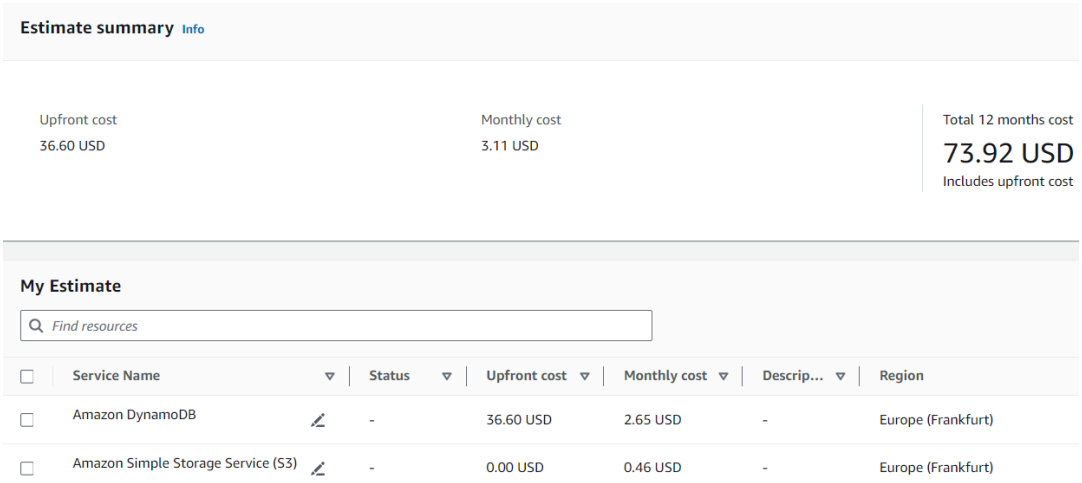



Рисунок 3.7 – Розрахунок вартості використання DynamoDB та S3 за рік

Результат проведеного аналізу порівняння рішень зберігання даних з використанням різних AWS сервісів можна бачити у таблиці 3.3.

Таблиця 3.3 – Порівняння рішень зберігання даних з використанням різних AWS сервісів

| Використані AWS сервіси для збереження даних | Вартість за рік | Недоліки | Переваги |
|---|---|---|---|
| S3 | 3630 грн | Повільний пошук (сканування) особливо при великих об’ємах даних.<br>Потребує додаткових досліджень у напрямку правильного зберігання для пришвидшення пошуку.<br>Чи зберігати все в одному файлі, наприклад у файлі JSON.<br>Чи розділяти, та для кожного товару використовувати окремий файл, та вже шукати по його назві. | Легко налаштувати та підтримувати.<br>Приваблива ціна.<br>Дуже гарна автоматична масштабованість. |

| | | | |
|---|---|---|---|
| RDS | 7123 грн | Механізми масштабування є, однак потребують додаткових витрат, та компетенції у налаштуванні.<br>Не повністю розкривається потенціал сервісу, де мінімальний об'єм зберігаємих даних за що платиться складає 20 ГБ. | Швидкий пошук та відповідь.<br>Потенційно може бути більш вигіднішим порівнюючи з іншими сервісами, особливо при значно більшій кількості запитів за секунду ніж була розглянута у прикладі. |
| DynamoDB + S3 | 2957 грн | Не повністю розкривається потенціал сервісів, вчасності DynamoDB, де мінімальний об'єм зберігаємих даних за що платиться складає 1 ГБ.<br>Використання двох різних сервісів потребує де що вищий рівень кваліфікації для налаштування, реалізації їх взаємозв'язку та подальшої підтримки. | Дуже гарна автоматична масштабованість.<br>Швидкий пошук та відповідь.<br>Дуже гарне ціноутворення, особливо у довгостроковій перспективі. |

За результатами проведеного аналізу порівняння рішень зберігання даних з використанням різних AWS сервісів, для розроблюваного сервісу перевагу було надано реалізації з використанням DynamoDB та S3 сервісів, так як виявлені недоліки незначні, а отримані переваги беззаперечні.

### 3.1.3 Вибір архітектури вебсерверу

У проектуванні та розробці будь-якої програмної системи вибір архітектури відіграє ключову роль у визначенні її ефективності, швидкості розробки, масштабованості та зручності обслуговування. Архітектура служить схемою для організації компонентів системи, визначає їх взаємодії та допомагає вирішити ключові проблеми, такі як продуктивність, безпека та масштабованість. Добре підібрана архітектура не тільки полегшує процес розробки, але й закладає основу для майбутніх вдосконалень та модифікацій.

Обираючи архітектуру, слід враховувати багато різних факторів. В основі, від чого слід відштовхуватися — це вимоги та цілі проекту. З ключових факторів, які слід враховувати, є функціональні вимоги, їх обсяг та складність.

Також важливим фактором є нефункціональні вимоги. Чи має система відповідати таким вимогам, як продуктивність, масштабованість, безпека,

зручність обслуговування та розширюваність. Наприклад, оцінюючи вимоги до масштабованості системи, слід враховувати очікуване зростання бази користувачів, обсягу даних можливих транзакцій. В залежності від результатів обрати архітектуру, яка може масштабуватися горизонтально або вертикально, підтримуючи певний рівень продуктивності застосунку.

Також слід оцінювати потребу в майбутніх вдосконаленнях, модифікаціях та інтеграціях, чи передбачаються вони в загалі, чи ні. Відштовхуючись від цього, обрати архітектуру, яка забезпечує гнучкість і розширюваність, дозволяючи системі розвиватися з часом без значних зусиль та відповідно витрат.

Обмеженням у виборі якихось архітектурних рішень може бути наявність певних нормативних правил. Це будь-які нормативні вимоги або галузеві стандарти, яким має відповідати система. В інформаційних системах, це зазвичай правила захисту даних або якісь стандарти безпеки.

Та мабуть найбільший вплив на вибір архітектури, є фактор обмеженості проекту. Це в першу чергу бюджет, часові рамки, доступність ресурсів, організаційна політика та інші. Слід обрати архітектуру, яка може бути реалізована в рамках заданих обмежень, одночасно відповідаючи цілям проекту.

У контексті касової системи самообслуговування, було виявлено та надано перевагу наступним чинникам:

– складність інформаційної системи. Враховуючи природу касової системи самообслуговування, як відносно простого додатку, є край важливим уникнути надмірного проектування, вибравши просту архітектуру, яка відповідає вимогам системи;

– термін розробки. Враховуючи часові обмеження, важливим є обрати архітектуру, яка сприяє швидкій розробці без зайвих складнощів. Яка буде зручною для команди розробників, та відповідати їх навичкам та стеку технологій;

– підтримка та розширюваність системи. Хоча пріоритетом є простота, обрана архітектура має знайти баланс, дозволяючи майбутні вдосконалення та модифікації без значних зусиль. Це також включає можливість інтеграції з різними сторонніми системами моніторингу та аналізу.

На базі проведеного аналізу предметної області та існуючих архітектурних моделей та рішень, для інформаційної системи касового самообслуговування,

було обрано відносно типову архітектуру для будь якого веб-застосунку, яка успішно підтвердила свою ефективність на практиці в багатьох проектах, а саме Model-View-Controller (MVC) [29] (рисунок 3.8).

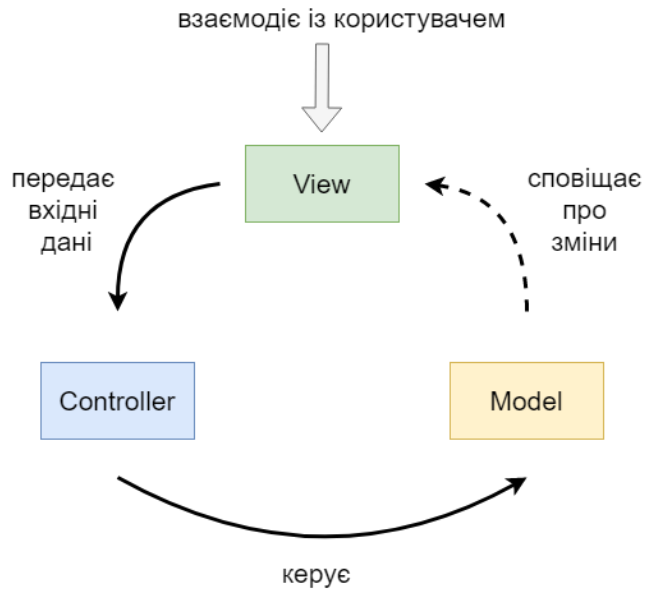



Рисунок 3.8 – Діаграма взаємодії між компонентами шаблону MVC

Даний архітектурний патерн представляє спосіб організації коду, який передбачає чіткий розподіл завдань, розділяючи програму на три взаємопов’язані компоненти, а саме:

– модель (model). Представляє дані та бізнес-логіку програми. Охоплює такі сутності, як продукти, користувачі, транзакції тощо. Інкапсулює основну функціональність системи, сприяючи повторному використанню та зручності обслуговування. Даний компонент зазвичай безпосередньо пов'язаний з якимось механізмом зберігання даних, наприклад файлова система чи стороння база даних;

– вид (view). Обробляє презентаційний рівень програми, її зовнішній вигляд, включаючи інтерфейс користувача. Використовує HTML, CSS і JavaScript для відтворення динамічного вмісту для користувачів. Дозволяє налаштовувати користувацькі інтерфейси, забезпечуючи безперебійну роботу користувача. Зазвичай це тонкого клієнта («thin client»), активних серверних сторінок вебсайту, що містять інтерфейс доступу до бази даних і об’єкти управління для відображення цих даних, реалізований на стороні веб-браузера;

– контролер (controller). Діє як посередник між моделлю та видом. Контролює роботу програми, обробляє запити користувачів, отримує дані з моделі та відповідно оновлює подання. Полегшує потік інформації всередині системи, організовуючи взаємодію між компонентами. Зазвичай реалізований на стороні серверу та може складатися з декількох частин, які можуть включати в себе реалізацію різних механізмів розподілу чи набори відповідних сценаріїв.

Головною перевагою даного архітектурний підходу є те, що він дуже простий у реалізації, тим паче майже будь який сучасний веб фреймворк реалізує даний архітектурний патерн, що ще більше полегшує його використання. Окрім того, даний підхід має низку корисних характеристик, а саме — це масштабованість, доступність та деякі характеристики продуктивності. Також даний підхід сприяє більш зручному тестуванню, так як дозволяє як окремо протестувати кожен з модулів (рівнів) так і взаємозв'язок між ними.

## 3.2 Опис обраного стеку технологій

### 3.2.1 Вибір веб-фреймворку та технології для реалізації бізнес-логіки

Оскільки розроблювана інформаційна система є веб-застосунком, доцільно було б використати якийсь веб-фреймворк. Вони значно спрощують процес розробки будь якого веб-додатку, надаючи готові рішення, які допомагають створювати ефективні веб-програми, які можна підтримувати та масштабувати. Однак велика кількість популярних фреймворків сильно ускладнює вибір найбільш підходящого. На вибір веб-фреймворку можуть впливати такі фактори, як мовні вподобання та переваги, підтримка спільноти, якість документації, масштабованість, легкість розгортання, крива навчання (залежність між ефективністю виконання завдання та кількістю отриманого досвіду) та обслуговування. Ось деякі з найпопулярніших фреймворків сьогодення: Node.js, Django, Ruby on Rails, Laravel, Spring Framework, та інші [30].

Spring Framework [31] є одним з найпопулярніших фреймворків, що виділяється своєю гнучкістю, масштабованістю та широким набором функціональних можливостей. Головна перевага Spring Framework полягає в наявності великої кількості готових рішень, які потребують лише певних конфігурацій, що відповідно прискорює та здешевлює розробку. Spring

Framework пропонує надійну та гнучку структуру на основі платформи Java. Він широко відомий тим, що має повний набір інструментів, які полегшують написання чистого, підтримуваного коду, скорочують час розробки та підвищують продуктивність. Spring Framework придатний для створення широкого діапазону додатків, від простих веб-застосунків до складних високопродуктивних систем.

Однією зі значних переваг Spring Framework є його універсальність та сумісність з різними сторонніми сервісам. Це обумовлено тим, що за рахунок його популярністю, в свій час, користувачами Spring було створено дуже багато інтеграцій з багатьма сервісами. Також, за рахунок цієї потужної підтримки спільноти та відкритого вихідного коду, Spring Framework має численні навчальні посібники, багато живих форумів, різні інтернет ресурси, та багато чого іншого. Все це, сильно спрощує та допомагає у вирішенні будь-яких проблем, які можуть виникати під час процесу розробки.

Також Spring Framework має багато розширень для підтримки різних типів проектів, включаючи вебдодатки (вбудований вебсервер Apache Tomcat) та модулі, що спрощують взаємодію з базою даних за допомогою простої у використанні високорівневої структури (Hibernate), яка абстрагує низькорівневі деталі (JDBC).

На додаток, практично теж саме можна сказати і про саму Java [32], яка э рідною для Spring Framework. Мова програмування , яка себе зарекомендувала, як надійне рішення, та знайшла широке застосування в розробці додатків різного типу та масштабу, від корпоративних систем, до різних веб-застосунків, та навіть мобільних додатків (особливо для Android) за рахунок своєї незалежності від конкретної платформи. Ця мультиплатформенність також передається і самому Spring Framework від Java, але насправді, у випадку розробки веб-додатку це не так критично, так як сам браузер її поставляє. Також, Java — це потужна підтримка спільноти, численні бібліотеки з відкритим вихідним кодом та багато чого іншого.

Тому взявши до уваги ці фактори, можна сказати, що Spring Framework ідеально підходить для реалізації веб-додатку, та інфраструктури проекту в цілому, а Java — для реалізації основної бізнес-логіки на стороні серверу (бекенду).

### 3.2.2 Вибір технології для реалізації клієнтської частини

В більшості випадках, реалізація клієнтського інтерфейсу включає використання трьох основних технологій, а саме: HTML [33], CSS [34] та JavaScript [35]. Однак, як і в випадку з бекендом, фронтенд має також дуже багато різних бібліотек та потужних фреймворків , які можуть зробити цей процес розробки більш ефективним. Наприклад, одні із найпопулярніших фронтенд фреймворків та бібліотек сучасності, є: React.js, Angular, Vue.js, jQuery, Bootstrap та інші.

Вибір між цими технологіями залежить в першу чергу від конкретних вимог проекту, складності програми та знайомства команди розробників з цими інструментами.

Проаналізувавши вимоги проекту, було отримано те, що система не потребує якогось складного інтерфейсу з динамічними ефектами. Це має бути простий у використанні, мінімалістичний інтерфейс, тому потреби в якихось потужних фрейморках, таких як React.js, Angular чи Vue.js немає. Тому для реалізації відповідного клієнтського інтерфейсу системи має бути достатньо базового HTML, CSS та JavaScript.

Однак для прискорення розробки клієнтського інтерфейсу, вчасності його стилістики, було вирішено, все таки підключити відносно легкий та популярний CSS фреймворк — Bootstrap [36]. Дана технологія полегшуй та пришвидшує розробку адаптивного та привабливого інтерфейсу користувача, головним чином завдяки попередньо створеним компонентам та шаблонам. Серед популярних компонентів інтерфейсу — це різні панелі навігації, каруселі, діалогові вікна, розкриваючі списки, та багато чого іншого. На додаток, незважаючи на свою простоту, він є достатньо гнучким, дозволяючи налаштувувати різні компоненти, коли це необхідно. Наприклад, відповідно до конкретних вимог проекту, розробники можуть вирішувати, які компоненти та функції Bootstrap виключати, а які ні. Та як і будь який популярний фреймворк, Bootstrap має велику та активну спільноту. Показником цього э чисельні навчальні ресурси, туторіали, активні форуми та багато чого іншого.

### 3.2.3 Вибір технології зберігання даних

В якості одного з механізмів зберігання даних було обрано реалізацію реляційної бази даних, а саме СУБД MySQL [37], яка за функціональністю порівняна з передовими СУБД сучасності та при цьому є відносно простою у використанні.

MySQL є однією з найпопулярніших баз даних, яка застосовується для створення ІС у веб середовищі, за рахунок своєї зручності (викликана великою кількістю якісного програмного забезпечення) та гнучкості. Вже не перший рік MySQL доводить, що стала непорушним стандартом в області СУБД для веб. За рахунок цих аргументів вибір бази даних пав саме на MySQL, а тепер в ній ще й розвиваються можливості для використання її в будь-яких критичних бізнес-додатках [38].

Основні переваги MySQL:

- багатопоточність, підтримка декількох одночасних запитів;
- широкий ряд інструментів для взаємодії з базою даних;
- записи фіксованої і змінної довжини;
- ODBC драйвер;
- гнучка система привілеїв і паролів;
- швидка робота, масштабованість;
- безкоштовна в більшості випадків;
- гнучка підтримка форматів чисел, рядків змінної довжини і міток часу;
- оптимізація зв'язків з приєднанням багатьох даних за один прохід.

Сервер забезпечує надійність, стабільність, стійкість, гнучку підтримку форматів чисел, рядків змінної довжини і міток часу, продуктивність та якісну сервісну підтримку, актуальну підтримку багатопоковості та багато чого іншого [39].

## 3.3 Розробка серверної частини вебзастосунку

### 3.3.1 Проектування бази даних

#### 3.3.1.1 Опис вимог до складу сутностей та атрибутів сутностей бази даних

За результатами моделювання було визначено вимоги до складу інформації (сутностей, атрибутів сутностей), що мають зберігатися в базі даних ІС (таблиця 3.4).

Таблиця 3.4 – Перелік вимоги до складу інформації, що мають зберігатися в базі даних

| Сутність | Поле | Опис |
|---|---|---|
| Employee (робітник підприємства) | id | унікальний індивідуальний номер |
| | username | ім’я користувача (робітника) в системі |
| | first_name | ім’я робітника |
| | last_name | фамілія робітника |
| Product (товар) | id | унікальний індивідуальний номер |
| | image_src | адреса зображення товару |
| | name | назва товару |
| | manufacturer | виробник товару |
| | price | ціна продукту за умовну одиницю |
| | barcode | штрих-код продукту (унікальний літерал) |
| | is_required_verification | чи потребує верифікації кошик з даним товаром |
| Receipts (покупка клієнта) | id | унікальний індивідуальний номер |
| | created_at | дата та час створення (закриття) покупки |
| | approver | робітник підприємства, що підтвердив дану покупку у разі необхідності |
| | cart_details | дані по кошику: обрані товари, їх кількість, ціна на момент покупки |

#### 3.3.1.2 Розробка моделі даних

На основі вимог розроблених на етапі аналізу предметної області була складена логічна модель даних. Та в подальшому, після проектування логічної моделі даних, вона була транспортована у фізичну форму.

Перелік сутностей моделі даних системи та короткий опис їх полів (таблиця 3.5).

Таблиця 3.5 – Перелік сутностей моделі даних системи

| Сутність | Поле | Тип даних | Обмеження | Опис |
|---|---|---|---|---|
| employees (робітники підприємства) | id | INT | NOT NULL, unique, (PK) | унікальний індивідуальний номер |
| | username | VARCHAR(255) | NOT NULL, unique | ім’я користувача (робітника) в системі |
| | first_name | VARCHAR(255) | NOT NULL | ім’я робітника |
| | last_name | VARCHAR(255) | NOT NULL | фамілія робітника |
| products (товари) | id | INT | NOT NULL, unique, (PK) | унікальний індивідуальний номер |
| | image_src | VARCHAR(255) | NULL | адреса зображення товару |
| | name | VARCHAR(255) | NOT NULL | назва товару |
| | manufacturer | VARCHAR(255) | NULL | виробник товару |
| | price | DECIMAL(9, 2) | NOT NULL, >= 0 | ціна продукту за умовну одиницю |
| | barcode | VARCHAR(128) | NOT NULL, unique | штрих-код продукту (унікальний літерал) |
| | is_required_verification | TINYINT(1) | NOT NULL | чи потребує верифікації кошик з даним товаром |
| receipts (покупки клієнтів) | id | INT | NOT NULL, unique, (PK) | унікальний індивідуальний номер |
| | created_at | DATETIME | NOT NULL | дата та час створення (закриття) покупки |
| | approver_id | INT | NULL, (FK) | ідентифікатор робітника підприємства, що підтвердив дану покупку у разі необхідності (зовнішній ключ) |
| cart_entries (записи в кошику) | id | INT | NOT NULL, unique, (PK) | унікальний індивідуальний номер |
| | receipt_id | INT | NOT NULL, (FK) | ідентифікатор покупки (зовнішній ключ) |
| | product_id | INT | NOT NULL, (FK) | ідентифікатор товару (зовнішній ключ) |
| | amount | INT | NOT NULL, >= 0 | кількість товарів у відповідній покупці |
| | unit_price | DECIMAL(9, 2) | NOT NULL, >= 0 | ціна продукту за умовну одиницю на момент створення відповідної покупки |

В процесі переходу до фізичної моделі було ініціалізовано нові обмеження, встановлено типи даних та допустимість нульових значень атрибутів. На основі логічної моделі була розроблена фізична модель даних для відповідної платформи СУБД, а саме була обрана MySQL. На відміну логічної моделі даних, фізична, на додаток  відображає первинні та зовнішні ключі, типи даних полів,

атрибути полів «NULL» і «NOT NULL», кардинальність зв’язків. Наприклад на такі атрибути як ціна та кількість товарів, були накладені обмеження невід’ємності, а на атрибут «username» в сутності «ymploees» були накладені додаткові обмеження унікальності.

#### 3.3.1.3 Реалізація механізму зберігання даних

Реалізація механізму зберігання даних для відповідної інформаціної системи була розділена на декількох основних частин (етапів проектування).

Етапи проектування механізму зберігання даних:

– вибір оптимальної платформи реляційної СУБД та відповідне середовище розробки бази даних;

– реалізації попередньо розробленої фізичної моделі даних у вигляді бази даних на обраній платформі СУБД. В частості цей етап проектування відбувається за рахунок CASE-методу прямого реінжерінгу (forward engineer);

– реалізація необхідних типів зв’язків таблиць для забезпечення цілісності даних за допомогою інтерфейсу відповідно обраного середовища розробки;

– заповнення таблиць бази даних обов’язковими даними (константи), за часту це підставкові таблиці до основних таблиць які виконують функцію так званих енамів (enum);

– підготовка тестового набору даних тестування та імітації поведінки справжнього середовища. Це дозволяє симулювати реальні сценарії та функції, забезпечуючи широке та вичерпне тестування.

Враховуючи вимоги інформаційної системи, обрано реляційну модель для зберігання даних (SQL). На відміну від нереляційної моделі (NoSQL), вся інформація в БД завжди строго структурується, підтримується висока атомарність даних, що зберігається. У таблиці обов'язково присутні рядки (із записами) та стовпці (із типом даних).

Основні переваги використання даної системи зберігання даних (SQL):

– масштабованість;

– переносність;

– зв’язаність;

– безпека.

MySQL може підтримувати роботу БД значних розмірів, що підтверджують її реалізації в Yahoo!, Google, HP, Associated Press. Згідно документації, що додається до MySQL, деякі БД, що використовуються компанією MySQL AB (розробником MySQL), зберігають до 50 млн. записів. Також MySQL працює на різних платформах, серед яких Unix, Linux, Windows, OS/2, Solaris, Mac OC. Окрім того, MySQL працює на різних платформах. MySQL має мережеву структуру. До MySQL можна одержувати доступ із будь-якої точки Internet декільком користувачам одночасно. MySQL має цілий ряд програмних інтерфейсів додатків (Application Programming Interface – API ), які дозволяють встановлювати з'єднання з MySQL із додатків, написаних на таких мовах як C, C++, Perl, PHP, Java, Python. Також MySQL має систему контроля доступу до даних, забезпечує шифрування даних при передаванні [38].

Для управління даними обрано СУБД MySQL Workbench 8.0 CE [40].
Серед переваг використання даної СУБД:

– підтримка MySQL Enterprise: підтримка корпоративних продуктів, таких як MySQL Enterprise Backup, MySQL Firewall та MySQL Audit;

– міграція даних: дозволяє переходити з Microsoft SQL Server, Microsoft Access, Sybase ASE, SQLite, SQL Anywhere, PostreSQL та інших таблиць, об'єктів та даних RDBMS до MySQL. Міграція також підтримує перехід зі старих версій MySQL на останні версії;

– розробка SQL: дозволяє створювати та керувати підключеннями до серверів баз даних. На додаток до того, що дозволяє користувачеві налаштовувати параметри підключення, MySQL Workbench надає можливість запускати запити SQL на підключеннях до бази даних за допомогою вбудованого редактора SQL;

– моделювання даних (дизайн): дозволяє графічно моделювати схему бази даних, здійснювати зворотне та пряме проектування між схемою та активною базою даних та редагувати всі аспекти бази даних за допомогою всеосяжного редактора таблиць. Редактор таблиць надає зручні засоби для редагування таблиць, стовпців, індексів, тригерів, розділів, параметрів, вставок та привілеїв, процедур та подань;

– адміністрування сервера: дозволяє керувати екземплярами сервера MySQL, керуючи користувачами, виконуючи резервне копіювання та

відновлення, перевіряючи дані аудиту, переглядаючи стан бази даних та контролюючи продуктивність сервера MySQL.

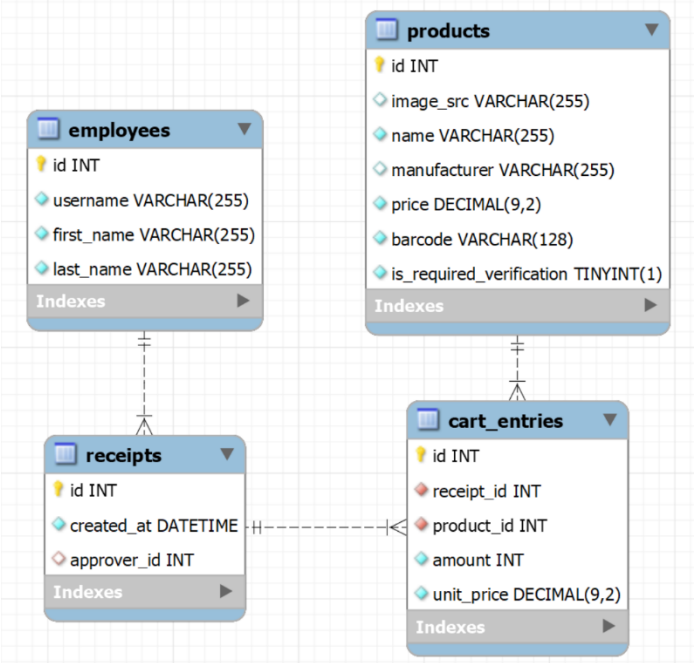



Рисунок 3.9 – Схема бази даних

Як система зберігання даних, була обрана підсистем низького рівня в СУБД MySQL InnoDB. Основною особливістю InnoDB є механізм транзакцій і зовнішніх ключів, що дозволяє значно спростити підтримку цілісності даних БД.

Підтримка механізму створення зовнішніх ключів, реалізує підтримку посилальної цілісності між таблицями. Ця можливості движку InnoDB звільняють від необхідність створення тригерів для підтримання посилальної цілісності при операціях UPDATE, INSERT та DELETE, наприклад на відміну від підсистеми СУБД MyIsam.

Таблиця 3.6 – Перелік типів посилальної цілісності зв’язків за зовнішнім ключем для всіх таблиць БД

| № | Ім’я таблиці 1, зовнішній ключ | Ім’я таблиці 2, первинний ключ | Тип посилальної цілісності для таблиці 1 | Тип посилальної цілісності для таблиці 2 | SQL-інструкція |
|---|---|---|---|---|---|
| 1 | cart_entries, | receipts, | ON UPDATE | ON UPDATE | SQL-інструкція |

| | | | | | |
|---|---|---|---|---|---|
| | receipt_id | id | RESTRICT | RESTRICT | UPDATE |
| 2 | cart_entries,<br>receipt_id | receipts,<br>id | ON DELETE<br>**CASCADE** | ON DELETE<br>**CASCADE** | SQL-інструкція<br>DELETE |
| 3 | cart_entries,<br>product_id | products,<br>id | ON UPDATE<br>RESTRICT | ON UPDATE<br>RESTRICT | SQL-інструкція<br>UPDATE |
| 4 | cart_entries,<br>product _id | products,<br>id | ON DELETE<br>RESTRICT | ON DELETE<br>RESTRICT | SQL-інструкція<br>DELETE |
| 5 | receipts,<br>stock_id | employees,<br>id | ON UPDATE<br>RESTRICT | ON UPDATE<br>RESTRICT | SQL-інструкція<br>UPDATE |
| 6 | receipts,<br>stock _id | employees,<br>id | ON DELETE<br>RESTRICT | ON DELETE<br>RESTRICT | SQL-інструкція<br>DELETE |

### 3.3.2 Реалізація вебсерверу

#### 3.3.2.1 Проектування та реалізація серверної логіки

На основі вимог до розроблюваного сервісу, було обрано одну з типових архітектур для подібних програм, а саме: багаторівнева архітектура або N-рівнева архітектура [41]. Цей архітектурний стиль поділяє програму на окремі рівні, кожен з яких містить різні компоненти, що відповідають за певні аспекти програми. Дана архітектура дуже популярна, в першу чергу через свою простоту. Також вона дуже добре поєднується з MVC патерном, який як раз ы використовується в нашій архітектурі. Канонічний варіант багаторівневої архітектури обов’язково передбачає три основних шари, а саме:

- рівень сховища (доступ до даних, repository);
- сервісний рівень (бізнес-логіка);
- рівень презентації (контролер).

Однак в залежності від складності розроблюваного застосунку, можуть додаватися додаткові шари. Наприклад рівень фасаду, четвертий рівень, який зазвичай розмінюється між сервісним рівнем та рівнем презентації. Цей рівень зазвичай використовується для розвантаження важкої логіки на сервісному рівні, викликаючи методи різних сервісів та поєднуючи їх в єдині великі бізнес-функціях для представлення їх рівню контролерів. Також, зазвичай на рівні фасаду, фігурує такий підрівень, як шар мапінгу або конвертації. Він відповідає за конвертацію доменних моделей в спеціальні об’єктами передачі даних (DTO, Data Transfer Objects) та навпаки [42].

DTO — це шаблон проектування, який описує прості об’єкти, що використовуються для передачі даних між процесами або рівнями. Однією з основних проблем, які вирішують DTO, є непотрібне розміщення внутрішніх деталей на зовнішніх рівнях програми. Доменні моделі часто охоплюють набагато більше інформації, ніж потрібно користувачу. Відкриття їх зовнішньому світу може призвести до витоку конфіденційних даних, потенційних ризиків для безпеки, збільшення складності, проблем з продуктивність та тісного зв’язку між архітектурними рівнями. Останній пункт важливіший ніж може здатися на перший погляд. API, що тісно пов’язане з моделями домену, знищує гнучкість архітектури, та робить її підтримку та еволюційність вкрай складною. Наприклад, якщо з’явиться якесь нове поле в доменній моделі, то не слід турбуватися про те, що воно може з’явитися в інтерфейсі у користувача. Чи навпаки, не слід додавати нове поле в доменну модель, якщо це поле можна підрахувати в процесі роботи та помістити в DTO. DTO дозволяють формувати ваші дані таким чином, щоб відображалася та транспортувалася лише необхідна інформація, фактично слугуючи кордоном і полегшуючи трансформацію та перевірку даних, перш ніж вони досягнуть вищих рівнів вашої програми.

Також слід пам’ятати, що в даній архітектурі, хоча і залежність між рівнями транзитивна, їх використання між рівнями не транзитивне. Це значить, що якщо рівень контролера залежить від рівня сервісу, а рівень сервісу залежить від рівня репозиторія, не бажано використовувати на рівні контролеру, класи рівня репозиторія, або нижче. Наприклад не бажано використовувати репозиторій, на рівні контролера чи далі. До речі з цього, виходить, що не бажано використовувати і доменні моделі на пряму, там на рівні контролера чи далі, в чому нам і допомагаю DTO.

Також, в подібній архітектурі можна виділити і більш дрібні, абстрактні шари, такі як: моделі домену. Це програмний шар, що містить сутності (класи) що відповідають сутностям зберігаємим у базі даних. В процесі роботи вони мапляться та наповняються даними відповідно до записів зберігаємих у базі даних.

Кожний з програмних шарів, залежить від попереднього, шар сервісу залежить від шару репозиторія, а шар контролера залежить від шару сервісу. Для реалізації та підтримки цих залежностей було використано один з ключових

механізмів Spring Framework, а саме Dependency Injection (DI, впровадження (ін'єкція) залежностей). Даний підхід значною мірою спрощує процес керування залежностями в програмі, що призводить до кращої організації, більш керованого та легкого тестування коду. Це також, по більшій мірі, реалізується за допомогою інверсії управління (IoC) [43], ми передаємо цю відповідальність за впровадження (управління) залежностями на Spring Framework.

Отже в результаті було виділено наступні шари та відповідно до них, створені необхідні класи (рисунок 3.10):

а) Domain Model Layer, рівень моделей домену, містить основні бізнес-сутності програми:

– Employee, співробітник, представляє працівника в системі;

– Product, продукт, представляє продукт у системі;

– Receipt, являє собою квитанцію в системі, основні дані про покупку;

– CartEntry, представляє запис у кошику для покупок.

б) Repository (DAO) Layer, рівень сховища, відповідає за доступ до даних і містить класи репозиторії, які взаємодіють з базою даних:

– EmployeeRepository, керує операціями з даними, пов'язаними зі співробітниками;

– ProductRepository, керує операціями з даними, пов'язаними з продуктами;

– ReceiptRepository, керує операціями з даними, пов'язаними з покупками;

– CartEntryRepository, керує операціями з даними, пов'язаними з записами в кошик.

в) Service Layer, сервісний рівень, містить бізнес-логіку програми, інкапсульовану в класах службах:

– EmployeeService та DefaultEmployeeService, інтерфейс, що надає бізнес-логіку, пов'язану зі співробітниками, та клас, що реалізує цей інтерфейс, надаючи реалізацію за умовчанням;

– ProductService та DefaultProductService, інтерфейс, що надає бізнес-логіку, пов'язану з продуктами, та клас, що реалізує цей інтерфейс, надаючи реалізацію за умовчанням;

– ReceiptService та DefaultReceiptService, інтерфейс, що надає бізнес-логіку, пов’язану з покупками, та клас, що реалізує цей інтерфейс, надаючи реалізацію за умовчанням;

– CartEntryService та DefaultCartEntryService, інтерфейс, що надає бізнес-логіку, пов’язану зі записами в кошик, та клас, що реалізує цей інтерфейс, надаючи реалізацію за умовчанням.

г) Converter Layer, шар конвертації (мапінгу), обробляє перетворення між об’єктами домену та DTO:

– DefaultProductConverter, обробляє перетворення даних, пов’язаних із продуктом.

д) Domain DTO Layer, рівень доменних DTO, включає класи, що використовуються для передачі даних між різними рівнями програми, вчасності доменними моделями:

– DefaultProductDto, DTO для даних, пов’язаних із продуктом.

е) Other DTO Layer, включає інші DTO, які використовуються для певних цілей у програмі:

– ProductSearchParameter, DTO для параметрів пошуку, пов’язаних із продуктами;

– SessionCart, DTO для даних кошика сеансу;

– EntryInfo, DTO для записів у кошику севнсів.

ж) Controller Layer, рівень контролера, обробляє HTTP-запити та відповіді, діючи як інтерфейс між користувачем і програмою:

– PurchaseController, керує операціями, пов’язаними з покупками, і взаємодіє з рівнем обслуговування.

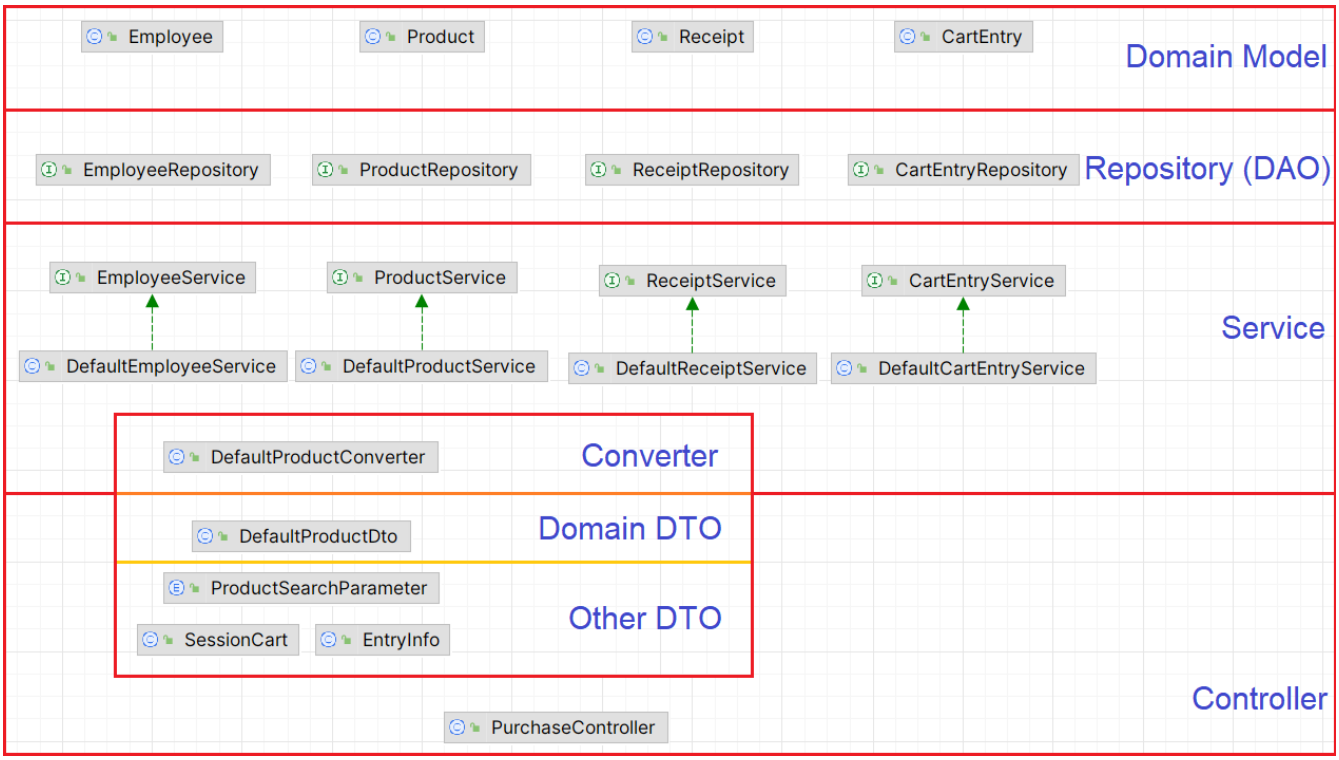



Рисунок 3.10 – Архітектура сервісу обробки послуг каси самообслуговування

Більш детальна інформація про публічне API на кожному з рівнів зображена на рисунках 3.11 – 3.12.


Рисунок 3.11 – API на рівні сервісу

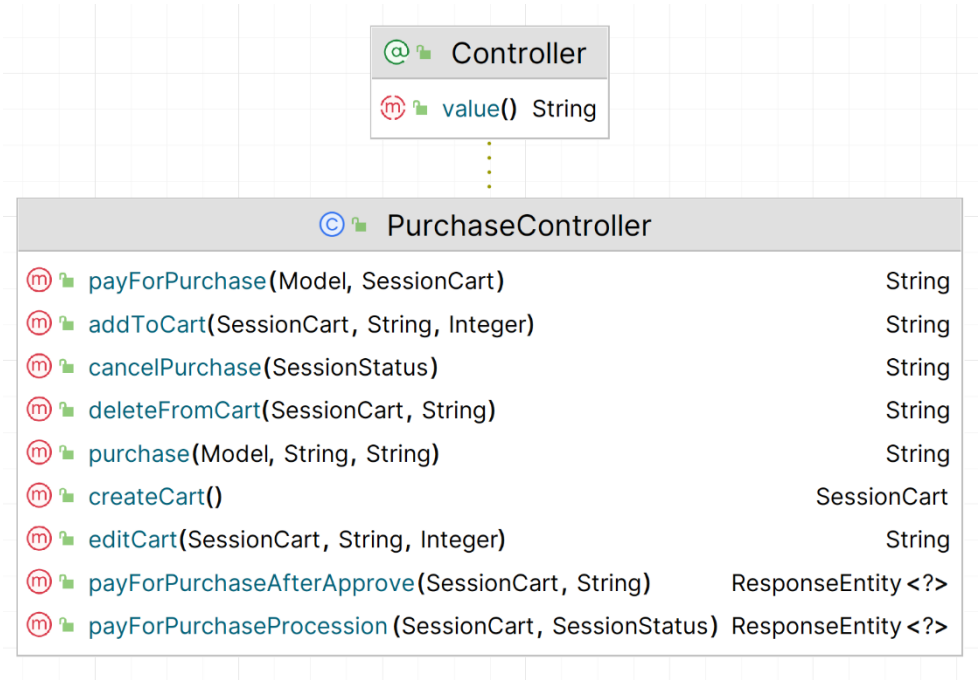



Рисунок 3.12 – API на рівні контролера

### 3.3.2.2 Реалізація логіки взаємодії з базою даних

Так як відповідний сервіс обробки покупок через касу самообслуговування базується на Spring Framework, вчасності Spring Boot, доцільно було б використати один з його найпотужніших модулів, а саме Spring Data JPA, який зосереджується на підтримці Java Persistence API, специфікація, що описує взаємодію між Java програмами та можливими механізмами зберігання даних, вчасності базами даних. Ця специфікація описую наче міст між об'єктно-орієнтованими моделями домену (Java об'єктами) та відповідною базою даних.

Spring Data JPA дозволяє розробникам легко зберігати та отримувати дані за допомогою абстракції JPA, що фактично відповідає програмному рівню доступу до даних. В свою чергу Spring Boot Starter, по замовченню постачає мінімально потрібний набір конфігурацій для роботи.

Як і у всього Spring фреймворку, основна перевага даного модулю, це гнучкість та швидкість розробки. Він скриває в себе «під капотом» увесь шаблонний код та різні шаблони, що абстрагує API доступу до системи зберігання даними, та дозволяє розробникам більше фокусуватися на якійсь специфічній логіці, та не витрачати час на написання шаблонного коду. До цього, в першу чергу, відносяться CRUD операції (Create, Read, Update, Delete)

(створення, читання, оновлення, видалення). Для їх використання, слід створити свій інтерфейс, відповідно до вашої сутності, та розширити його від спеціально інтерфейсу, а саме JpaRepository, та в дженеріку вказати тип вашої сутності та тип ідентифікатору (рисунок 3.13).

```
5
    4 usages
6   public interface ReceiptRepository extends JpaRepository<Receipt, Integer> {
7   }
8
```

Рисунок 3.13 – Створення інтерфейсу доступу до бази даних для сутності `Receipt`

Також, це методи запитів — абстракція, що дозволяє виконувати запити, просто визначивши метод інтерфейсу, при цьому конвенція найменування методу для визначення нових запитів є інтуїтивно зрозумілою та проста для сприйняття. Наприклад:

Метод: `List<Product> findByBarcodeContains(String name);`

Еквівалентний запиту: `SELECT * FROM products WHERE barcode LIKE '%?%';`

В результаті, відповідно до кожної з сутностей були створені свої інтерфейси репозиторії (рисунок 3.14).

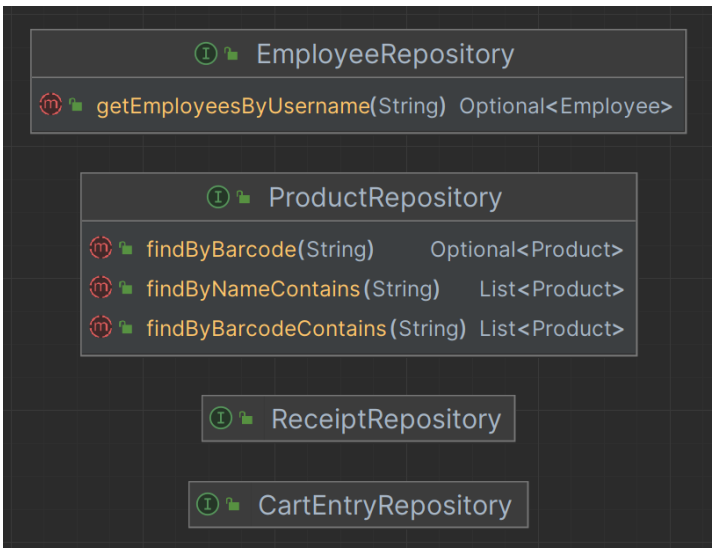



Рисунок 3.14 – Створенні інтерфейси доступу до бази даних

Іншою важливою перевагою подібних фреймворків, є їхня незалежність від конкретної бази даних. В першу чергу, це реалізується через спеціальну мову запитів. Наприклад, Spring Data JPA, використовує Hibernate реалізацію, що дозволяє розробникам створювати запити до бази даних за допомогою HQL (Hibernate Query Language), яка не залежить від типу використовуваної бази даних. Таким чином, розробники можуть перемикати бази даних, просто змінюючи кілька конфігурацій, не змінюючи кодову базу. Запит з останнього прикладу еквівалентний наступному: `FROM Products WHERE barcode LIKE :barcode`.

Spring Data JPA також надає багато іншого функціоналу, це керування транзакціями для забезпечення цілісності даних. Це вбудований аудит для відстеження змін у даних. Це також потужні механізми кешування для підвищення продуктивності програми. Окрім цього, це підтримка відкладеного завантаження (Lazy Loading) та підтримка розбиття сторінок (пагінація) з сортування, що дуже корисно при роботі з великими обсягами даних.

Hibernate Framework, що використовується в Spring Data JPA — це в першу чергу, ORM (Object/Relational Mapping). Це те, що дозволяє на основі Java класів та спеціальних анотацій створювати схеми баз даних, та в подальшому записи в базі даних автоматично мапити на реальні Java об’єкти, що є надзвичайно зручно.

Щоб Hibernate зрозумів, які йому об’єкти мапити до сутностей в базі даних, їх треба позначити спеціальними анотаціями. Вчасності, це анотації:

– @Entity, використовується на рівні класу, та визначає Hibernate сутність. За замовчуванням ім'я класу використовується як ім'я таблиці. Якщо потрібна специфічна назва таблиці, то це можна вказати через проперті анотації `name`;

– @Id, використовується на рівні поля, щоб позначити поле як ідентифікатор або первинний ключ сутності. Кожна сутність повинна мати принаймні одне поле з анотацією цією анотацією. Зазвичай з нею в парі використовують іншу анотацію, а саме: @GeneratedValue. Через неї вказують стратегію генерації значень для первинного ключа. Існує чотири стратегії:

1) AUTO – стратегія за замовченням, що залишає провайдеру персистентності право вибору найбільш підходящої стратегії. Обрана стратегія залежить від діалекту конкретної бази даних;

2) IDENTITY – встановлення ідентифікатора відбувається на стороні бази даних, та генеруються під час операції вставки в базу даних, не раніше.

Зазвичай виконується через ключове слово IDENTITY у SQL, що може призвести до автоматичного створення ідентифікатора базою даних;

3) SEQUENCE – встановлення ідентифікатора відбувається за рахунок спеціального об’єкта послідовність у базі даних (не всі бази даних підтримують дану стратегію);

4) TABLE – стратегія, що використовує окрему таблицю для створення ідентифікаторів. Це найменш ефективна стратегія, оскільки їй потрібно виконати додатковий доступ до таблиці перед звичайною вставкою (пошук останнього згенерованого ідентифікатора), що також блокує таблицю. Однак, це хороший вибір, насамперед, у випадках, коли база даних не підтримує послідовності або коли база даних повинна генерувати первинні ключі незалежно, наприклад, у розподіленій системі.

– @Column, не обов’язково вказувати (за замовчення всі поля класу, який анотований @Entity, враховуються як стовпці сутності, та кожне з них буде зіставлене з назвою поля та його типом). Використовується на рівні поля, щоб вказати деталі стовпця, такі як `name` (щоб встановити назву стовпця), `nullable` (щоб вказати, чи може стовпець мати нульові значення), `length` (щоб визначити довжину стовпця) тощо.

На рисунку 3.15, на прикладі класу `Product`, зображено Hiberante сутність.

```java
@Entity(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String imageSrc;

    @Column(nullable = false)
    private String name;

    private String manufacturer;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(nullable = false, unique = true)
    private String barcode;

    @Column(nullable = false)
    private Boolean isRequiredVerification;

    @OneToMany(mappedBy = "product")
    private List<CartEntry> cartEntries = new ArrayList<>();
}
```

Рисунок 3.15 – Hiberante сутність `Product`

Також, можна помітити анотацію @OneToMany, так, Hiberante також дозволяє будувати взаємозв’язки між сутностями. Вчасності, Hiberante пропонує 4 типи зв’язки, а саме:

– @OneToMany – використовується для визначення зв’язку «один до багатьох» між двома сутностями. Рядок, який відмічений даною анотацією, буде пов’язаний з кількома рядками іншої сутності. Тобто дана сутність може відноситися до багатьох інших, тому в коді ми і визначаємо відповідне поле як список пов’язаних сутностей (List<CartEntry> cartEntries).

– @ManyToOne – використовується для визначення зв’язку «багато до одного» між двома сутностями. Зазвичай дана анотація використовується в парі з попередньою (@OneToMany), тільки вже вказується над полем, тип якого відповідає сутності до якої відноситься дана (рисунок 3.16). У цьому випадку кілька рядків сутності, що має анотацію @ManyToOne, відносяться до одного рядка іншої сутності.

– @OneToOne – використовується для визначення зв’язку один-до-одного між двома сутностями. У зв’язку @OneToOne один екземпляр сутності пов’язаний з одним і лише одним екземпляром іншої сутності.

– @ManyToMany – використовується для визначення зв’язку «багато-до-багатьох» між двома сутностями. У зв’язку @ManyToMany кілька рядків сутності можуть бути пов’язані з кількома рядками іншої сутності.

Окрім декларації зав’язків між сутностями, через Hiberante також можна налаштовувати і тип посилальної цілісності між таблицями. Наприклад, використовуючи анотацію @OnDelete, можна визначити що робити з сутностями пов’язаними з даними. Має бути вказано над полем, що вказує на пов’язану сутність, та над самою пов’язаною сутністю (рисунки 3.16 – 3.17). Існує декілька можливих дій, які можна виконувати під час видалення пов’язаних сутностей, а саме, це:

– OnDeleteAction.NO_ACTION: вказує, що не потрібно виконувати жодних дій при видаленні сутності;

– OnDeleteAction.CASCADE: каскадні видалення через зв’язки. Якщо сутність видалено, пов’язані з нею сутності також буде автоматично видалено;

– OnDeleteAction.RESTRICT: відхиляє операцію видалення для батьківської таблиці, якщо в поточній дочірній таблиці є запис;

– OnDeleteAction.SET_NULL: встановлює значення null у полі зовнішнього ключа дочірньої таблиці під час видалення запису з батьківської таблиці, від якого залежить запис у дочірній таблиці;

– OnDeleteAction.SET_DEFAULT: встановлює для поля зовнішнього ключа в дочірній таблиці значення за замовчуванням під час видалення запису з батьківської таблиці, від якого залежить запис у дочірній таблиці.

Окрім декларації специфічної поведінки при видаленні, JPA специфікація також надає інші способи перехоплення та реагування на подію над сутностями, наприклад такі анотації, як @PreUpdate та @PostUpdate, що вказуються над методами, де можна, при потребі, прописати логіку взаємодії з іншими сутностями при їх зміні.

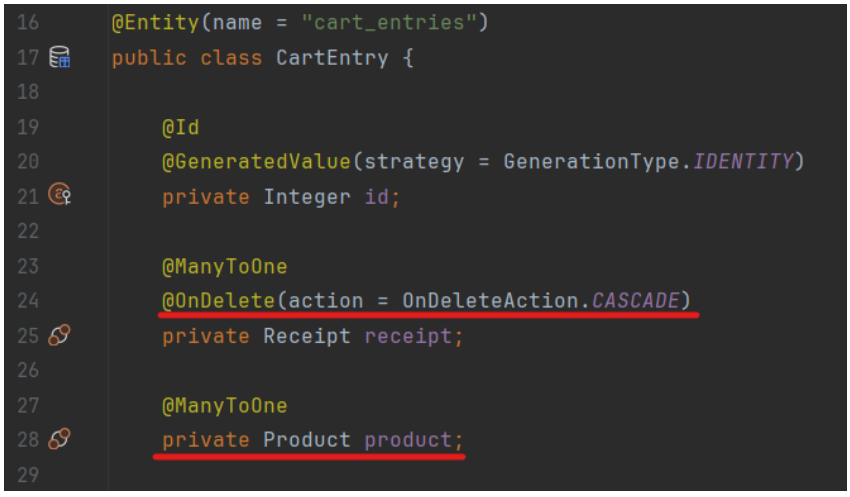



Рисунок 3.16 – Зв’язок @ManyToOne та тип CASCADE посилальної цілісності

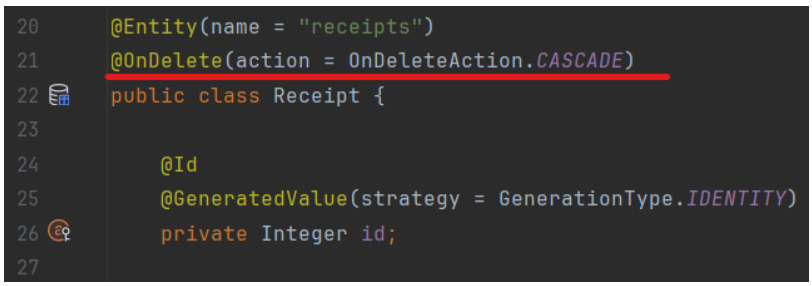



Рисунок 3.17 – Тип CASCADE посилальної цілісності

В результаті було створено всі чотири Hibernate сутності (а саме: `Employee`, `Product`, `Receipt` та `CartEntry`) відповідно до попереднього проектування моделі даних (рисунок 3.18).

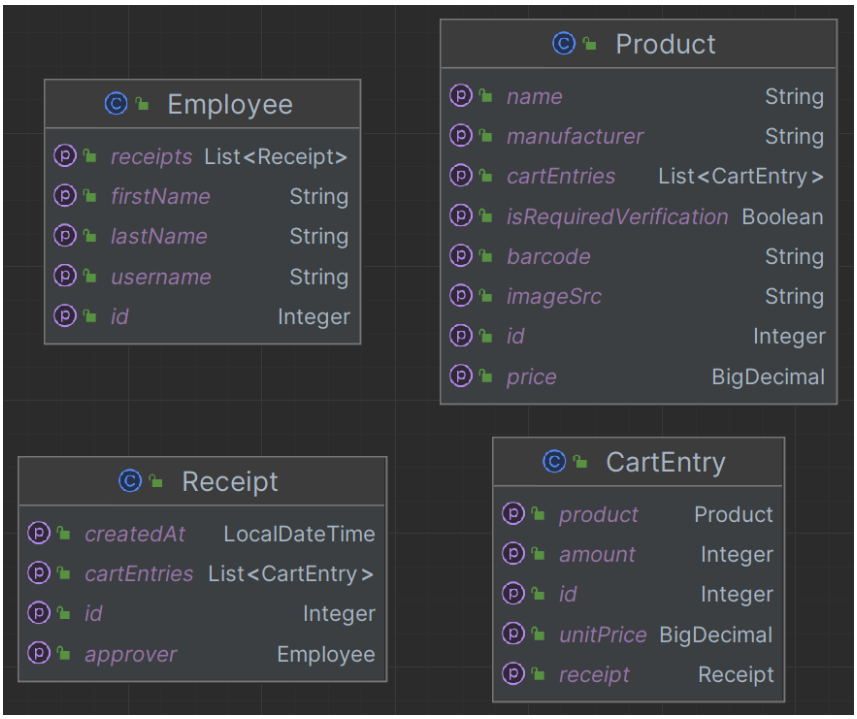



Рисунок 3.18 – Створені Hibernate сутності

Хоча Hiberante і здатний генерувати на основі задекларованих сутностей відповідну схему бази даних, це не вважається хорошою практикою. Надання Hibernate повноважень змінювати схему кожного разу, коли змінюються сутності, може бути корисним на стадіях розробки чи тестування, або для якихось відносно простих схем, але це не рекомендовано для робочих середовищ. Основні з причин, це:

– не повноцінний контроль: в деяких випадках Hibernate може створити неоптимальну схему;

– портативність баз даних: інструмент генерації схем Hibernate не повністю сумісний з усіма типами баз даних, в першу чергу, через відмінності в типах даних та можливі обмеженнях у різних системах баз даних;

– ризик втрати даних: якщо Hibernate налаштовано на створення схеми під час кожного розгортання (що за замовчуванням), існуючі дані можуть бути втрачені, оскільки таблиці можуть бути видалені та створені заново.

Хорошим рішенням може бути: окремо обробляти зміни в схемі бази даних, використовуючи спеціальний інструмент міграції схеми, наприклад Flyway або Liquibase. Подібні технології побудовані на принципах системи контролю версій, подібних до тих, що використовуються при розробці програмного забезпечення. Це дає більше контролю над змінами у схемах та

дозволяє відстежувати їх версії в результаті забезпечуючи їх узгодженість у різних середовищах (тестових, або у вже робочих). Кожна із змін, створює в спеціальній таблиць запис, ще його називають changelog або changeset (журнал змін), по яким потім можна буде відновити певну версію схеми бази даних. Окрім генерації схем баз даних, подібні технології на додаток підтримують імпортування даних (datasets) у різних форматах, таких як JSON, CSV чи звичайний SQL. І зазвичай вони підтримують свій діалект (синтаксис), що не залежить від конкретної бази даних, що також дуже гнучко та зручно.

В проекті було використано Liquibase. Це універсальний механізм, здатний обробляти складні сценарії розгортання завдяки підтримці попередніх умов, контекстів та міток. Liquibase зосереджується на обробці трьох типів змін об'єктів бази даних, а саме: структурні зміни (наприклад, створення, модифікація або видалення таблиць), зміни моделі даних (наприклад, вставка, оновлення або видалення даних), а також збережені процедури/тригери.

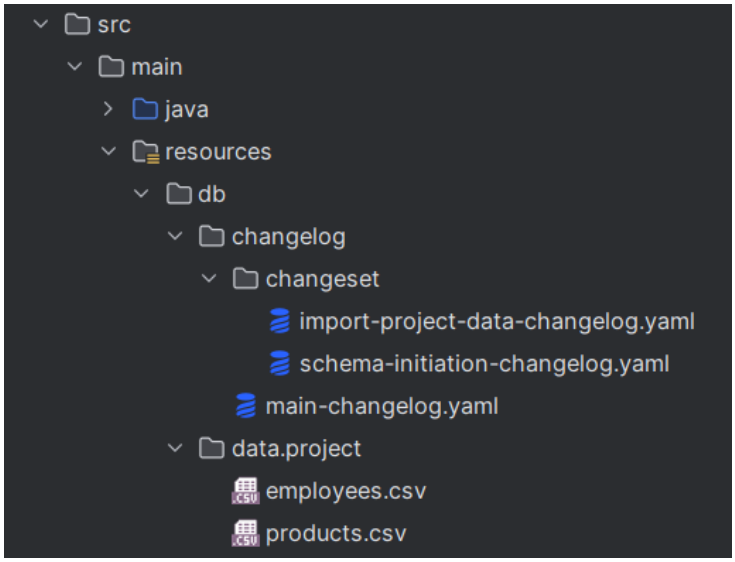



Рисунок 3.19 – Журнали змін (changelog) та дані для імпорту у проекті

Liquibase підтримує два формати для changelog файлів, а саме XML та YAML. Також changelog файли можуть включати інші changelog файли. Для зручності, зазвичай створюють головний changelog, через який викликаються другорядні changelog файли, такі як створення схеми баз даних чи імпорт різних типів даних, наприклад тестових або якихось специфічних даних (essential) для підтримки працездатності системи (рисунки 3.20).

```yaml
databaseChangeLog:

  - include:
      file: changeset/schema-initiation-changelog.yaml
      relativeToChangelogFile: true

  - include: # import some test data for imitation real environment
      file: changeset/import-project-data-changelog.yaml
      relativeToChangelogFile: true

```

Рисунок 3.20 – Головний changelog файл

Звичайні ж changelog файли зазвичай включають, так звані, changesets. Це кейси, що описують конкретну, зазвичай одну, дію. Це може бути імпорт якихось даних за вказаним шляхом (рисунок 3.21), чи це може бути створення таблиці (рисунок 3.23), чи створення зовнішнього ключа для якоїсь таблиці (рисунок 3.22).

```yaml
databaseChangeLog:

  # Import project data

  - changeSet:
      id: fill_products_table
      author: admin
      changes:
        - loadData:
            tableName: products
            file: db/data/project/products.csv
```

Рисунок 3.21 – Changelog файл з імпортом тестових даних

```yaml
- changeSet:
    id: add_fk_receipts_on_employee
    author: admin
    objectQuotingStrategy: QUOTE_ONLY_RESERVED_WORDS
    changes:
      - addForeignKeyConstraint:
          baseColumnNames: approver_id
          baseTableName: receipts
          constraintName: FK_RECEIPTS_ON_EMPLOYEE
          referencedColumnNames: id
          referencedTableName: employees
```

Рисунок 3.22 – Changelog файл створення foreign key

```yaml
- changeSet:
    id: create_receipts_table
    author: admin
    objectQuotingStrategy: QUOTE_ONLY_RESERVED_WORDS
    changes:
      - createTable:
          columns:
            - column:
                autoIncrement: true
                constraints:
                  nullable: false
                  primaryKey: true
                  primaryKeyName: pk_receipts
                name: id
                type: INT
            - column:
                constraints:
                  nullable: false
                name: created_at
                type: DATETIME
            - column:
                name: approver_id
                type: INT
          tableName: receipts
```

Рисунок 3.23 – Changelog файл створення таблиці `receipts`

### 3.3.2.3 Тестування серверної логіки

Тестування є невід'ємною частиною розробки будь якого програмного застосування. Воно представляє з себе перевірку відповідності між реальною та очікуваною поведінкою програми. Існує досить багато різних типів тестів, наприклад такі як unit тести окремих модулів чи інтеграційні тести взаємодії між модулями, наприклад між вебсервером та сервером бази даних.

Взагалі, на скільки важливим є тестування? Тестування відіграючи вирішальну роль у забезпеченні якості, функціональності та надійності програмного забезпечення. Надійна стратегія тестування гарантує, що в продукті немає помилок, збоїв чи інших проблем, які можуть перешкоджати його функціональності. Він перевіряє відповідність продукту бізнес-вимогам і потребам користувачів. Крім виявлення дефектів, тестування дозволяє аналізувати продуктивність за різних умов, забезпечуючи тим самим масштабованість і надійність. Крім того, регулярне тестування на ранніх стадіях

процесу розробки може запобігти дорогим виправленням на майбутніх етапах, тим самим заощадивши час і ресурси. По суті, тестування є фундаментальним для створення програмного забезпечення, якому користувачі довіряють і на яке можуть покластися.

Різні типи тестів, як-от модульні тести, інтеграційні тести, функціональні тести та наскрізні тести, забезпечують кілька рівнів захисту від програмних проблем. Хоча написання тестів іноді може здатися трудомістким, інвестиції загалом того варті з точки зору якості та зручності програмного забезпечення.

При розробці додатку було використано архітектурний патерн Controller-Service-Repository. Хоча він дещо і схожий на концепцію MVC (Model-View-Controller), однак він дещо відрізняється. Як вже було сказано Repository в даному шаблоні представляє з себе рівень для взаємодії з моделями та виконання операцій БД. Controllers містить в собі логіку програми та передачу введених користувачам даних до Services, в свою чергу Services — це проміжне програмне забезпечення між контролером (Controllers) і репозиторієм (Repository). Збирає дані з контролера, виконує перевірку та бізнес-логіку, а також викликає репозиторії для маніпулювання даними.

Даний патерн дуже поширений не тільки за свою простоту, а за те, що він чудово справляється з розділенням проблем. Наприклад якщо код пов’язаний із зберіганням/вилученням даних, він повинен потрапити в репозиторій. Якщо він має справу з розкриттям функціональних можливостей, він потрапляє в контролер. Будь-що унікальне в бізнес-логіці буде входити на рівень сервісу. Репозиторію не важливо, який компонент його викликає; воно сліпо робить те, що його просять. Рівню сервісу не важливо, як до нього отримати доступ, він просто виконує свою роботу, використовуючи репозиторій, де потрібно. А контролер просто передає роботу на рівень сервісу, щоб він міг залишатися гарним і м’яким.

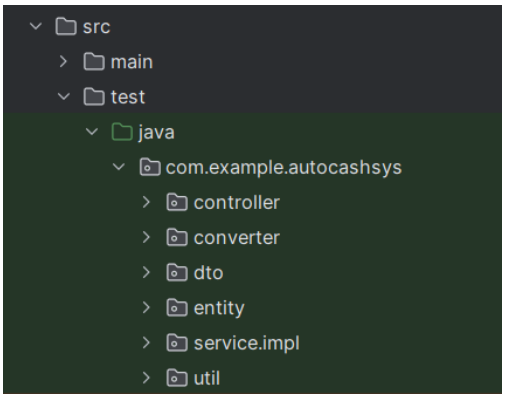



Рисунок 3.24 – Проектне розміщення тестових пакетів

На перший погляд може здатися, що даний підхід є зайвим, так як очевидних привілеїв він не надає, однак він дійсно починає приносити дивіденди в філософії модульного тестування. Завдяки чіткому розділенню проблем ми можемо мокувати (надання (реалізація) об'єкта з заданими аспекти програмного оточення, що моделюється) сусідні шари і турбуватися лише про те, щоб перевірити проблеми цього конкретного шару. Наші тести контролера турбуються лише про коди відповідей та значення, і ми можемо мокувати сервіс, щоб викликати ці умови. Наприклад Рівень сервісу можна навіть протестувати як POJO (Plain Old Java Object – простой Java об'єкт, який не є наслідником від якогось специфічного об'єкта і не реалізує ніяких службових інтерфейсів зверх тих, які потрібні для бізнес-моделей), мокуючи репозиторій, ми можемо перевірити всю бізнес-логіку в ньому, не турбуючись про те, щоб перевіряти рівень контролера.

Доречі про мокування. Мокінг є ключовим аспектом модульного тестування в розробці програмного забезпечення, і для цього є досить багато хороших технологій, вчасності один з найпопулярніших фреймворків в окрузі Java з відкритим вихідним кодом — це Mockito Framework [44]. «Mock» (макет) об’єкти імітують реальні об’єкти, відтворюючи їхню функціональність, що допомагає ізолювати тестований компонент і перевіряти взаємодію між об’єктами. Mockito спрощує створення макетів об’єктів, завдяки своєму відносно простому API. Це усуває потребу в шаблонному коді, роблячи ваші тести чистими, зрозумілими та без помилок. Mockito також включає «перевірку поведінки», дозволяючи вам перевірити, чи викликаються ваші методи макетного об’єкта з потрібними параметрами. Крім того, він підтримує

«stubbing», де ви можете диктувати вихід методів вашого макетного об’єкта, допомагаючи вам емулювати різні сценарії у ваших тестах. Загалом Mockito робить ваші тести менш крихкими, більш читабельними та заохочує до написання зручного для тестування коду, що зрештою покращує якість вашого програмного забезпечення.

В результаті було протестовано програмний шар сервісу який включає базовий функціонал кожної з базових сутностей, такі як: `ProductService`, `ReceiptService`, ` Employee Service` та ` CartEntryService`. А також програмний шар контроллера, з його класами: `GlobalControllerAdvice`, `HomeController` та `PurchaseController`. Та інші проміжні та утилітні класи, такі як: `StringUtils` та `DefaultProductConverter` (рисунок 3.25).

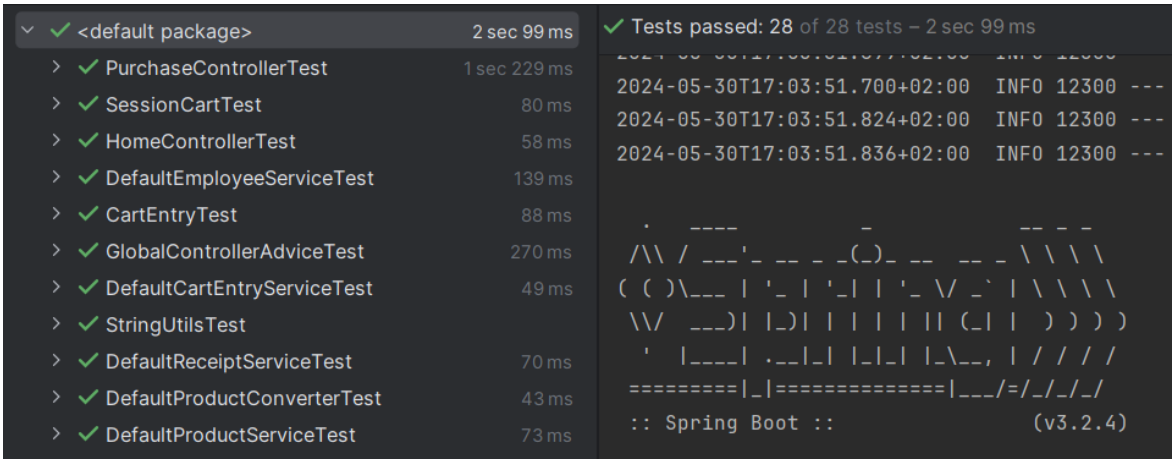
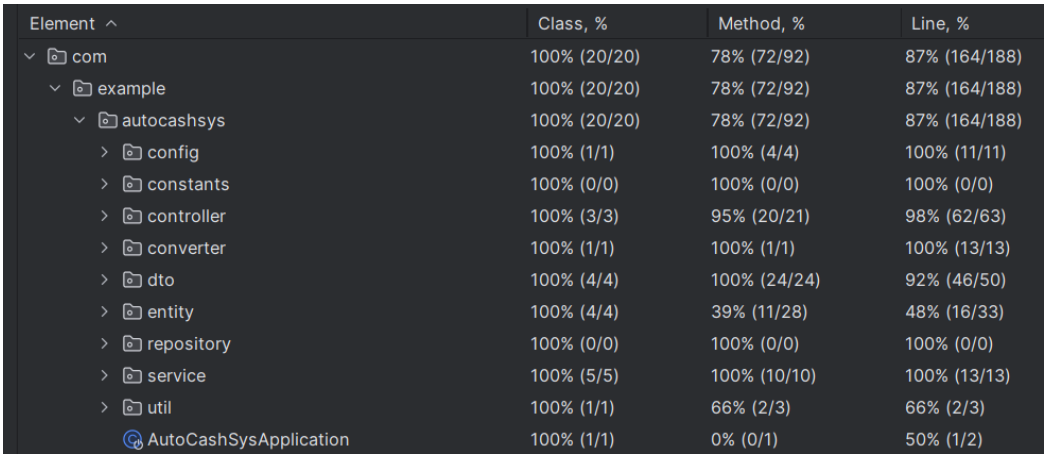



Рисунок 3.25 – Результат відпрацювання тестів

Хорошим індикатором якості ваших тестів та додатку в загалом, є рівень покриття вашого коду тестами (code coverage). Це важливий показник у тестуванні програмного забезпечення, який вимірює кількість коду, охопленого тестами, часто представленого у відсотках. Його основна функція полягає в тому, щоб показати, які частини кодової бази досягають тести, а які розділи пропущені. Високе охоплення вказує на те, що більшість кодової бази охоплено тестами, таким чином зменшуючи ймовірність невиявлених помилок (рисунок 3.26).



| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| com | 100% (20/20) | 78% (72/92) | 87% (164/188) |
| example | 100% (20/20) | 78% (72/92) | 87% (164/188) |
| autocashsys | 100% (20/20) | 78% (72/92) | 87% (164/188) |
| config | 100% (1/1) | 100% (4/4) | 100% (11/11) |
| constants | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| controller | 100% (3/3) | 95% (20/21) | 98% (62/63) |
| converter | 100% (1/1) | 100% (1/1) | 100% (13/13) |
| dto | 100% (4/4) | 100% (24/24) | 92% (46/50) |
| entity | 100% (4/4) | 39% (11/28) | 48% (16/33) |
| repository | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| service | 100% (5/5) | 100% (10/10) | 100% (13/13) |
| util | 100% (1/1) | 66% (2/3) | 66% (2/3) |
| AutoCashSysApplication | 100% (1/1) | 0% (0/1) | 50% (1/2) |

Рисунок 3.26 – Покриття коду тестами

Однак важливо підкреслити, що високе покриття коду безпосередньо не означає високу якість тестування. Хоча покриття коду є корисним інструментом для виявлення неперевірених частин вашої кодової бази, воно не гарантує, наскільки добре протестовано різні випадки. Тому це не повинно бути єдиним показником ефективності тесту. На додаток до прагнення до високого охоплення коду, не менш важливо забезпечити широкий спектр тестових сценаріїв для виявлення крайніх випадків і незвичайних взаємодій.

## 3.4 Розробка інтерфейсу клієнтської частини вебзастосунку

### 3.4.1 Розробка структури вебсайту

При проектуванні інтерфейсу були виділені функціональні блоки вебсайту схему навігації між ними, саму структуру діалогу. Що до структури сайту, було використано «зовнішню» структуру, оскільки предметною областю передбачається, що користувач отримує доступ до основного функціоналу сайту з кожної сторінки. Даний підхід значно спрощує користування сайтом в порівнянні з «внутрішньою» структуру сайту.

Також важливим кроком в розробці інтерфейсу користувача є структура вебсайту [45]. Це як інформаційна архітектура, що представляє спосіб організації та представлення вмісту вебсайту користувачам. Добре спланована структура вебсайту може істотно змінити взаємодію з користувачем, пошукову оптимізацію та навіть рівень конверсії. Вибір правильної моделі значною мірою

залежить від характеру вебсайту, розміру та різноманітності вмісту, який він містить, і конкретних цілей, яких він прагне досягти. Таким чином, вибираючи відповідну структуру вебсайту, слід враховувати такі фактори, як навігація користувача, оптимізація SEO та загальний досвід користувача.

Отже, серед існуючих структур вебсайтів була використана «лінійна (послідовна) структура». Представляє з себе схему де з головної сторінки здійснюється перехід на другу сторінку, з неї — на третю і так далі. Найбільш проста і зрозуміла структура, в якій кожна сторінка посилається на попередню (або на Головну) і на наступну. Іншими словами, ідеально підходить для розроблюваної системи, так як в ній, інтерфейс, структура сайту, будується навколо головного бізнес-процесу системи, а саме, «Зробити Покупку». Лінійна модель сайту крок за кроком направляє користувачів у певній послідовності, відповідно до головного бізнес-процесу. Дана модель сайту, як раз і розповсюджена в подібних випадках, коли потрібна певна послідовність, і найчастіше її можна побачити на якихось вебсайтах з оформленням замовлення.

На рисунку 3.27 відображена карту сайту з візуальним відображенням назви вебсторінок розроблювального застосування ІС. В квадратних дужках вказано статуси користувачів системи, що мають доступ до вебсторінок.

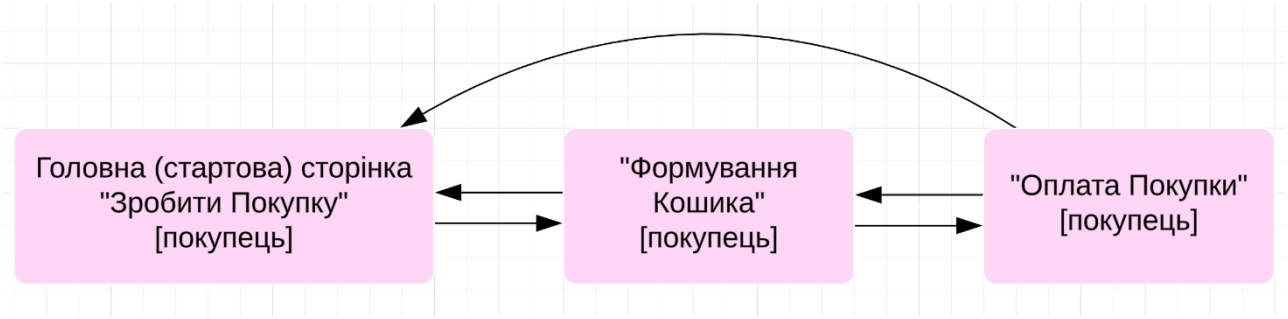



Рисунок 3.27 – Карта вебсайту

В процесі розробки стильових рішень та дизайну програмного за стосунку було враховано всі особливості предметної області, в першу чергу портрети користувачів їх сценарії поведінки, призначення самої інформаційної системи.

Також було розроблено загальні стильові правила, які призначені для того, щоб створити однаковість інтерфейсу і його поведінки. Стильові правила затверджуються як частина вимог до інформаційної системи, а дотримання правил повинно постійно піддаватися перевірці протягом усього етапу розробки

інтерфейсу системи як програмного за стосунку. Стильові правила втілюють в собі кращі напрацювання в дизайні інтерфейсів. Дотримання стильових правил надає ряд таких переваг, як: однаковості між різними екранами / сторінками системи чи скорочення часу розробки та підвищення якості інтерфейсу [38].

Спираючись на особливості предметної області та на найбільш популярні з існуючих стилів інтерфейсу, що поділяються на типи згідно їх класифікаціями за різними ознаками, було обрано декілька основних стилів інтерфейсу та об’єднано в один, а саме: мінімалістичний-корпоративний з відтінками класичного стилю. Так як подібна інформаційна система не передбачається велика кількість користувачів, та що головне, вона не заточена на збільшення клієнтської бази за рахунок якихось особливостей інтерфейсу. В першу чергу опір іде на зручність та зрозумілість інтерфейсу у вирішенні практичних задач поставлених бізнес-логікою інформаційної системи. Тим паче розробка мінімалістичного інтерфейсу зекономить на реалізації самого інтерфейсу, так як потребує порівняно менш кваліфікованих розробників чи дизайнерів.

Що до підбору гармонійних комбінацію кольорів для оформлення інтерфейсу опираючись вищезазначений аналізом було вирішено розбавити сірість та млявість програмного додатку за рахунок кольорів, що є за найпростіших та найдешевшим з способів хоч трішечки оживити інтерфейс. А саме вибір кольору пав на сіро-блакитний та його відтінки. В першу чергу цей колір використовується для заднього фону.

Однак у стилі інтерфейсу фігурують і такі контрастні кольори, як: сирій, зелений, червоний, та жовтий. В основному даними кольорами позначені головні елементи сайту, що дозволяє користувачам краще зосереджуватися на взаємодії з ними, в основному, це якісь кнопки. А такі компоненти, як header (шапки сайту) та foother  (його підвалу) слід виділити по особливому, наприклад використати якийсь темний відтінок, для створення ефекту «рамки», що дозволяє фокусувати увагу користувача на тому що знаходиться в середині — головному бізнес-процесі.

### 3.4.2 Прототипування інтерфейсу

Аналізуючи поставлену задачу, було обрано «еволюційний» вид прототипування (підхід до розробки Інтерфейсів, який представляє з себе

розробку прототипів сторінок застосунку, використовуючи обраний інструментарій), так як цей підхід передбачає послідовне збільшення достовірності вихідного зразка, поки, зрештою, він не стає закінченою системою, що дозволяє в процесі розробки корегувати деякі інтерактивності (тонкощів взаємодії) інтерфейсу чи в загалом сприяє виявленню все більших і більших тонкощів в аспектах дизайну і його вдосконалення. Небезпечним нюансом для даного підходу є деякі труднощі в розширенні при неправильному порядку розроблюваних типів прототипів, тому при даному підході завжди треба починати розробку прототипів низької достовірності та завершувати прототипом високої достовірності, який в більшій мірі буде представляти закінчений продукт [38].

Що до розмірності прототипу, було зроблено більший акцент на розробку глобальних прототипів. А в загалом вибір описаний вище обумовлений тим, що розроблювана система не потребує якихось нових складних ідей чи концепцій інтерфейсу, які слід перевіряти, що є не дуже вигідно для прототипів високої достовірності.

Для підтримки якості розроблюваного інтерфейсу, було виділено ряд критеріїв які в першу чергу дотримувалися при його розробці а саме це [38]:

- надійність;
- супроводжуваність;
- практичність;
- ефективність;
- мобільність;
- функціональність.

Для реалізації прототипів інтерфейсу системи обрано один з найпопулярніших інтерфейсних інструментів Bootstrap (бібліотека HTML, CSS та JS).

Попередньо було проаналізовано, що має бути розміщено у шапці (header) чи підвалі (footer) сайту. Так як структура сайту лінійна, якоїсь окремої навігаційної панелі не передбачається. Єдине, що важливе може бути розміщено у шапці (header) сайту, це форма вибору локалізації (зміни мови інтерфейсу). Відповідно до вимог проекту має бути доступна державна мова, а саме: Українська, та будь яка інша мова міжнародного рівня: Англійська. Відповідна форма зміни мови інтерфейсу зображена на рисунку 3.28.

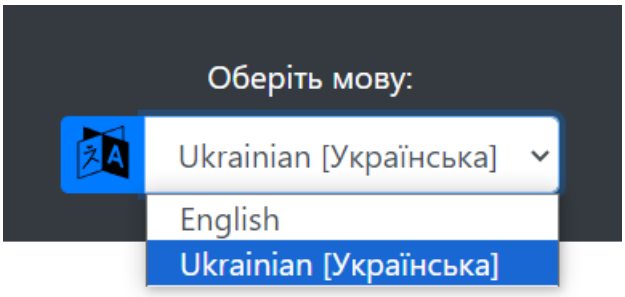



Рисунок 3.28 – Форма зміни мови інтерфейсу

Головна (мейн) сторінка представляє з себе стартову сторінку з єдиною кнопкою «Зробити Покупку», при натисканні на яку розпочинається головний бізнес-процес (рисунок 3.29).

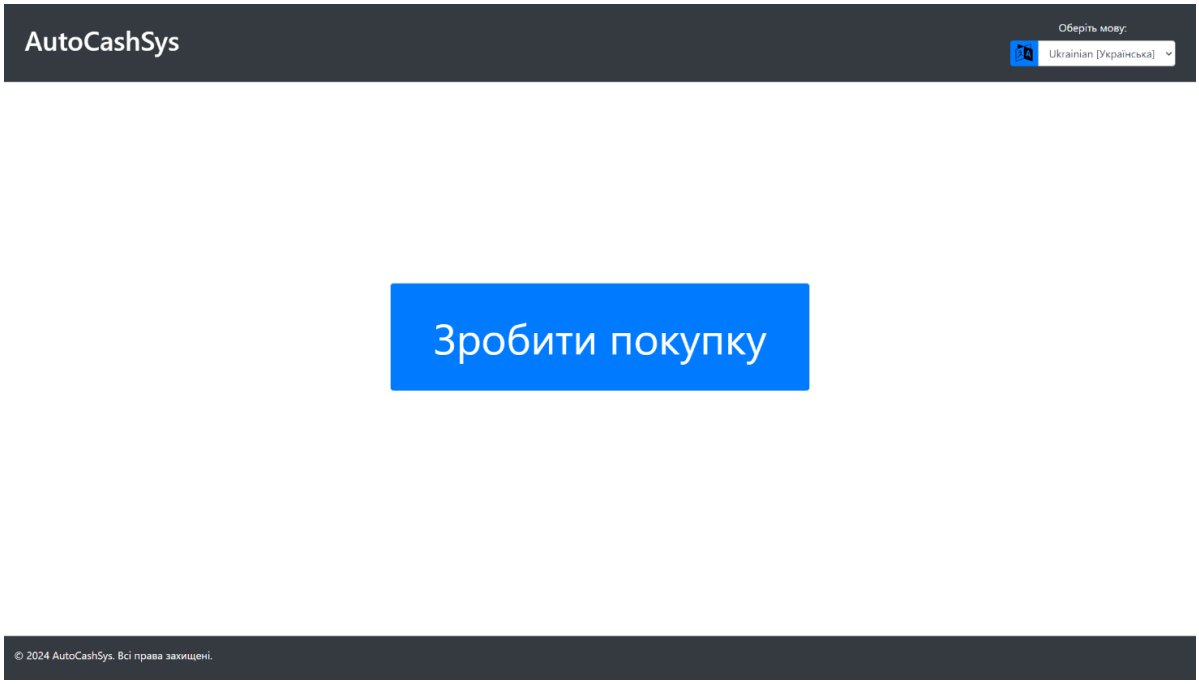



Рисунок 3.29 – Прототип високої деталізації головної (мейн) сторінки

Перший крок головного бізнес-процесу — це наступна сторінка після «стартової», а саме: «Формування Кошика» (рисунок 3.30).

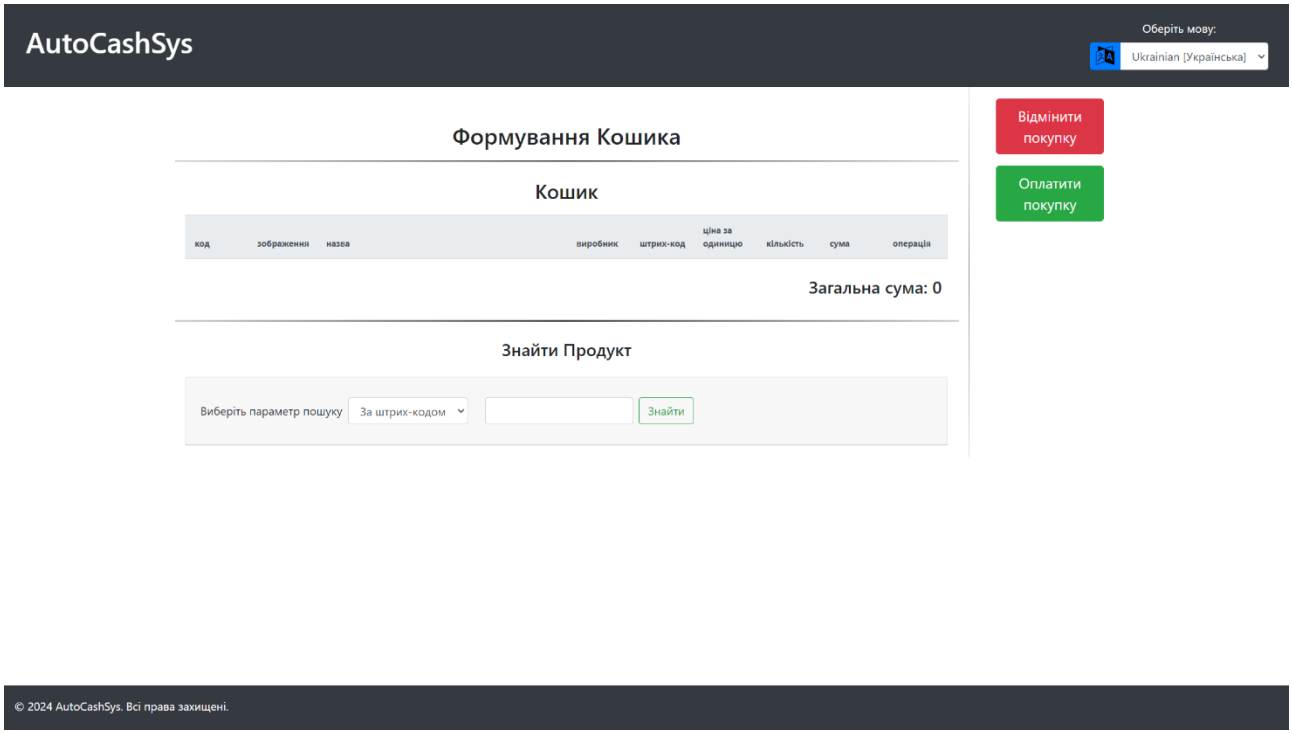



Рисунок 3.30 – Прототип високої деталізації сторінки формування кошика

Це, мабуть, найважчий елемент інтерфейсу даного застосунку, з точки зору користувача. Ця сторінка складається з трьох ключових компонентів, а саме: зона контролю та навігації, пошук товару та сам кошик.

Перший компонент включає в себе кнопки контролю над процесом покупки, а саме, це «Відмінити покупку» та «Оплатити покупку».

Іншим важливим компонентом — є «Знайти Продукт». У разі пошкодження штрих-коду на товарі та неможливості його відскакувати, покупець має можливість знайти товар за певними параметрами, вчасності за назвою товару або його цифровим представленням штрих-коду. Після знаходження відповідного товару, користувач може зразу додати його до свого віртуального кошика у бажаній кількості. Знайдені товари мають відображатися з відповідними полями, щоб покупцю було легше орієнтуватися між отриманими товарами та ідентифікувати бажаний. Це в першу чергу: зображення товару, його назва, виробник та ціна. (рисунок 3.31).

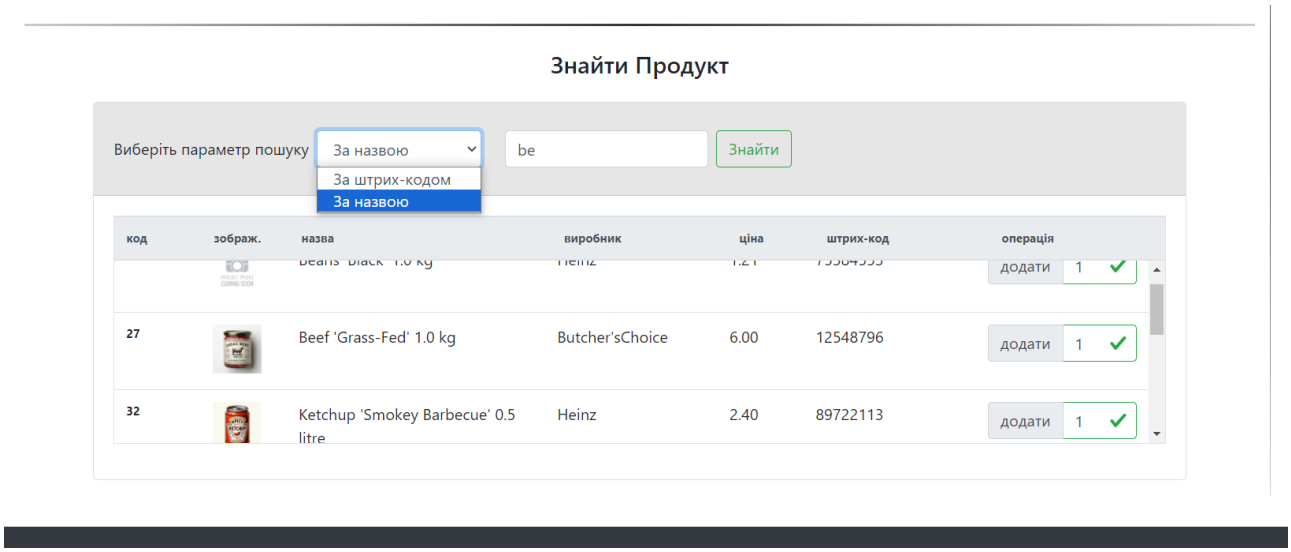



Рисунок 3.31 – Прототип високої деталізації компоненту «Знайти Продукт»

Наступним, важливим компонентом даної сторінки — є сам «Кошик». Це таблиця, що відображає відскановані або додані товари до кошику на даний момент.

Тут також, вкрай важливо чітко відобразити всі товари, з відповідними полями, щоб користувач почувався комфортно, та міг без проблем оцінити об’єм та ціну своєї покупки. Важливо відобразити такі поля, як: зображення товару, його назва, виробник, ціну товару за умовну одиницю, кількість цих товарів у кошику, загальна сума конкретного товару відповідно до його кількості у кошику. Також, обов’язково — це загальна сума за увесь кошик.


Рисунок 3.32 – Прототип високої деталізації компоненту «Кошик»

Також, це можливі операції над товарами у кошику, такі як: редагування, де відбувається зміна кількості у кошику відповідного товару, та видалення, що повністю видаляє обраний товар з кошику. Редагування пропонується робити через діалогове вікно, що має з’являтися при натисканні на відповідну кнопку (рисунок 3.33).

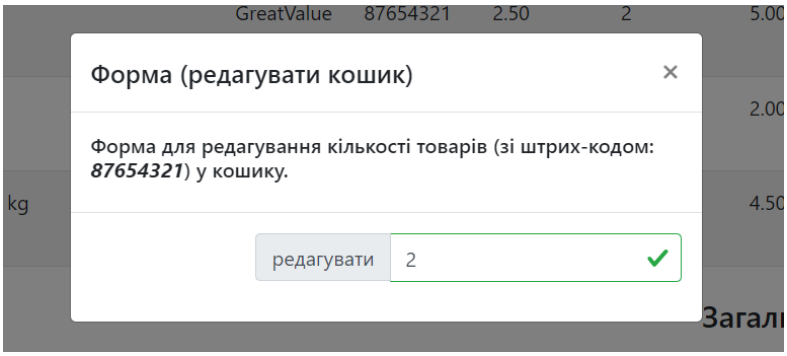



Рисунок 3.33 – Прототип високої деталізації діалогового вікна редагування кошику

Наступим кроком — є оплата покупки. Це відносно проста сторінка, на якій, подібна до попередньої сторінки, є панель навігації. Це кнопка «Повернутися до оформлення кошика» з можливістю повернутися до попереднього кроку, де покупець може додати нові товари до кошика, що він забув, або навпаки, видалити існуючі, наприклад, якщо йому не вистачає коштів. Також, це кнопка «Відмінити покупку» з можливістю повної відміни покупки та повернення до «Стартової» сторінки.

Також дана сторінка має відображати загальну суму покупки, та сповіщати покупця про те, що система очікує оплати через термінал (рисунок 3.34).

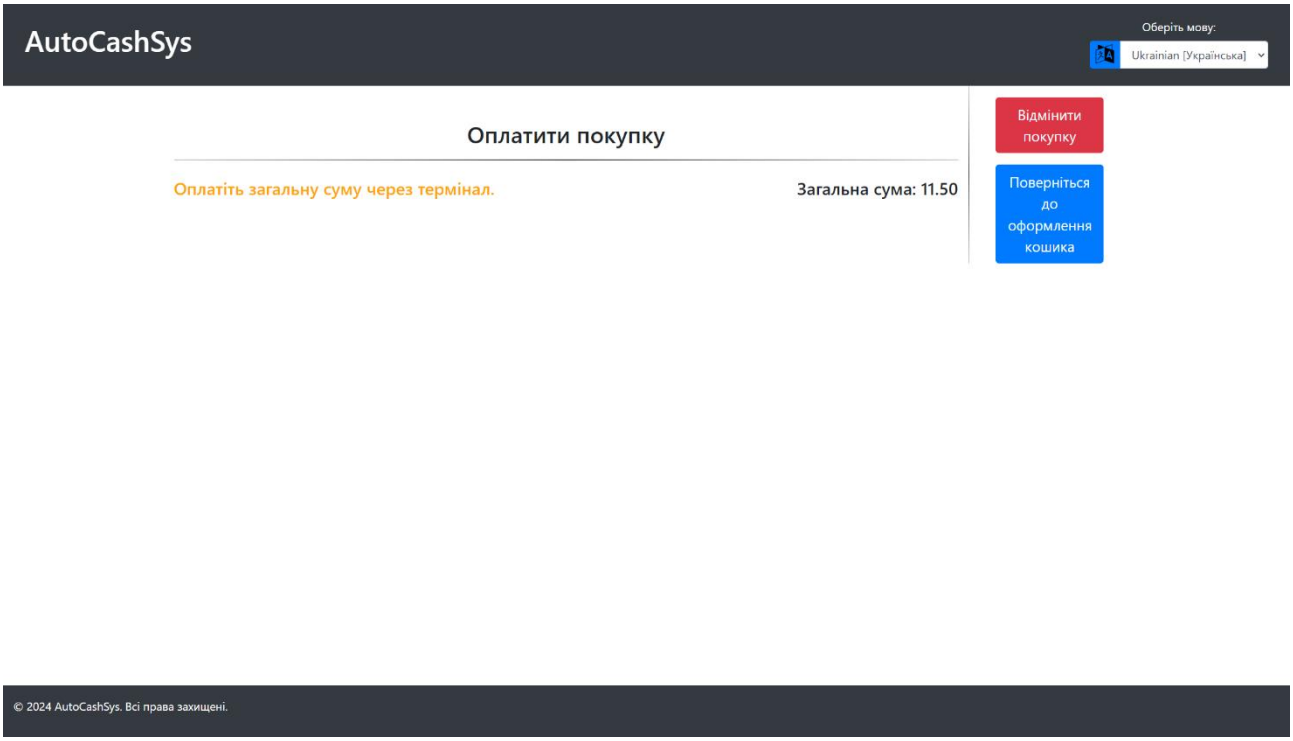



Рисунок 3.34 – Прототип високої деталізації діалогового вікна редагування кошику

### 3.4.3 Тестування інтерфейсу

Тестування інтерфейсу являється досить складним та неоднозначним процесом та не зводиться до прямування строгим та чітким процедурам, тому перед тестуванням UI було розроблено план тестування який включав в себе ряд критеріїв яким в першу чергу підлягався перевірці розроблений інтерфейс, а саме це [38]:

- інтерфейс продукту відповідає прототипам;
- типографіка;
- відповідність стилю;
- адаптивність;
- відповідність стандартам;
- використання функціоналу;
- перевірка орфографії;
- перевірка полів та стандартних елементів;
- елементи інформування.

Аналізуючи (тестуючи) розроблені сценарії поведінки користувача інтерфейсного застосунку, найбільшого сумніву коректності підлягають сценарій формування кошика. Тому відповідно до виявлених недоліків були розроблені рекомендації що до виправлення помилок інтерфейсу у прототипі, а саме:

При скануванні товарів, було за рекомендовано додати спеціальне вікно для відображення області що на даний момент сканується сканером. Це має допомогти користувачу наводити сканер на штрих-код, та пришвидшити процес сканування товарів (рисунок 3.35).

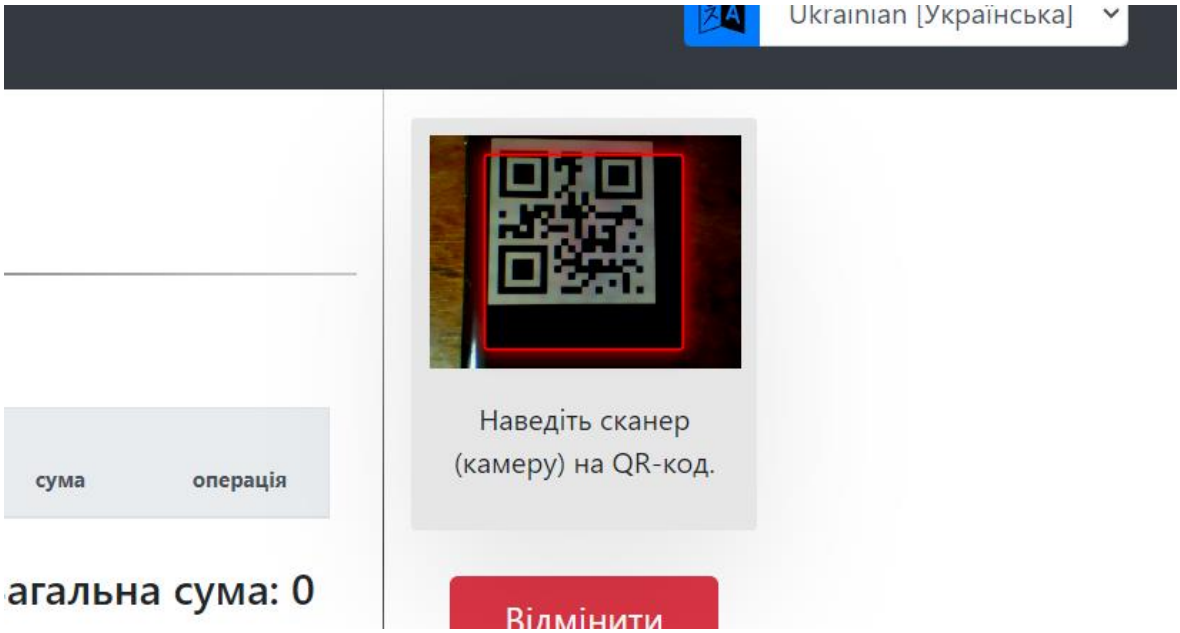



Рисунок 3.35 – Вікно з відображенням скандувальної області

Також в результаті проведеного тестування були виявлені деякі недоліки в адаптивності, а саме при розширенні екрану менше ніж 1360 х 768, некоректно відображалося вікно пошуку товарів на сторінці «Формування кошику». Після виправлення даного недоліку було повторно проведено тестування адаптивності UI на екранах різних розмірів та на різних орієнтаціях (portrait, landscape).

Також були виявлені деякі недоліки в топографічному плані, було виявлено некоректне виведення занадто довгих назв продуктів при їх відображенні у кошику. Дану проблему було вирішено заміною довгого остатку трьома точками при досягненні назви певної кількості символів (рисунок 3.36).


Рисунок 3.36 – Вирішення недоліку з довгими назвами

Також, для підвищення лояльності користувачів системи, на кожне з безповоротний дій, наприклад таких як: відмінити покупку, або видалити товар з кошику, було додано спеціальні діалогові вікна з підтвердженням операції (рисунки 3.37 – 3.38).

Також було додано певної динаміки до інтерфейсу, а саме відстеження поточного стану процесу. Це дозволяє унеможливити деякі дії, які в протилежному випадку могли б легко заплутати покупця, або ж привести процес до не консистентного вигляду. В першу чергу це унеможливлення переходу до етапу оплати покупки поки кошик пустий або загальна сума покупки дорівнює нулю (рисунки 3.39). Також, це унеможливлення таких дій як: відміна покупки та перехід до формування кошика, вже після успішної оплати покупки та її подальшої обробки (рисунки 3.40).

Це має унеможливити випадкові та небажані дії від покупця та підвищити ясність та усвідомлення послідовності дій в операції, що в свою чергу, має пришвидшити адаптування та навчання покупця новій технології та в загалом покращити користувацький досвід.

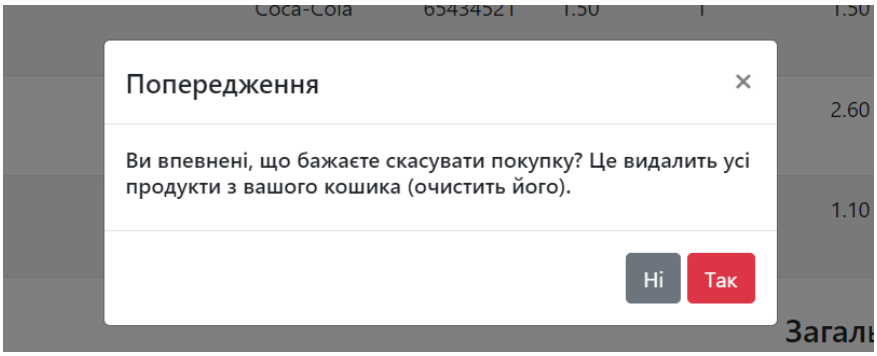



Рисунок 3.37 – Діалогове вікно підтвердження скасування покупки

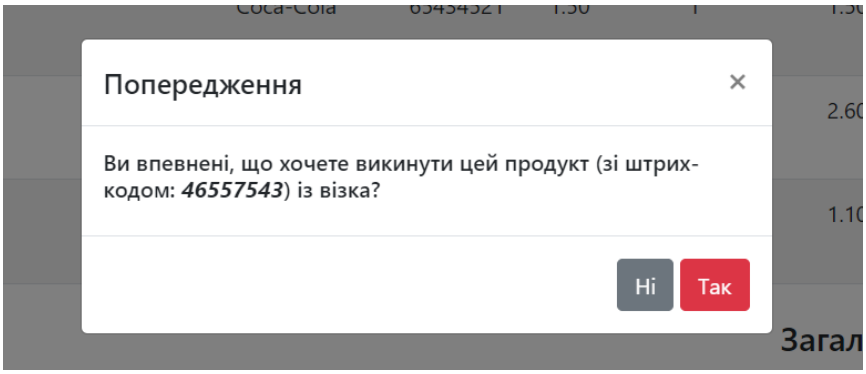



Рисунок 3.38 – Діалогове вікно підтвердження видалення товару з кошика

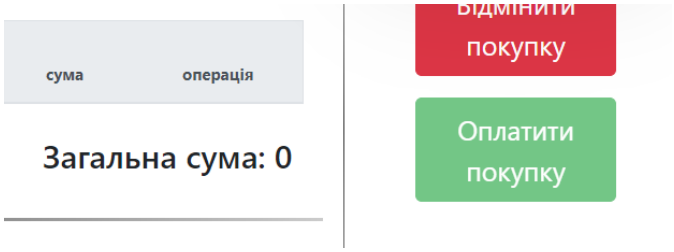



Рисунок 3.39 – Кнопка «Оплатити покупку» не активна

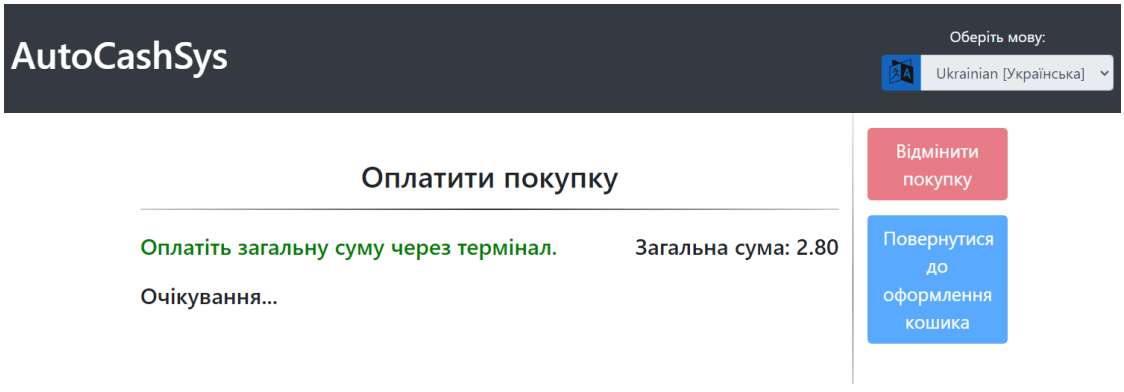



Рисунок 3.40 – Кнопки «Відмінити покупку» та «Повернутися до оформлення кошика» не активні

Також, не менш важливим є обробка та сповіщення користувача о можливих помилках. Це один з ключових аспектів забезпечення якісної взаємодії з будь-яким програмним забезпеченням, будь то вебсайт, мобільний додаток або настільний додаток. Зворотній зв’язок із користувачем, у разі виникнення помилки, дуже важливий. Це допомагає користувачу зрозуміти, що сталося, чому це сталося, і, якщо можливо, що можна зробити, щоб це виправити. Важливо, щоб повідомлення про помилки передавались мовою, яку користувач може зрозуміти, уникаючи технічного жаргону.

Програма без комплексної стратегії обробки помилок ніколи не зможе запропонувати безперебійну роботу системи та взаємодії користувача з нею. Це може легко розчарувати користувача та негативно вплинути на його загальне сприйняття системи.

На рисунку 3.41 зображено обробку та сповіщення покупця про помилку, що виникла при спробі оплатити покупку.

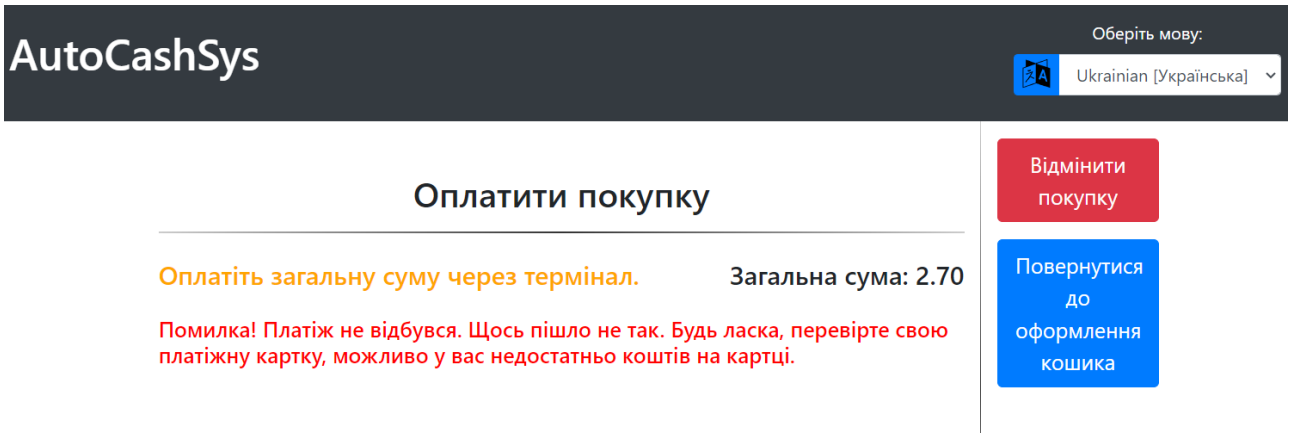



Рисунок 3.41 – Помилка при оплаті

# ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було спроектовано архітектуру системи касового самообслуговування та розроблено вебсервіс відповідальна за процес оформлення покупки та її облік.

Було розглянуто ряд важливих питань при системному проектуванні, наприклад таких як, важливість перед проектного аналізу системи, її класифікація серед існуючих систем, відповідно вибір правильної архітектури та стеку технологій. Це може значно прискорити саму розробку, так і в подальшому зберегти потрібну архітектуру, масштабованість, продуктивність, безпеку, розширюваність, надійність та багато чого іншого. А ігнорування цих важливих етапів проектування можуть обійтися розробнику дуже дорого, довго та ризиковано.

Розробка була розділена на декілька основних частин, а саме, в першу чергу, це вибір архітектури розроблюваного застосунку, проектування інфраструктури системи, вчасності з використання хмарних технологій Amazon Web Services (AWS) провайдера та вибір стеку технологій. Для реалізації клієнтської частини обрано доволі типові технології для веб розробки, а саме HTML, CSS та JS на додаток з бібліотекою Bootstrap. Даний інструментарій дозволив розробити інтерфейс вебсайту відповідно до поставлених критеріїв якості. Для реалізації механізму зберігання даних було обрано СУБД MySQL, яка відповідає високим стандартам та пропонує багатий набір функціональних можливостей. Щодо основної бізнес-логіки на стороні серверу, було обрано Spring Framework. Spring є одним з найпопулярніших фреймворків, що виділяється своєю гнучкістю, масштабованістю та широким набором функціональних можливостей. Головна перевага Spring Framework полягає в наявності великої кількості готових рішень, які потребують лише певних конфігурацій, що відповідно прискорює та здешевлює розробку.

В рамках дослідження розглянуто декілька способів зберігання даних при розгортанні розроблюваної системи в AWS середовищі. Було розглянуто три способи, а саме: з використанням тільки AWS S3 сервісу, з використанням AWS RDS сервісу, та з використання AWS DynamoDB та S3. З розглянутих, останній виявився найбільш привабливим, так як гарантував автоматичну

масштабованість, швидкий пошук та дуже гарне ціноутворення, особливо у довгостроковій перспективі.

В майбутньому розроблена система може бути розширена за рахунок інтеграції з іншими пов’язаними системами. Як було розглянуто в роботі, це може бути система звичайного розрахунково-касового обслуговування [39] чи якась система аналітики, що на базі зафіксованих покупок прогнозує зменшення кількості певного товару на складі та відповідно замовляє нові поставки до потрібних складів. Подібна автономна система прихильна до легкого масштабування може допомогти будь якому роздрібному підприємству перейти на глобальний рівень.

Результати роботи доповідалися на 28-му Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті», м. Харків [1].

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Олексій К. Системне проектування касової системи самообслуговування. *Радіоелектроніка та молодь у XXI столітті*, м. Харків, 15 квіт. 2024 р.

2. Technology challenges of global systems, managing global software development.
URL: https://web.archive.org/web/20240519064318/https://paginas.fe.up.pt/~acbrito/laudon/ch16/chpt16-4main.htm (дата звернення: 19.05.2024).

3. Rodgers E. Grocery store statistics: where, when, & how much people grocery shop.
URL: https://web.archive.org/web/20240531172053/https://www.driveresearch.com/market-research-company-blog/grocery-store-statistics-where-when-how-much-people-grocery-shop/ (дата звернення: 31.05.2024).

4. International journal of business administration.
URL: https://repositorio.ufmg.br/bitstream/1843/45009/2/E-commerce%20a%20short%20history%20follow-up%20on%20possible%20trends.pdf (дата звернення: 04.05.2024).

5. H. Kis, Г. Р. Кісь та Г. Р. Кись, «The use of B2B and B2C e-commerce models as a tool of anti-crisis management of industrial enterprises,» Поліський національний університет, 2020.

6. P. Minkyu Lee, «StarUML documentation v6,» MKLabs co., Ltd., 2023. URL: https://docs.staruml.io/. (дата звернення: 28.04.2024).

7. а. С.В. Козир, д.-р. т. н. п. В.В. Слєсарєв, к. ф.-м. н. д. С.А. Ус та в. О.В. Хазова, «Modeling processes in IDEF0 (Integration Definition for Function Modeling),» Ministry of Education and Science of Ukraine National Technical University “Dnipro Polytechnik”, 2019. URL: https://sau.nmu.org.ua/ua/osvita/metod/Modeling%20processes%20in%20IDEF0.pdf. (вата звернення: 13.05.2024).

8. В. Р. України, «Про застосування реєстраторів розрахункових операцій у сфері торгівлі, громадського харчування та послуг,» Законодавство України, 31.12.2023. URL: https://zakon.rada.gov.ua/laws/show/265/95-%D0%B2%D1%80#Text. (дата звернення: 17.04.2024).

9. В. Р. України, «Про захист прав споживачів,» Законодавство України, 19.11.2022. URL: https://zakon.rada.gov.ua/laws/show/1023-12#Text. (дата звернення: 17.04.2024).

10. В. Р. України, «Про захист персональних даних,» Законодавство України, 27.04.2024. URL: https://zakon.rada.gov.ua/laws/show/2297-17#Text. (дата звернення: 14.04.2024).

11. JavaRush, «Клієнт-серверна архітектура,» URL: https://javarush.com/ua/quests/lectures/ua.questservlets.level14.lecture00. (дата звернення: 01.06.2024).

12. KIIT_Polytechnic, «Introduction to Internet of Things,» URL: https://kp.kiit.ac.in/pdf_files/06/6th-Sem_CSE_Internet-of-Things_SM.pdf. (дата звернення: 01.06.2024).

13. A. H. C, «The Power of Microservice Architecture,» Linkedin, 31 1 2024. URL: https://www.linkedin.com/pulse/power-microservice-architecture-akash-h-c-vfihc/. (дата звернення: 02.06.2024).

14. L. M. A. Hernández, «REST architecture for enterprise web application development,» Tecnológico Nacional de México/Instituto Tecnológico de Villahermosa, 3 7 2020. URL: https://iydt.wordpress.com/wp-content/uploads/2020/07/3_11_rest-representational-state-transfer-architecture-for-enterprise-web-application-development.pdf. (дата звернення: 14.04.2024).

15. F. Hinsenkamp, «What is a Message Broker?,» Big Tech Coach, 23 5 2024. URL: https://www.bigtech.coach/system-components/what-is-a-message-broker. (дата звернення: 28.05.2024).

16. I. Douglas, «Can gRPC replace REST and WebSockets for Web Application Communication?,» Google, 4 12 2023. URL: https://grpc.io/blog/postman-grpcweb/. (дата звернення: 28.05.2024).

17. J. Campbell, «Kubernetes vs Docker,» Atlassian, URL: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker. (дата звернення: 02.06.2024).

18. «Microsoft Azure,» Microsoft, URL: https://azure.microsoft.com/en-us/. (дата звернення: 03.06.2024).

19. «Google Cloud Platform,» Google, URL: https://cloud.google.com/. (дата звернення: 03.06.2024).

20. «Amazon Web Services (AWS),» Amazon, URL: https://aws.amazon.com/. (дата звернення: 03.06.2024).

21. L. Voss, «AWS: The Good, the Bad and the Ugly,» 18 12 2012. URL: https://web.archive.org/web/20160429075023/http://blog.awe.sm/2012/12/18/aws-the-good-the-bad-and-the-ugly/. (дата звернення: 03.06.2024).

22. «AWS API Gateway,» Amazon, URL: https://aws.amazon.com/api-gateway/. (дата звернення: 05.06.2024).

23. «AWS EC2,» Amazon, URL: https://aws.amazon.com/ec2/. (дата звернення: 05.06.2024).

24. «AWS S3,» Amazon, URL: https://aws.amazon.com/s3/. (дата звернення: 06.06.2024).

25. «AWS Relational Database Service,» Amazon, URL: https://aws.amazon.com/rds/. [(дата звернення: 06.06.2024).

26. «AWS Virtual Private Cloud,» Amazon , URL: https://aws.amazon.com/vpc/. (дата звернення: 06.06.2024).

27. «AWS Pricing Calculator,» Amazon, URL: https://calculator.aws/#/. (дата звернення: 07.06.2024).

28. «Модель-Вид-Контролер,» Wikipedia, URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. (дата звернення: 07.05.2024).

29. D. Tymoshchenko, «Most Popular Backend Frameworks in 2024,» Acropolium, 30 8 2023. URL: https://acropolium.com/blog/most-popular-backend-frameworks-in-2021-2022-pros-and-cons-what-to-choose/. [Дата звернення: 14 5 2024].

30. Tanzu, «Why Spring?,» VMware, 2024. URL: https://spring.io/why-spring. (дата звернення: 27.06.2024).

31. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith та G. Bierman, «The Java Language Specification (Java SE 21 Edition),» Oracle, 23 8 2023. URL: https://docs.oracle.com/javase/specs/jls/se21/html/index.html. (дата звернення: 25.06.2024).

32. MDN_Docs_Web, «HTML: HyperText Markup Language,» Mozilla, URL: https://developer.mozilla.org/en-US/docs/Web/HTML. (дата звернення: 25.06.2024).

33. T. A. Jr., E. J. E. /. fantasai, F. Rivoal та C. Lilley, «CSS specification,» W3C Group Note, 7 12 2023. URL: https://www.w3.org/TR/css-2023/. (дата звернення: 25.06.2024).

34. S.-y. Guo, M. Ficarra та K. Gibbons, «ECMAScript 2025 Language Specification,» Draft ECMA-262, 6 6 2024. URL: https://tc39.es/ecma262/. (дата звернення: 25.06.2024).

35. «Get started with Bootstrap,» Bootstrap, URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/. (дата звернення: 25.06.2024).

36. «Why MySQL?,» Oracle, 2024. URL: https://www.mysql.com/why-mysql/. (дата звернення: 25.06.2024).

37. О. Кущ, «Розробка компонентів інформаційної системи розрахунково-касового обслуговування,» *Харківський Національний Університет Радіоелектроніки,* с. 34, 41, 45-48, 54, 2022.

38. О. Кущ, «Розробка компонентів інформаційної системи розрахунково-касового обслуговування,» в *Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління*, Харків, 2022.

39. «MySQL Workbench,» Oracle, 2024. URL: https://www.mysql.com/products/workbench/. (дата звернення: 25.06.2024).

40. M. Ozkaya, «Layered (N-Layer) Architecture,» Medium, 6 9 2021. URL: https://medium.com/design-microservices-architecture-with-patterns/layered-n-layer-architecture-e15ffdb7fa42. (дата звернення: 25.06.2024).

41. S. Morenets, «Навіщо використовують DTO. Приклади в Java-застосунках,» DOU, 12 8 2021. URL: https://dou.ua/forums/topic/34411/. (дата звернення: 25.06.2024).

42. E. Baeldung, «An Introduction to CDI (Contexts and Dependency Injection) in Java,» Baeldung, 8 1 2024. URL: https://www.baeldung.com/java-ee-cdi. (дата звернення: 25.06.2024).

43. «Tasty mocking framework for unit tests in Java,» Mockito, URL: https://site.mockito.org/. (дата звернення: 25.06.2024).

44. S. Wells, «Website structure A to Z,» Slickplan, 10 1 2024. URL: https://slickplan.com/blog/types-of-website-structure. (дата звернення: 02.06.2024).