Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інтелектуальних інформаційних технологій та автоматики
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Бакалаврська дипломна робота на тему:

РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ПРОДАЖУ НАСТІЛЬНИХ ІГОР

Виконав: студент 4 курсу, групи 3КН-20б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки,спеціальності)
Миронюк Д. А.
(прізвище та ініціали)

Керівник: к.т.н., доцент кафедри КН
Денисюк В. О.
(прізвище та ініціали)

«07» 06 2024 р.

Рецензент: к.т.н., доцент кафедри САІТ
Козачко О. М.
(прізвище та ініціали)

«07» 06 2024 р.

Допущено до захисту
Зав. кафедри
«07» 06 2024 р.

Вінниця ВНТУ – 2024 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший(бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН проф.,
д.т.н. Яровий А.А.
«07» 03 2024р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТА
Миронюка Дмитра Анатолійовича

1. Тема роботи: Розробка програмного модуля для продажу настільних ігор

2. Керівник роботи: Денисюк Валерій Олександрович, доцент.
затверджені наказом вищого навчального закладу «11» 03 2024 року №90

3. Термін подання студентом роботи 27.05.2024

4. Вихідні дані до роботи:

Мова програмування – C#;
Розробка на базі ASP.NET фреймворку;
Інтерфейс користувача має бути інтуїтивно зрозумілий.

5. Зміст текстової частини (перелік питань, які потрібно розробити):

Вступ; аналіз предметної області настільних ігор;розробка програмного модуля для продажу настільних ігор; програмна реалізація модуля для продажу настільних ігор.

6. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень): ER-модель програмного модулю для продажу настільних ігор; UML діаграма класів програмного модуля для продажу настільних ігор; діаграма прецедентів для гостя та зареєстрованого користувача; діаграма прецедентів для постачальника; UML діаграма класів-репозиторіїв шару доступу до даних UML; діаграма послідовності фільтрації настільних ігор.

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	Завданн прийня
1-3	Денисюк В. О., доцент каф. КН		

8. Дата видачі завдання 07.03.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Приміт ка
		початок	закінчення	
1	Вступ. Аналіз предметної області настільних ігор.	06.05.24	12.05.24.	
2	Розробка програмного модуля для продажу настільних ігор.	13.05.24	19.05.24	
3	Програмна реалізація модуля для продажу настільних ігор.	20.05.24	26.05.24	
4	Розробка інструкції користувача.	27.05.24	29.05.24	
5	Висновки та аналіз отриманих результатів	30.05.24	31.05.24	
6	Оформлення матеріалів до захисту БДР	01.06.24	06.06.24	

Студент _____ **Миронюк Д.**
(підпис) (прізвище та ініціа

Керівник роботи _____ **Денисюк В.**
(підпис) (прізвище та ініціа

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 107 сторінок формату А4, на яких є 69 рисунків, 2 таблиці, список використаних джерел містить 31 найменування.

Бакалаврська дипломна робота присвячена створенню програмного модуля для продажу настільних ігор. В даній бакалаврській роботі проаналізовано етапи процесу розробки веб-додатків, наведено переваги і недоліки програм аналогів. Досліджено архітектури, що використовуються для створення програмних засобів, спроектовано компоненти програмного модулю, створено та протестовано програмний модуль для продажу настільних ігор. Програмний продукт розроблений в інтегрованому середовищі Visual Studio Community за допомогою мов програмування C#, HTML, CSS та фреймворків ASP.NET Core, Entity Framework Core, Blazor.

Ключові слова : ASP.NET Core, інтернет-магазин, настільні ігри.

ABSTRACT

The bachelor thesis consists of 107 pages of A4 format, on which there are 69 figures, 2 tables, the list of used sources contains 31 names.

The bachelor thesis is devoted to the creation of a software module for the sale of board games. In this bachelor's work, the stages of the web application development process are analyzed, the advantages and disadvantages of similar programs are given. The architectures used to create software tools were studied, the components of the software module were designed, and the software module for selling board games was created and tested.

The software product is developed in the integrated environment of Visual Studio Community using C#, HTML, CSS programming languages and ASP.NET Core, Entity Framework Core, Blazor frameworks.

Keywords: ASP.NET Core, online store, board games.

ЗМІСТ

ВСТУП

Актуальність теми.

Розробка програмного модулю для продажу настільних ігор є дуже актуальною, оскільки існує багато користувачів, яким цікава тема настільних ігор в цілому. Часто саме цим користувачам потрібна можливість бистро та зручно ознайомитись з великим набором настільних ігор та в деталях можливість переглянути характеристики окремої гри. Веб-додаток може бути найкращим способом надання таких можливостей, так як не потребує додаткового програмного забезпечення від користувача та надає актуальну інформацію.Дослідження допоможе створити ефективні інструменти, які полегшать користувачам вибір настільних ігор з урахуванням індивідуальних переваг користувача та надаватимуть деталі про окрему гру.

Мета дослідження.

Метою дослідження є розширення функціональних можливостей та інформативності програмних модулів для продажу настільних ігор.

Задачі дослідження

Для досягнення поставлених цілей необхідно подальше вирішення наступних завдань дослідження:

- Огляд і аналіз відомих аналогів;
- Проектування компонентів програмного модулю;
- Реалізація компонентів програмного модулю;
- Перевірка роботи і взаємодії компонентів.

Об’єктом дослідження є процес створення веб-додатку з продажу настільних ігор.

Предметом дослідження є алгоритм та програмне забезпечення для продажу настільних ігор.

Практичною цінністю одержаних результатів є те, що розробники можуть використати готову серверну частину з відкритим інтерфейсом для своїх програм.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ НАСТІЛЬНИХ ІГОР

1.1 Аналіз розробки інтернет-магазину

Створення веб-додатків включає кілька етапів, що вимагають знань інтернет технологій та програмування. Різні джерела по різному описують життєвий цикл веб розробки, отже узагальнивши цю інформацію й орієнтуючись саме на обрано тему дипломної роботи виділимо й опишемо такі етапи[1]:

1. Збір вимог та аналіз
2. Планування
3. Дизайн
4. Розробка фронтенду
5. Розробка бекенду
6. Створення бази даних
7. Інтеграція
8. Тестування й контроль якості
9. Розгортання
10. Підтримка та розвиток

Етап збору вимог за аналізу визначення мети та обсягу проекту, а також збір вимог до додатку. Визначається, які функції повинні бути реалізовані. Для визначення завдань, контролю строків і спілкування в команді можна виділити такі засоби управління проектами як, наприклад, Jira, Trello або Asana.

На етапі планування визначаються необхідні ресурси для реалізації проекту. Відповідно до вимог та потрібного функціоналу визначається архітектура додатку та складається набір технологій, що будуть використані для реалізації додатку. Можуть використовуватись програмні засоби для складання схем, такі як draw.io або Lucidchart, для створення діаграм архітектури, прототипів та інших візуальних зображень.

На етапі дизайну визначається зовнішній вигляд і взаємодія з

користувачем. Веб-дизайнер створює макети, включаючи графіку, кольорову схему, розташування елементів і стиль оформлення. Для розробки дизайну можуть використовуватись як і спеціально розроблені для веб дизайну 10 редактори такі як Sketch, Adobe Experience Design чи Figma, так і універсальні редактори по типу Adobe Photoshop. Спеціалізовані інструменти для розробки інтерфейсу користувача дають можливість створення інтерактивних прототипів та тестування взаємодії з користувачем.

Фронтенд веб-додатка відповідає за те, як додаток відображається і взаємодіє з користувачем у браузері. Використовуючи HTML для створення структури веб-сторінок. CSS для оформлення сторінок, включаючи кольори, шрифти, розташування елементів і анімацію, використовуючи фреймворки наприклад Bootstrap чи Tailwind. JavaScript та фреймворки, такі як React, Angular або Vue.js, для реалізації динамічного функціоналу, взаємодії з користувачем та комунікації з сервером.

Бекенд відповідає за обробку запитів користувача, збереження та отримання даних з бази даних і надання відповідей. Розробники використовують мови програмування, такі як Python, Java, C#, та фреймворки (Asp.Net Core, Django, Rails), щоб створити логіку додатку, підключити або створити базу даних, реалізувати серверну логіку, маршрутизацію та обробку запитів.

Багато веб-додатків потребують зберігання даних, таких як інформація про користувачів, продукти або динамічний контент. Для цього використовують реляційні або нереляційні бази даних. Реляційні бази даних, такі як MySQL, PostgreSQL або Microsoft SQL Server, для зберігання структурованих даних. Нереляційні бази даних, такі як MongoDB або Firebase, для зберігання гнучкої та нереляційної інформації. Розробники створюють схему бази даних і реалізують логіку зберігання, звертання та оновлення даних.

Для зручності користування веб-додатком використовують сторонні сервіси, апі та бібліотеки, що розширяють функціональність додатка,

наприклад платіжні шлюзи, служби геолокації, взаємодія з поштовими компаніями. Для їх використання потрібно провести етап інтеграції для коректної роботи фронтенда та бекенда з сторонніми ресурсами.

На етапі тестування та контролю якості веб-додаток піддається ретельному тестуванню, щоб виявити та виправити помилки та недоліки. Тестування може включати атомарні тести, інтеграційні тести та випробування в реальних умовах. Забезпечення якості грає важливу роль у створенні надійних та ефективних веб-додатків. Для цього можуть використовуватись такі інструменти, як Selenium або Cypress, для автоматизації тестування веб-додатків.

Після успішного тестування веб-додаток готовий до розгортання. Це означає розміщення додатку на веб-сервері та налаштування необхідних середовищ виконання. Розгортання може включати такі кроки, як налаштування веб-сервера, встановлення необхідних залежностей і підключення до бази даних. Для розміщення веб-додатків і обробки вхідних запитів можуть використовуватись веб-сервери, такі як Apache або Nginx. Для розгортання та масштабування можна використовувати хмарні платформи, такі як AWS, Microsoft Azure або Google Cloud.

Після розгортання веб-додатку зазвичай необхідна підтримка і обслуговування. Це може включати виправлення помилок, оновлення функцій, оптимізацію продуктивності і відповідь на вимоги користувачів. Веб-додаток може постійно розвиватися та розширюватись з врахуванням нових потреб і технологічних змін.

1.2 Аналіз програм аналогів інтернет-магазину настільних ігор

Аналіз інтернет-магазинів для продажу настільних ігор дозволяє оцінити їх дизайн, користувацький досвід, функціональність та конкурентні переваги. На українському ринку доступно багато веб-додатків, але кожен з них має свої особливості, а також переваги і недоліки. Тому проведемо аналіз наявних онлайн магазинів з продажу:

Rozetka.ua - це один з найпопулярніших українських онлайн-ретейлерів[2], що пропонує широкий асортимент товарів у різних категоріях, включаючи електроніку, побутову техніку, комп'ютери, меблі, одяг, косметику та багато іншого. На головній сторінці Rozetka.ua користувачі можуть знайти різні акційні пропозиції, новинки, бестселери та рекомендації товарів(див. рисунок 1.1). Сайт має зручний пошуковий двигун, який дозволяє швидко знайти потрібний товар за назвою, брендом або категорією. Кожен товар має опис, фотографії, відгуки покупців та іншу інформацію, що допомагає зробити обдуманий вибір.

Однак, через його все направленість є незручним для пошуку та покупки настільних ігор, так як має малу кількість фільтрів для підбору товару. Також надто мало інформації надає про окрему гру. Також категорії “настільні ігри” неможливо найти без відповідного запишу в пошукові, так як таку категорію товарів не включили в меню вибору категорій.

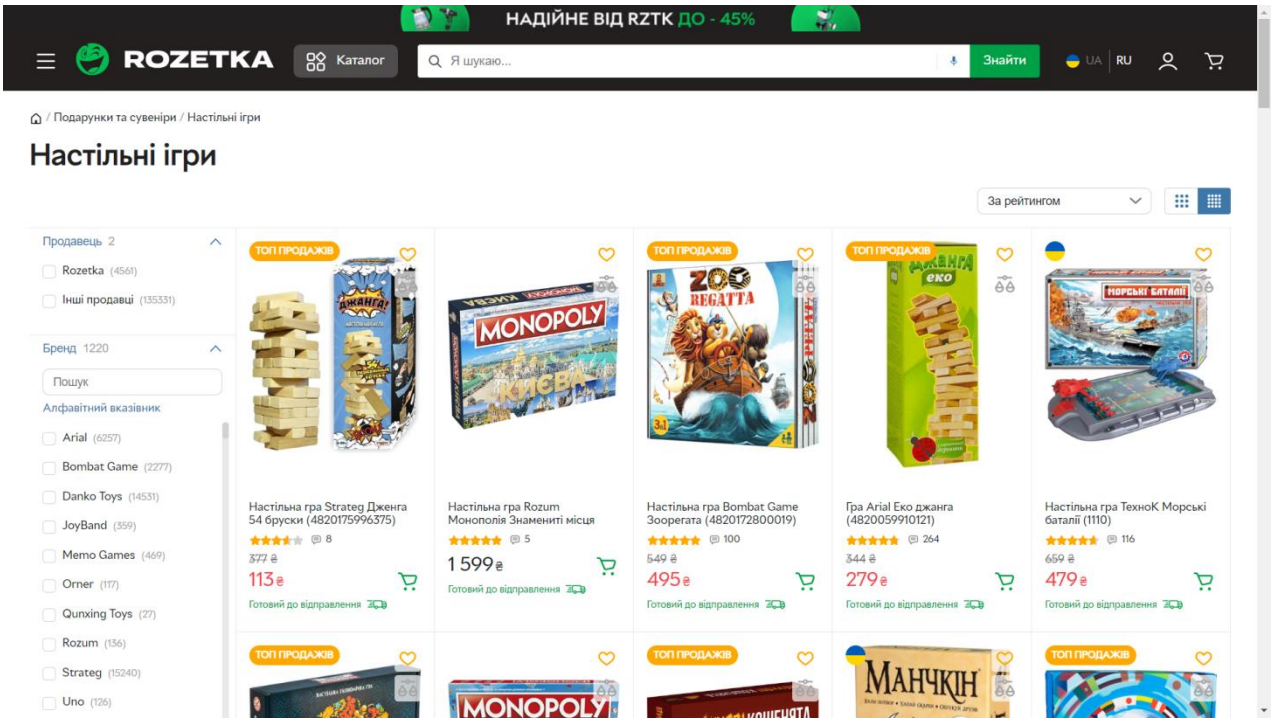

Рисунок 1.1 – Каталог настільних ігор інтернет-магазину Rozetka

Дім Ігор - відомий магазин по продажу настільних ігор. Цей магазин пропонує незвичним асортиментом настільних ігор та аксесуарів до них. Формування асортименту відбувається на основі індивідуального досвіду та

потреб покупців і працівників магазину[3]. Також дім ігор відомий своєю зручною платформою для замовлення товарів та швидкою доставкою. Недоліком є неповнота інформації щодо настільних ігор, яка наводиться в описі продукту, за більшою інформацією вказується посилання на сторонній ресурс(рис. 1.2).

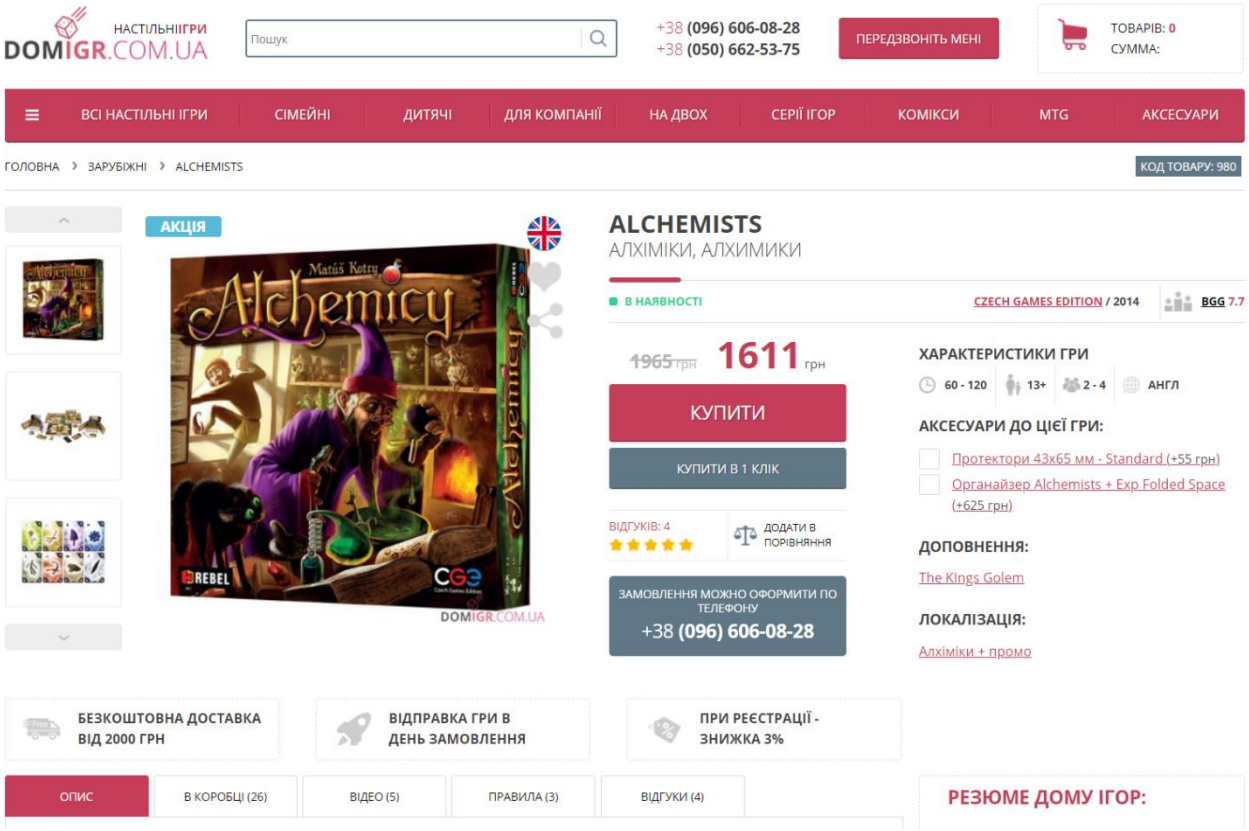

Рисунок 1.2 – Сторінка настільної гри у магазині Дім Ігор

Ігромаг - один з найвідоміших українських видавництв настільних ігор та інтернет-магазин по продажу ігор свого видавництва та інших видавництв. Має широкий асортимент товару, що постійно поповнюється за рахунок співпраці з іноземними видавництвами. Асортимент складається з цікавих ігор, які побудовані не лише на удачі, а й змушують думати логічно і стратегічно[4]. В даному магазині настільних ігор замовлення обробляються цілодобово, що дозволяє дуже швидко здійснювати покупки. Якщо виникли запитання - спеціалісти магазину готові надати професійну допомогу з вибором, який дозволить визначитись і придбати ігри настільні. Недоліком є надмірність в пошукових фільтрах і незручний їх вибір(рис. 1.3).

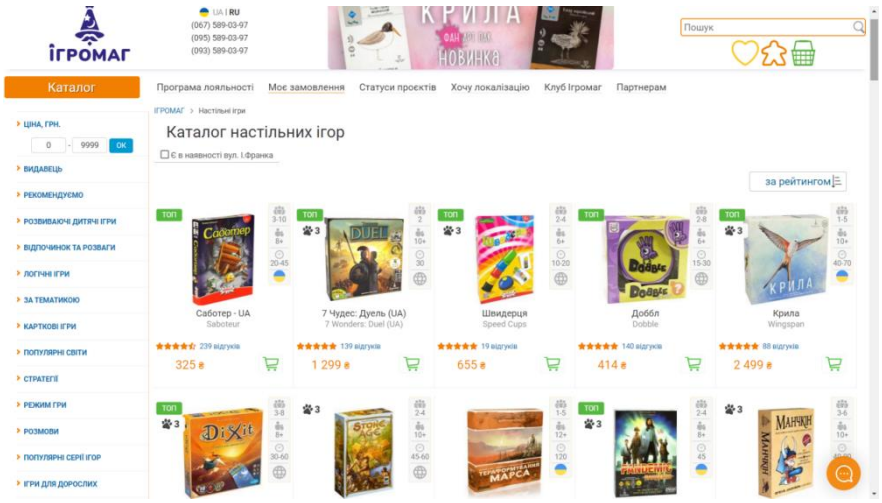

Рисунок 1.3 – Каталог настільних ігор інтернет-магазину Ігромаг

Гігач(Geekach) - це магазин, який пропонує найширший асортимент настільних ігор. У каталозі - кращі ігри від українських виробників, хітові ігри на англійській мові й польські видання мовнонезалежних ігор, які коштують відносно дешево при такій же якості. Вам зустрінуться ігри європейської та американської шкіл геймдизайну, а також гібридні стратегії[5]. Магазин проконсультує вас з приводу будь-якої гри, порадить аналоги й допоможе визначитися, яку купити настільну гру для себе або на подарунок. Також в наявності будь-які інші види друкованої продукції. Недоліком є надмірність в пошукових фільтрах і незручний їх вибір(рис. 1.4).

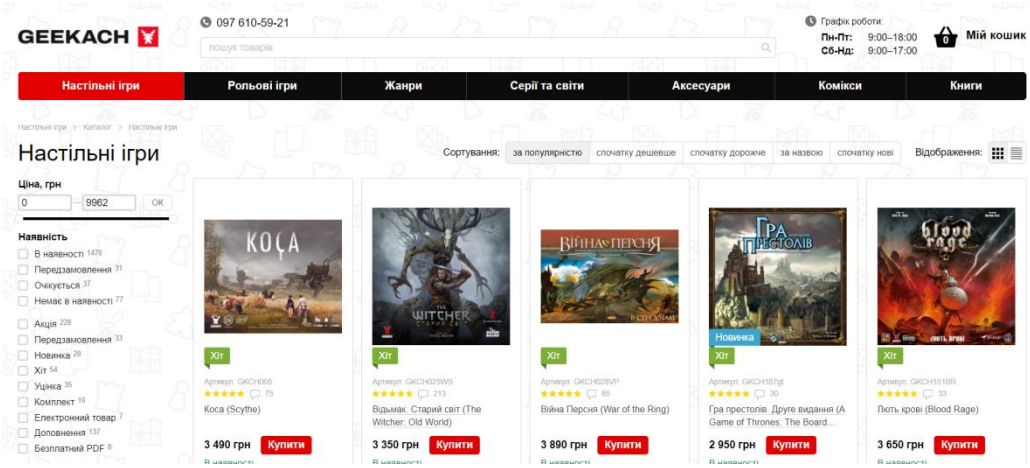

Рисунок 1.4 – Каталог настільних ігор інтернет-магазину Geekach

GameLand - магазин, де ви зможете придбати настільні ігри та аксесуари до них на будь-який смак і колір — ігри європейської чи американської школи,

варгейми, патігейми, дитячі, логічні, ігри на спритність, квести, органайзери, протектори та багато іншого[6]. Магазин із радістю запропонує списки та категорії, з яких варто почати своє перше знайомство з цим чудовим хобі. Крім того, до їх складу входить власне видавництво настільних ігор Нова Ера! Тому магазин ретельно працюємо над тим, щоби принести до нашої країни найкращі настільні ігри українською мовою. Недоліком є надмірність в пошукових фільтрах і незручний їх вибір, малий вибір товару(рис. 1.5).

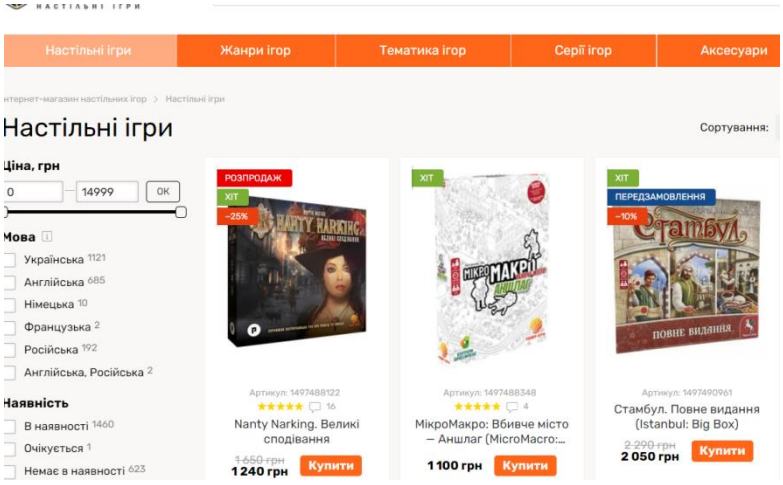

Рисунок 1.5 – Каталог настільних ігор інтернет-магазину Gameland

1.3 Висновки

В даному розділі було проаналізовано предметну область, проаналізовано етапи процесу розробки веб-додатків, визначено актуальні підходи до веб-розробки та обґрунтовано доцільність їх використання під час розробки веб-додатків.

Було розглянуто п’ять наявних аналогічні програмні рішення і виявлено їх переваги та недоліки. Наведені рішення мають малу кількість пунктів для фільтрування, один з них при малій кількості категорій фільтрування має надмірне розбиття окремої категорії на підкатегорії. Також в наведених рішеннях наводиться недостатня кількість інформації про продукт, лише вказуються посилання на сторонні ресурси з додатковою інформацією.

2 РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ПРОДАЖУ НАСТІЛЬНИХ ІГОР

2.1 Вибір архітектури розробки програмного модуля для продажу настільних ігор

Розробка будь-якого програмного забезпечення починається з визначення вимог до програмного забезпечення, а потім створення відповідної специфікації вимог до програмного забезпечення (SRS). Послідовно команда розробників чи розробник проходить різні етапи, такі як тестування, прийняття, розгортання, обслуговування тощо. Кожен процес розробки програмного забезпечення виконується за допомогою кількох послідовних кроків який відповідає цьому життєвому циклу розробки програмного забезпечення (SDLC).

На етапі проектування життєвого циклу розробки програмного забезпечення архітектура програмного забезпечення визначається та документується. Тож розглянемо один із важливих елементів життєвого циклу розробки програмного забезпечення (SDLC), тобто архітектуру програмного забезпечення.

Так як і архітектура будівлі, архітектура програмного забезпечення описує проектування та групу компонентів у системи, які складають будівельні блоки програмного забезпечення. Архітектура програмного забезпечення пояснює структурний склад програмного забезпечення та взаємодію між елементами. Принцип, який визначає схему організації програмного забезпечення для цих програмних систем, називається архітектурним шаблоном(далі архітектурним патерном)[7].

Архітектурний патерн фіксує структуру проектування різних систем і елементів програмного забезпечення, щоб їх можна було повторно використовувати. У процесі написання програмного коду розробники неодноразово стикаються з подібними проблемами в рамках проекту, в

компанії та в своїй кар’єрі. Один із способів вирішити цю проблему — створити шаблони проектування, які дають інженерам можливість багаторазового використання для вирішення цих проблем, дозволяючи розробникам програмного забезпечення досягати однакових структурних результатів для даного проекту.

Архітектурний патерн програмного забезпечення визначає фундаментальну організацію системи та простіше визначає структуроване рішення. Він визначає, як компонуються компоненти програмної системи, їхній взаємозв’язок і зв’язок між ними. Він служить схемою програмного застосування та основою розробки для команди розробників.

- Архітектура програмного забезпечення визначає перелік речей, що полегшує процес розробки програмного забезпечення.
- Архітектура програмного забезпечення визначає структуру системи.
- Архітектура програмного забезпечення визначає поведінку системи.
- Архітектура програмного забезпечення визначає взаємозв’язок компонентів.
- Архітектура програмного забезпечення визначає структуру зв’язку.
- Архітектура програмного забезпечення врівноважує потреби зацікавлених сторін.
- Архітектура програмного забезпечення впливає на структуру команди.
- Архітектура програмного забезпечення зосереджена на важливих елементах.
- Архітектура програмного забезпечення фіксує ранні дизайнерські рішення.

Архітектори поділяють характеристики архітектури на широкі категорії залежно від роботи, вимог, що рідко з’являються, структури тощо. Нижче пояснюються деякі важливі характеристики, які зазвичай розглядаються.

Характеристики операційної архітектури:

- Доступність

- Продуктивність
- Надійність
- Низька відмовостійкість
- Масштабованість

Характеристики структурної архітектури:

- Конфігурація
- Розширюваність
- Простота підтримки
- Портативність
- Ремонтопридатність

Наскрізні характеристики архітектури:

- Доступність
- Безпека
- Зручність використання
- Конфіденційність
- Здійсненність

SOLID принципи архітектури програмного забезпечення:

Кожен символ слова SOLID визначає один принцип архітектури програмного забезпечення. Цей твердий принцип дотримується, щоб уникнути помилок стратегії продукту. Архітектура програмного забезпечення повинна відповідати принципу SOLID, щоб уникнути будь-яких архітектурних збоїв або збоїв у розробці(рис. 2.1).

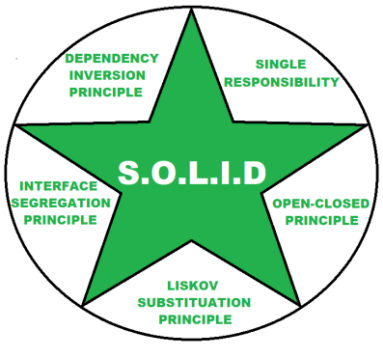

Рисунок 2.1 – принципи SOLID

- Єдина відповідальність – Кожна служба повинна мати єдину мету.
- Принцип відкрито-закрито – Програмні модулі повинні бути незалежними та розширюваними.
- Принцип підстановки Ліскова – Незалежні служби повинні мати можливість спілкуватися та замінювати одна одну.
- Принцип поділу інтерфейсу – Програмне забезпечення має бути розділене на такі мікросервіси, там не повинно бути жодних надмірностей.
- Принцип інверсії залежностей – Модулі вищих рівнів не повинні залежати від модулів нижчого рівня, а зміни на вищому рівні не впливатимуть на нижчий рівень.

Архітектура програмного забезпечення знаходиться на етапі проектування життєвого циклу розробки програмного забезпечення. Це один із початкових етапів усього процесу розробки програмного забезпечення. Без архітектури програмного забезпечення перехід до розробки програмного забезпечення подібний до будівництва будинку без проектування архітектури будинку.

Таким чином, архітектура програмного забезпечення є однією з важливих частин розробки програмного додатку. Нижче наведено причини важливості архітектури програмного забезпечення з точки зору технічних аспектів і аспектів розробки.

- Вибирає атрибути якості, які потрібно оптимізувати для системи.
- Сприяє ранньому створенню прототипів.
- Дозволяє побудувати систему покомпонентно.
- Допомагає керувати змінами в системі.

Крім усього цього, архітектура програмного забезпечення також важлива для багатьох інших факторів, таких як якість програмного забезпечення, надійність програмного забезпечення, ремонтопридатність програмного забезпечення, можливість підтримки програмного забезпечення та продуктивність програмного забезпечення тощо.

Переваги програмної архітектури:

- Забезпечує міцну основу для програмного проекту.
- Допомагає підвищити продуктивність.
- Зменшує вартість розробки.

Недоліки програмної архітектури:

- Іноді отримання хороших інструментів і стандартизації стає проблемою для архітектури програмного забезпечення.
- Початковий прогноз успіху проекту на основі архітектури не завжди можливий.

З вищезгаданого видно, наскільки важлива архітектура програмного забезпечення для розробки програмного додатку. Отже, хороша архітектура програмного забезпечення також відповідає за надання якісного програмного продукту.

Так як архітектура програмного забезпечення є настільки важливим складовою будь-якого програмного продукту, існує багато різних типів шаблонів архітектури програмного забезпечення, тому розглянемо лише деякі із них і те, як вони є невід'ємною частиною розробки програмного забезпечення.

Багаторівневий шаблон, ймовірно, один із найвідоміших патернів архітектури програмного забезпечення. Багато розробників використовують його, навіть не знаючи його назви. Ідея полягає в тому, щоб розділити ваш код на «рівні», де кожен рівень несе певну відповідальність і надає послуги вищому рівню[8].

Багаторівневий шаблон, мабуть, один із найвідоміших шаблонів архітектури програмного забезпечення. Багато розробників використовують його, навіть не знаючи його назви. Ідея полягає в тому, щоб розділити ваш код на «рівні», де кожен рівень несе певну відповідальність і надає послуги вищому рівню.

Патерни багаторівневої архітектури — це n-рівневі патерни, де компоненти організовані горизонтальними шарами. Це традиційний метод

розробки більшості програмного забезпечення, який має бути незалежним. Це означає, що всі компоненти взаємопов'язані, але не залежать один від одного.

Кожен рівень шаблону n-рівневої архітектури має певне призначення і відповідальність у програмі. Наприклад, презентаційний(клієнтський) рівень буде відповідати за обробку всього інтерфейсу користувача та логіки зв’язку браузера, тоді як бізнес-рівень буде відповідати за виконання конкретних бізнес-правил, пов’язаних із запитом(рис. 2.2).

Однією з потужних особливостей шаблону багаторівневої архітектури є розподіл проблем між компонентами. Компоненти в межах певного рівня мають справу лише з логікою, яка відноситься до цього рівня, іншими словами, кожен рівень має свою власну відповідальність[9].

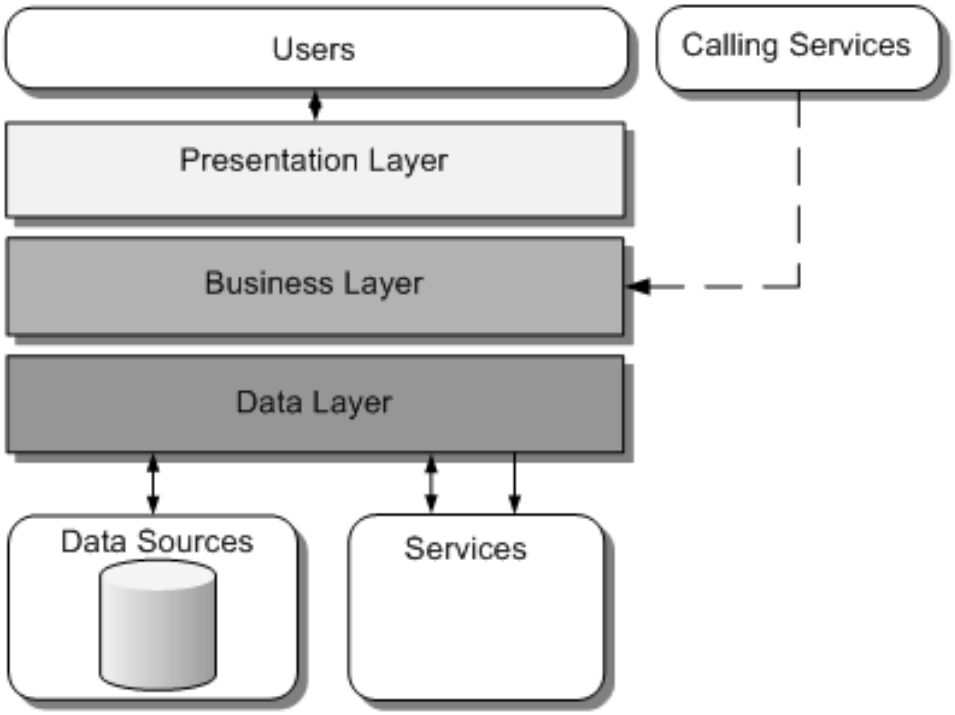

Рисунок 2.2 – багаторівнева архітектура

Переваги :

- Масштабованість, ви можете розділяти рівні, не впливаючи на інші рівні, а потім масштабувати кожен належним чином.
- Цілісність даних, каскадні ефекти запобігають, а обслуговування стає легшим. Можна змінити код, не впливаючи на дані.

- Повторне використання. Різні шари можна використовувати в різних проектах або в одному проекті ви можете використовувати той самий шар у різних місцях. Наприклад, ви можете написати розширення для своїх проектів і, якщо ви хочете, використовувати їх в іншому проекті
- Безпечно, ви можете захистити шари, не впливаючи на інші шари.

Недоліки :

- Збільшення зусиль. Замість того, щоб писати програмне забезпечення на одному рівні/шарові, розробнику потрібно розрізняти рівні та давати посилання на інші проекти (ваші бібліотеки класів) відповідно.
- Збільшення складності, при використанні N-Tier архітектуру, так як доведеться створити проект на початку відповідно до n-шарової логіки. Створення різних шарів ускладнює, оскільки більше шарів означає більше речей, про які потрібно думати в майбутньому.

Шаблон архітектури клієнт-сервер[9] - це розділення системи на клієнтів (інтерфейс користувача) і сервер (обробка даних) для керування обміном даних та взаємодії з користувачем(рис. 2.3).

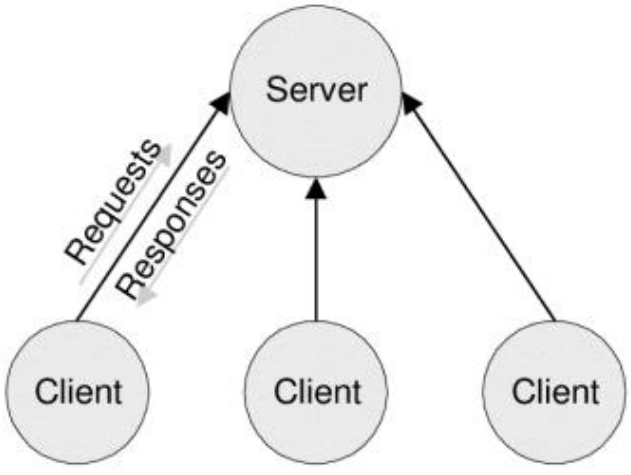

Рисунок 2.3 – клієнт-сервер архітектура

Нижче наведено особливості цієї моделі:

1. Клієнтські компоненти надсилають запити на сервер, який обробляє їх і

відповідає.

2. Коли сервер приймає запит від клієнта, він відкриває з’єднання з клієнтом за певним протоколом.
3. Сервери можуть мати стан і без нього. Сервер із збереженням стану може отримувати кілька запитів від клієнтів. Він підтримує запис запитів від клієнта, і цей запис називається “сесією”.

Зазвичай запити обробляються в окремих потоках на сервері. Зв'язок між процесами викликає накладні витрати. Запити та результати часто доводиться трансформувати або сортувати, оскільки вони мають різне представлення на клієнті та сервері та через мережний трафік.

Розподілені системи з багатьма серверами з однаковими функціями повинні бути прозорим для клієнтів: клієнтам не повинно бути необхідності розрізняти сервери. Коли ви вводите URL-адресу для Google, для наприклад, вам не потрібно знати точну машину, до якої здійснюється доступ (прозорість розташування), платформу машини (платформа прозорість), маршрут вашого запиту тощо. Проміжний рівні можуть бути вставлені для певних цілей: кешування, безпеки або завантаження балансування, наприклад.

Іноді для сповіщення про подію потрібні зворотні виклики. Це також може бути розглядається як перехід до шаблону Peer-to-Peer.

Переваги :

- Цілісність даних. Клієнти отримують доступ до даних із сервера за допомогою авторизованого доступу, що покращує обмін даними.
- Незалежні клієнт-серверні програми можна створювати незалежно від платформи чи стеку технологій.
- Ремонтопридатність, це розподілена модель із певними обов’язками для кожного компонента, що полегшує обслуговування.

Недоліки :

- Проблеми масштабованості сервера під час інтенсивного трафіку.
- Комплексне управління зв'язком між клієнтами та сервером.
- Потенційна єдина точка відмови, якщо сервер вийде з ладу.

Архітектура REST. REST означає Representational State Transfer[10]. Архітектура REST - архітектура клієнт-сервер, де клієнти відокремлені від серверів єдиний інтерфейс, сервери пропонують адресні ресурси та комунікацію не має стану. Веб-додатки зі зв'язком без стану дотримуються правил цього патерну(рис. 2.4).

Кожен запит клієнта повинен містити всю інформацію, необхідну серверу для успішної відповіді. Усю інформацію про стан потрібно передати назад клієнту як частину відповіді[5]. Це «S» і «T» у REST; ми завжди повертаємо стан клієнту. Сервер може мати стан, але в цьому випадку кожен серверний стан має бути доступним для адресації (наприклад, за URL-адресою). Архітектура REST також є багаторівневою системою: для клієнта вона прозора він підключений безпосередньо до сервера або через одного чи кількох посередників.

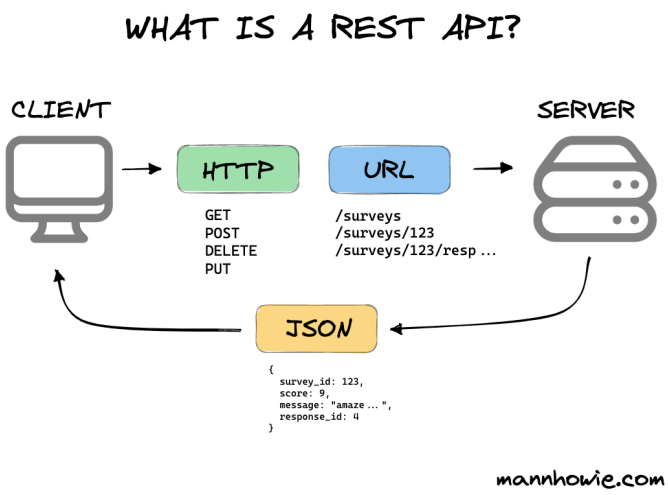

Рисунок 2.4 – Схема роботи REST API

Уніфіковане обмеження інтерфейсу одна з з головних вимог до додатків з використанням REST архітектури. Уніфікований інтерфейс надає перевагу загальному технічному контракту, який визначає, як клієнт і сервер обмінюються запитами та відповідями. Його можна однаково застосовувати до багатьох різних систем, видаливши з цього інтерфейсу будь-яку мову бізнес-домену[11].

Це обмеження має чотири додаткові вимоги для створення єдиного інтерфейсу:

- Ресурс та ідентифікація ресурсів
- Маніпулювання ресурсами через представлення
- Самоописні повідомлення
- Гіпермедіа як основа стану додатків

Ресурс є основною абстракцією в будь-якій системі на основі REST. Ресурсом може бути що завгодно – ваше водійське посвідчення, документ, зображення вашого домашнього улюбленця: усі ресурси. Навіть такі тимчасові речі, як сьогоднішня погода чи завтрашнє обіднє меню, можуть бути ресурсами. Ресурс — це “концептуальне відображення сутності або набору сутностей, а не сутність, яка відповідає відображенню в будь-який момент часу”. Ресурс може навіть зіставлятися з порожніми поняттями. Наприклад, вам може знадобитися ресурс для завтрашньої газети. Це відповідає завтрашній газеті; самого паперу ще не існує, але ресурс може існувати.

Ідентифікація ресурсів є ще одним важливим аспектом визначення єдиного інтерфейсу. Кожен ресурс повинен мати унікальний ідентифікатор, який, в ідеалі, ніколи, ніколи не змінюється. Це дозволяє клієнтам надійно взаємодіяти з ресурсом у системі на основі REST протягом тривалого часу. Орган іменування, відповідальний за призначення ідентифікаторів ресурсів, повинен підтримувати це дійсне зіставлення з часом. Візьмемо, наприклад, міжнародний стандартний номер книги або ISBN і те, як вони використовуються в усьому світі для призначення 13-значного унікального ідентифікатора книгам та іншим носіям. Міжнародне агентство ISBN є органом, відповідальним за підтримку та присвоєння цього унікального номера.

Представлення — це моментальний знімок стану даного ресурсу в часі. Стан надходить у форматі, описаному метаданими повідомлення. Формат може бути двійковим або текстовим парами ключ-значення.

Вся інформація, якою обмінюються клієнт і сервер, повинна міститися в

повідомленнях між ними. Це означає, що стандартний набір методів, типів подання має міститися в самому повідомленні, а не бути прихованим у будь-якій реалізації клієнта чи сервера.

Загальна ідея гіпермедіа як основа стану додатків полягає в тому, що клієнт підтримує стан програми через представлення ресурсів. Валідні переходи включаються в самі представлення за допомогою різних механізмів (тобто зв’язування). Клієнт розуміє все, що він може зробити з ресурсом в поточному стані та представленні.

Шаблон брокер використовується для структурування розподілених систем із відокремленими компонентами, які взаємодіють за допомогою віддаленого виклику служби[10]. Такі системи дуже негнучкі, коли компоненти повинні знати один одного розташування та інші деталі. Компонент-брокер відповідає за координацію зв’язку між компонентами: він пересилає запити та передає результати та помилки.

Сервери публікують свої можливості (послуги та характеристики) брокеру. Клієнти запитують послугу у брокера, а брокер потім перенаправляє клієнта до відповідної служби зі свого реєстру(рис. 2.5).

Використання шаблону брокер означає, що жоден інший компонент, крім брокера повинен зосереджуватись на низькорівневому міжпроцесному зв’язку.

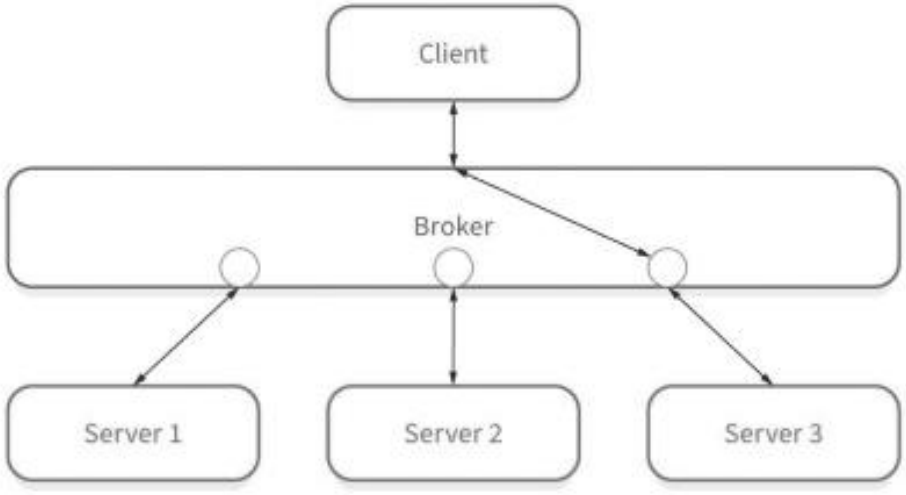

Рисунок 2.5 – архітектура брокер

Прикладом, у якому використовується шаблон Broker, є веб-сервіси. У програмі веб-служб, як показано на рисунку 2.6, сервер з реєстром UDDI є брокером. UDDI означає Universal Discovery, Description and Integration. Це сховище веб-сервісів. The IDL, який використовується для веб-сервісів, це WSDL. SOAP (Simple Object Access Protocol) — це транспортний протокол для повідомлень, написаних у форматі XML.

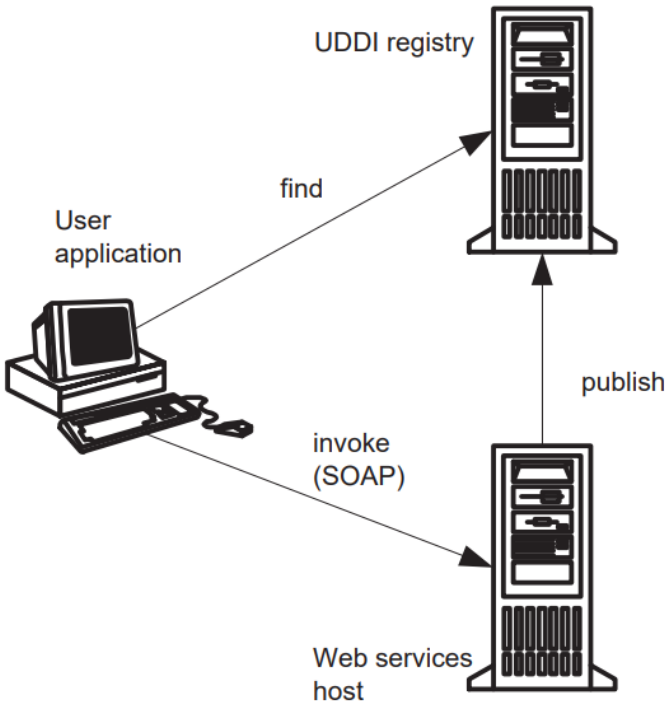

Рисунок 2.6 - приклад використання патерну брокер

Іншим прикладом шаблону брокера є CORBA, для співпраці між гетерогенні об'єктно-орієнтовані системи та веб-сервіси.

Патерн Broker дозволяє динамічно змінювати, додавати, видаляти та переміщувати об’єкти, і це робить розповсюдження прозорим для розробника. Це вимагає стандартизації описів послуг. При взаємодії двох брокерів можуть знадобитися мости для того, щоб приховати деталі впровадження.[10]

Переваги[9] :

- Незалежність, розділення зв'язку між сервером і клієнтом
- Гнучкість, можливі невеликі зміни завдяки інтерфейсу. Компоненти можна замінити без створення проблем для інших частин

програмного забезпечення, і клієнту потрібно знати лише проксі-сервер на стороні клієнта від локального комп'ютерного брокера та ім'я сервера, щоб запитувати інформацію

- Проксі дозволяють створювати клієнт і сервер на різних мовах програмування. Вони не залежать від платформи

Недоліки :

- Збій локального брокера також впливає на відповідні компоненти
- Складніша комунікація між компонентами системи
- Потребую високої пропускної здатності
- Проксі залежать від брокера

Шаблон мікросервісів передбачає створення кількох додатків або мікросервісів, які можуть працювати взаємозалежне. Хоча кожен мікросервіс можна розробити та розгорнути незалежно, його функціональні можливості переплітаються з іншими мікросервісами.

Ключовою концепцією шаблону мікросервісів є окреме розгортання підпрограм(рис. 2.7). Це створює спрощений потік виконання, який дозволяє легко розгортати мікросервіси та підвищує масштабованість програми. Ще одна ключова особливість цього шаблону полягає в тому, що це розподілена архітектура, тобто компоненти структури можуть бути повністю відокремлені та доступні через протоколи віддаленого доступу, такі як REST, SOAP або GraphQL. Цей розподілений характер шаблону забезпечує його високі властивості масштабованості[7].

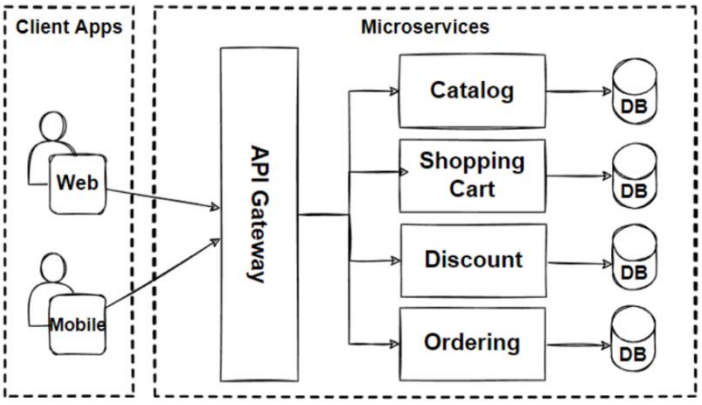

Рисунок 2.7 – мікросервісна архітектура

Переваги :

- Покращена ізоляція несправностей. Великі додатки можуть залишатися здебільшого справними через збій одного модуля.
- Усуваючи прив’язку до постачальника або технології.
- Менші та швидші розгортання, менші кодові бази та обсяг = швидші розгортання, які також дозволяють почати досліджувати переваги безперервного розгортання.
- Масштабованість: оскільки ваші служби є окремими, ви можете легше масштабувати найбільш необхідні у відповідний час, а не всю програму.

Недоліки :

- Зв’язок між службами складний, оскільки тепер усе є незалежною службою, вам потрібно ретельно обробляти запити, що передаються між вашими модулями.
- Більше послуг означає більше ресурсів, кілька баз даних і керування транзакціями можуть бути болючими.
- Глобальне тестування складне, тестування програми на основі мікросервісів може бути громіздким.
- Виправлення та знаходження джерела помилок є набагато складнішим, так як кожна служба має власний набір журналів(логів) для перегляду.
- Проблеми з розгортанням продукту потребує координації між кількома службами.

Патерн архітектури мікроядра складається з двох типів компонентів архітектури: основної системи та плагіних модулів. Логіка програми розділена між незалежними модулями плагінів і базовою системою ядра, що забезпечує розширюваність, гнучкість і ізоляцію функцій програми та спеціальної логіки обробки[8].

Патерн архітектури мікроядра є природним шаблоном для реалізації додатків на основі продукту. А програмна програма на основі продукту – це

така, яка упакована та доступна для завантаження у версіях як типовий сторонній продукт. Однак багато компаній також розробляють і випускають свої внутрішні бізнес-додатки, такі як програмні продукти, укомплектовані версіями, примітками до випуску та функціями, що підключаються[3].

Це стосується програмних систем, які повинні мати можливість адаптуватися до мінливих системних вимог. Він відокремлює мінімальне функціональне ядро від розширених функціональних можливостей і частин, призначених для клієнта. Він також слугуватиме розеткою для підключення цих розширень і координації їхньої співпраці.

Патерн архітектури мікроядра дозволяє додавати додаткові функції програми як плагіни до основної програми, забезпечуючи розширюваність, а також розділення та ізоляцію функцій.

Патерн архітектури мікроядра складається з двох типів компонентів архітектури: основної системи та плагінних модулів. Логіка програми розділена між незалежними модулями плагінів і базовою системою ядра, що забезпечує розширюваність, гнучкість і ізоляцію функцій програми та спеціальної логіки обробки. А основна система шаблону архітектури мікроядра традиційно містить лише мінімальну функціональність, необхідну для забезпечення працездатності системи(рис. 2.8).

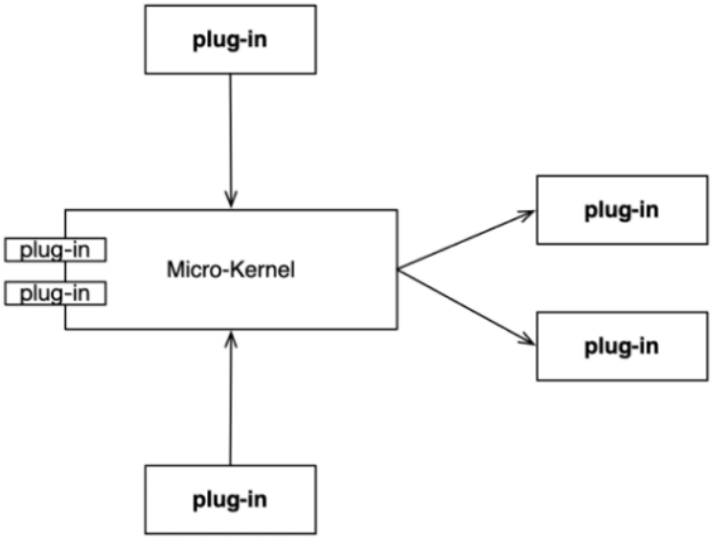

Рисунок 2.8 – архітектура мікроядер

Можливо, найкращим прикладом архітектури мікроядра є Eclipse IDE. Завантаження основного продукту Eclipse надає вам трохи більше, ніж редактор. Однак, коли ви починаєте додавати плагіни, він стає корисним продуктом, який можна налаштовувати.

Переваги :

- Велика гнучкість і розширюваність
- Деякі реалізації дозволяють додавати плагіни під час роботи програми
- Хороша транспортабельність
- Простота розгортання
- Швидка реакція на постійно мінливе середовище
- Плагінні модулі можна тестувати ізольовано, і їх можна легко імітувати основною системою, щоб продемонструвати або створити прототип певної функції з невеликими змінами або без змін у системі ядра.
- Висока продуктивність, оскільки ви можете налаштувати та оптимізувати програми, щоб включати лише ті функції, які вам потрібні.

Недоліки

- Надання послуг у мікроядерній системі є дорогим у порівнянні зі звичайною монолітною системою. Перемикач контексту або виклик функції, необхідні, коли драйвери реалізовані як процедури або процеси відповідно.
- Може бути важко визначити, що належить до мікроядра, а що ні.
- Попередньо визначений API може не підійти для майбутніх плагінів.

Просторова архітектура (англ. Space-based architecture,SBA)[8] — це підхід до розподілених обчислювальних систем, де різні компоненти взаємодіють один з одним шляхом обміну кортежами або записами через один або кілька спільних просторів. Це відрізняється від більш поширених підходів служби черги повідомлень, де різні компоненти взаємодіють один з одним

шляхом обміну повідомленнями через брокера повідомлень. У певному сенсі обидва підходи обмінюються повідомленнями з деяким центральним агентом, але те, як вони обмінюються повідомленнями, дуже відмінне(рис. 2.9).

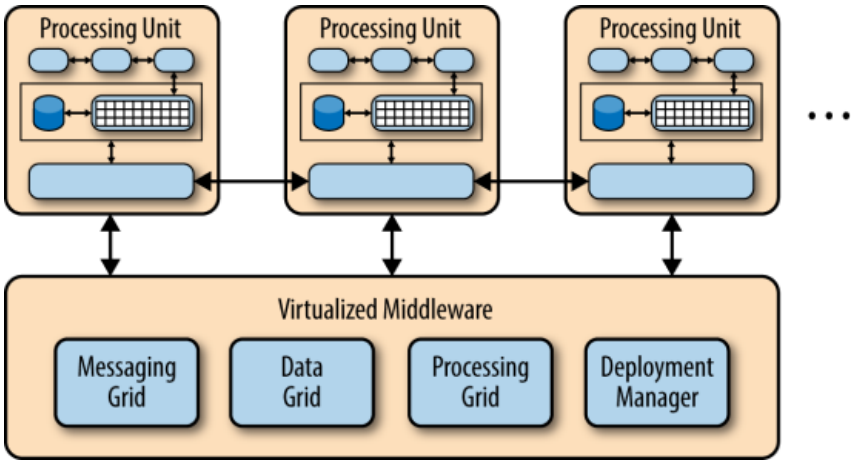

Рисунок 2.9 – просторова архітектура

Шаблон на основі простору мінімізує фактори, які обмежують масштабування програми. Цей шаблон отримав свою назву від концепції кортежного простору, ідеї розподіленої спільної пам’яті. Висока масштабованість досягається шляхом усунення обмеження центральної бази даних і використання натомість реплікованих сіток даних у пам’яті. Дані програми зберігаються в пам’яті та реплікуються між усіма активними процесорами. Блоки обробки можуть динамічно запускатися та вимикатися в міру збільшення та зменшення навантаження користувача, таким чином вирішуючи змінну масштабованість. Оскільки центральної бази даних немає, вузьке місце бази даних видалено, забезпечуючи майже нескінченну масштабованість у програмі.

Більшість додатків, які відповідають цій моделі, є стандартними веб-сайтами, які отримують запит від браузера та виконують певну дію. Хорошим прикладом цього є аукціонний сайт для торгів. Сайт постійно отримує ставки від користувачів Інтернету через запит браузера. Програма отримуватиме ставку на певний товар, записуватиме цю ставку з міткою часу, оновлюватиме останню інформацію про ставку для товару та надсилатиме інформацію назад у браузер.

Архітектурний шаблон на базі простору спеціально розроблений для вирішення та вирішення проблем масштабованості та паралельності. Це також корисний шаблон архітектури для програм, які мають змінні та непередбачувані обсяги одночасних користувачів. Висока масштабованість досягається шляхом усунення обмеження центральної бази даних і використання натомість реплікованих сіток даних у пам'яті[7][8].

Простора архітектура розроблена для уникнення функціонального збою під високим навантаженням шляхом розподілу як обробки, так і зберігання між кількома серверами.

Переваги :

- Швидко реагує на постійні зміни середовища.
- Незважаючи на те, що просторові архітектури зазвичай не роз'єднані та розподілені, вони динамічні, а складні хмарні інструменти дозволяють легко "виштовхувати" додатки на сервери.
- Висока продуктивність досягається завдяки механізмам доступу до даних у пам'яті та кешування, вбудованих у цей шаблон.
- Висока масштабованість пояснюється тим фактом, що централізована база даних практично не залежить від централізованої бази даних, що, по суті, усуває це обмежене вузьке місце з рівняння масштабованості.

Недоліки:

- Досягнення дуже високих навантажень користувачів у тестовому середовищі є дорогим і трудомістким, що ускладнює тестування аспектів масштабованості програми.
- Складне кешування та сітка даних у пам'яті роблять цей шаблон відносно складним для розробки, здебільшого через відсутність знайомства з інструментами та продуктами, які використовуються для створення такого типу архітектури. Крім того, при розробці таких типів архітектур необхідно приділяти особливу увагу, щоб переконатися, що ніщо у вихідному коді не впливає на продуктивність і масштабованість.

Патерн архітектури трубного фільтра має мету в розділені основних компонентів/процесів на велику кількість незалежних компонентів, які можна обробляти одночасно(рис. 2.10). Конструкція конвеєрного фільтра найкраще підходить для додатків на основі веб-сервісів, які обробляють дані в потоці та можуть генерувати від простих до складних структур.

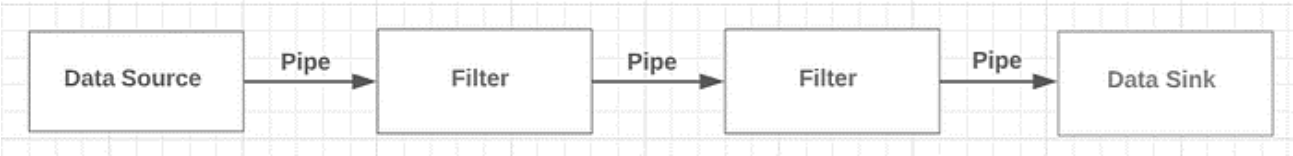

Рисунок 2.10 - архітектура трубного фільтра

Переваги:

- Підходить для обробки, яка вимагає чітких систематичних кроків для перетворення послідовних фрагментів даних завдяки інтуїтивно зрозумілому процесу обробки
- Кожен фільтр можна легко модифікувати — за умови, що вхідні та вихідні дані залишаються незмінними
- Фільтри можна багаторазово використовувати, старі фільтри можна замінити новими або нові фільтри можна легко вставити в процес обробки — за умови, що вхідні та вихідні дані між фільтрами сумісні
- Кожен компонент реалізується як окреме, відмінне завдання, отже, має природний поділ завдань.

Недоліки:

- Неефективно та незручно передавати повний набір даних по всій системі труб і фільтрів, оскільки не кожен компонент потребуватиме повного набору даних
- Надійність може бути проблемою, якщо дані втрачаються на шляху між компонентами
- Занадто багато фільтрів може уповільнити вашу програму, створивши вузькі місця або тупикові блокування, якщо один конкретний фільтр обробляється повільно або не працює

В результаті аналізу поширених архітектурних патернів програмного забезпечення було обрано REST патерн, так як інші патерни направлені на

вирішення проблем складних високонавантажених систем, що створює надмірну комплексність при написані програмного модуля для продажу настільних ігор та потребує більшої кількості ресурсів ніж REST патерн.

Клієнтом програмного модуля буде виступати запущена фронтенд частина, з якою і буде взаємодіяти користувач, Сервер буде складатись з контролерів до яких будуть надходити запити та з шару доступу до даних(Data access layer, DAL).

2.2 Розробка UML-діаграм програмного модуля

Перед тим як почати розробляти WEB-додаток, потрібно звернути уваги на визначення всіх варіантів реалізації UML-діаграм. Розглянемо визначення та види діаграм.

UML-діаграма є уніфікованою мовою моделювання, яка дозволяє візуалізувати процеси та функціонування систем. UML підтримує всі етапи життєвого циклу складного об’єкта та його компонентів, надаючи для цього різноманітні графічні засоби – діаграми [12]. UML-діаграми поділяються на два основні типи: структурні та діаграми поведінки (рис. 2.11).

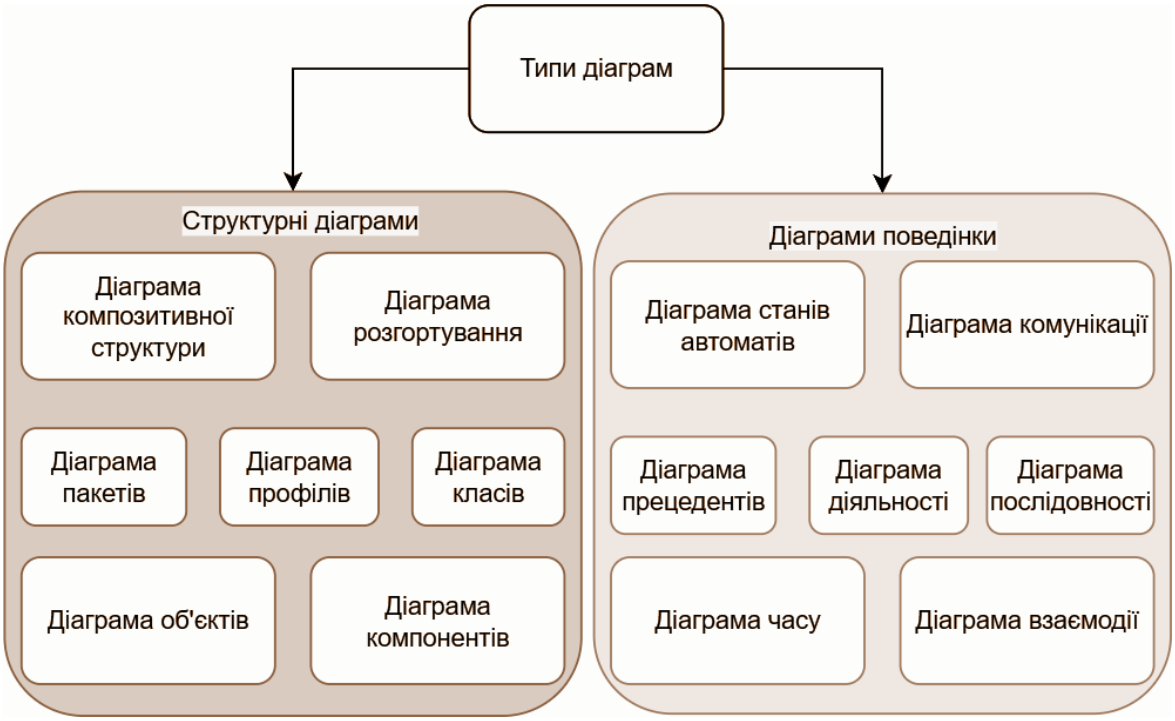

Рисунок 2.11 – Загальне представлення видів UML-діаграм

Структуровані UML-діаграми використовуються для зображення статичної структури системи. Головними видами структурних UML-діаграм є діаграми класів, об'єктів, компонентів, розгортання та пакетів.

Діаграми поведінки UML описують динамічну поведінку системи. До таких діаграм належать діаграми взаємодії, станів та активностей. Наприклад, діаграма послідовності відображає послідовність взаємодії між об'єктами в системі в межах конкретного сценарію.

Після аналізу UML-діаграм було вирішено використовувати такі види діаграм: діаграми класів, послідовності та прецедентів.

Шар доступу до даних відповідає за представлення таблиць СУБД у вигляді окремих об’єктів(сутностей) та для відокремлення логіки взаємодії з базою даних від іншої частини програми. Тому цей шар буде складатись з двох складових - модуля об’єктно-реляційного відображення та з модуля, що реалізовує патерн репозиторій.

Об'єктно-реляційне відображення (англ. Object-relational mapping, ORM) — технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних»[13]. ORM використовує дескриптори метаданих для створення рівня між мовою програмування та реляційною базою даних. Таким чином, він з’єднує код об’єктно-орієнтованої програми (ООП) з базою даних і спрощує взаємодію між реляційними базами даних і мовами ООП(рис. 2.12)[14].

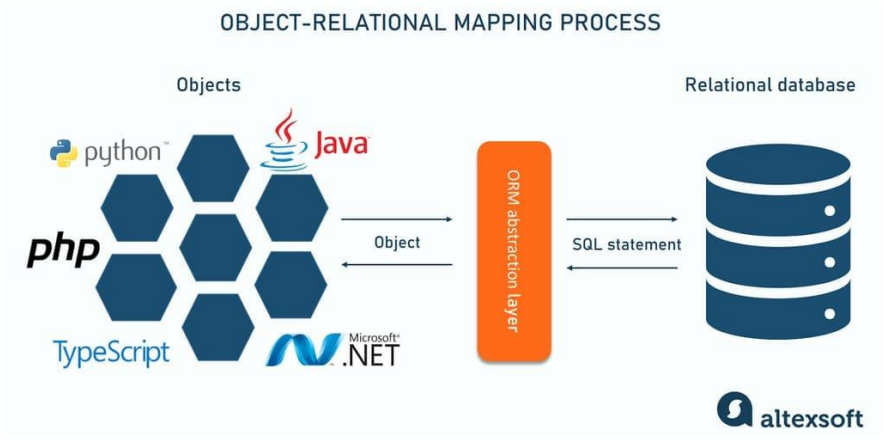

Рисунок 2.12 – Рівень абстракції ORM між об’єктами та таблицями в реляційній базі даних

Ідея ORM базується на абстракції. Механізм ORM дає змогу адресувати об’єкти, отримувати доступ до них і маніпулювати ними, не враховуючи, як ці об’єкти пов’язані зі своїми джерелами даних. ORM дозволяє програмістам підтримувати узгоджене уявлення про об’єкти протягом тривалого часу, навіть коли змінюються джерела, які їх доставляють, приймачі, які їх отримують, і програми, які отримують до них доступ[15]. Тому отримуємо клас ApplicationDbContext, що буде керувати операціями з СУБД та класи-сутності, що будуть представляти окрему таблицю з бази даних.

У більшості сценаріїв бази даних використовуються одночасно декількома підпрограмами додатку, кожен з яких виконує зміни в даних незалежно один від одного. Коли одні й ті самі дані змінюються одночасно, можуть виникнути невідповідності та пошкодження даних, наприклад коли два клієнти змінюють різні стовпці в одному рядку, які певним чином пов’язані. Для забезпечення узгодженості даних у разі таких одночасних змін використовуються механізми контролю паралелізму, одним з яких є керування паралельним виконанням на основі часових позначок, яке використовує часові позначки для впорядкування та планування транзакцій[16].

Для реалізації обрано оптимістичний паралелізм, який передбачає, що конфлікти паралелізму є відносно рідкісними. На відміну від песимістичних підходів, які блокують дані заздалегідь і лише потім переходять до їх модифікації, оптимістичний паралелізм не вимагає блокувань, але організовує помилку модифікації даних під час збереження, якщо дані змінилися після запиту. Про цю помилку паралельного виконання повідомляється програмі, яка відповідним чином справляється з нею, можливо, повторюючи всю операцію над новими даними[17].

Оптимістичний паралелізм реалізується шляхом налаштування властивості як маркера паралелізму у вигляді мітки часу(timestamp). Маркер паралельності завантажується та відстежується під час запиту до таблиць, як і будь-який інший параметр.

Потім, під час операцій оновлення або видалення, значення маркера

паралельності в базі даних порівнюється з вихідним значенням, отриманим з програми. Якщо маркери часу виявились однаковими - проводимо відповідну операцію та оновлюємо маркер часу[17].

У кожної настільної гри є автори та художники, що створили гру, кожен автор чи художник приймали участь в створені багатьох ігор, тип зв'язку багато до багатьох, але це не обов'язковий зв'язок(наприклад шахи не мають конкретного автора чи дизайнера).Також є видавництво, що виробляє гру з дозволу авторів і кожне видавництво виготовляє декілька ігор, обов'язковий зв'язок один до багатьох. У кожної гри в базі даних є лише один певний постачальник(продавець), що займається продажем багатьох ігор, обов'язковий зв'язок один до багатьох. Кожна гра належить певним категоріям та включає в себе групу механік, механіка і категорія представлені в багатьох іграх, так як гра може і не належати до якоїсь певної категорії чи описуватись раніше зареєстрованими механіками отримуємо необов'язковий зв'язок багато до багатьох. Можливість оформлення замовлень є лише в зареєстрованих користувачів, в кожне замовлення входять група ігор з певною кількістю для кожної гри та з актуальною ціною з застосуванням знижок, зв'язок обов'язковий багато до багатьох. Кожне замовлення оформлене на одного відповідного користувача, отримуємо можливий зв'язок, що є необов'язковий для користувача, але обов'язковий для замовлення, один до багатьох. Кожне замовлення направлене до одного відповідного постачальника, що має панель управління його настільними іграми та замовленнями.З сутностями замовлення та грою має можливий зв'язок, так як для цих сутностей зв'язок є обов'язковим, але є необов'язковим для сутності постачальника, отримуємо зв'язки один до багатьох.

За допомогою опису сутностей програмного модулю можемо створити ER-модель як на рисункові 2.13.

Для кожної сутності спільними є два поля - поле ідентифікатор, що слугує для ідентифікації окремого запису та для зв'язку між таблицями; та з поля відмітки часу(timestamp) для контролю паралелізму.

Доцільно створити єдиний клас BaseEntity, що буде мати ці два поля та який буде базовим класом на інших класів.

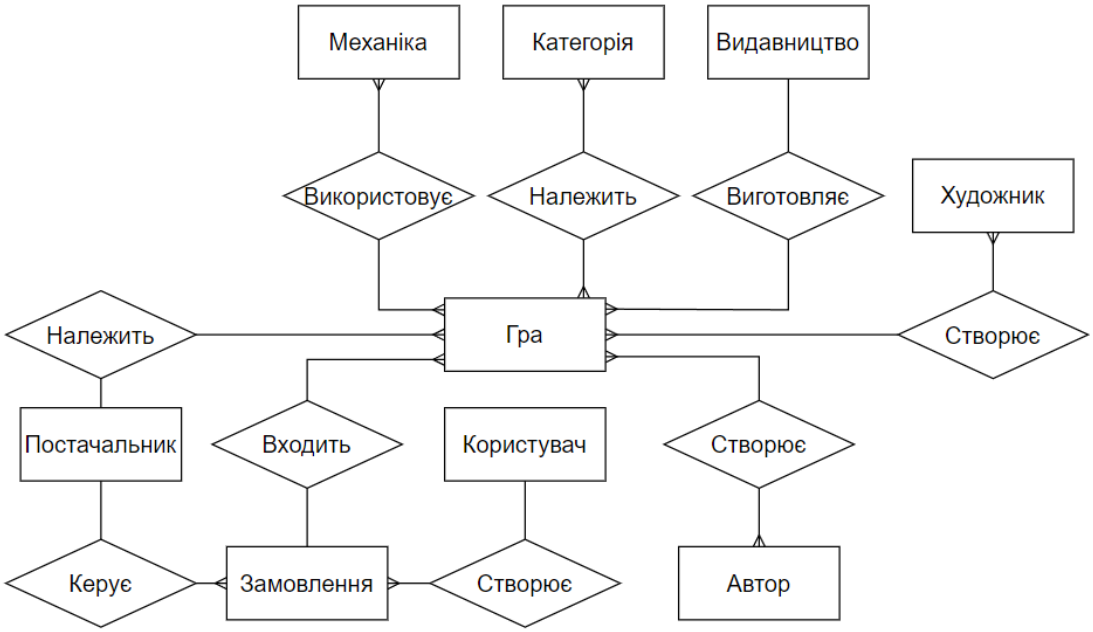

Рисунок 2.13 – ER-модель програмного модулю для продажу настільних ігор

Для реалізації зберігання всіх необхідних даних потрібні наступні класи : Artist, Author, Publisher, Vendor, Boardgame, Category, Mechanic, User, Order. Для проекції відношення багато для багатьох створимо додаткові класи-сутності BoardgameToArtist, BoardgameToAuthor, BoardgameToCategory, BoardgameToMechanic, VendorEmployee, OrderItem.

Діаграма класів UML[18] — це графічна нотація, яка використовується для побудови та візуалізації об’єктно-орієнтованих систем. Діаграма класів в уніфікованій мові моделювання — це тип статичної структурної діаграми, яка описує структуру системи, показуючи системні:

- класи,
- їхні атрибути,
- операції (або методи),
- і зв’язки між об’єктами.

Зобразимо отримані класи-сутності використовуючи UML діаграму класів(рис. 2.14).

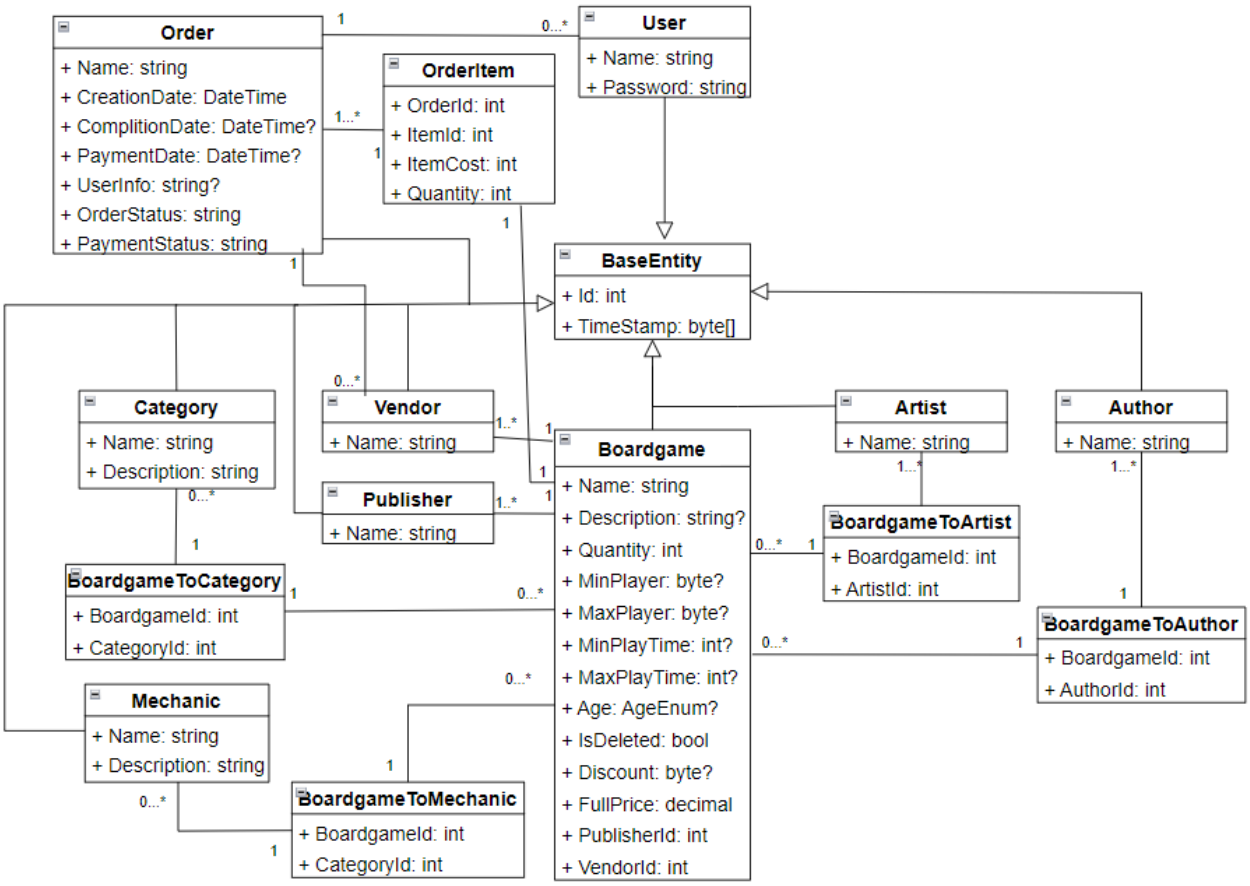

Рисунок 2.14 – UML діаграма класів програмного модуля для продажу настільних ігор

Для подальшої розробки програмного модуля для продажу настільних ігор визначимо типи користувачів та їх варіанти взаємодії з додатком.

Анонімний користувач або гість - має можливість переглядати каталог товарів та фільтрацією товарів за категоріями, механіками, авторами, художниками, цінами, кількістю гравців та віковою категорією. Також він зможе додавати предмети в кошик для подальшого оформлення замовлення після реєстрації. Також може переглядати детальну інформацію про конкретну гру.

Зареєстрованому користувачу доступні всі можливості анонімного користувача і додатково може створювати замовлення на основі предметів в кошику та переглядати список своїх замовлень.

Постачальник здатний додавати настільні ігри та редагувати їх, керувати та переглядати замовленнями від користувачів, додавати та редагувати

категорії, механіки, авторів, художників. Також він може переглядати детальну інформацію про товар у вигляді, в якому бачить її користувач.

Для візуалізації типів користувачів та їх можливостей використаємо UML діаграму прецедентів(рис. 2.15-2.16). UML діаграма прецедентів(діаграми варіантів використання, use case diagrams) – це узагальнена модель функціювання системи, що представляє динамічні або поведінкові аспекти складного об’єкта. Суть даної діаграми полягає в тому, що проектований складний об’єкт представляється у вигляді множини сутностей або акторів, що взаємодіють із складним об’єктом за допомогою так певних модельованих варіантів використання. При цьому виконавцем (actor), дійовою особою або актором називається будь-яка сутність, що взаємодіє із системою ззовні. Це може робити людина, технічне обладнання, програмне забезпечення, веб-додаток або будь-який інший складний об’єкт, який може бути джерелом впливу на складний об’єкт, що проектується(як визначить сам розробник). Тоді як варіант використання (use case) потрібне для опису сервісів, які складний об’єкт надає актору.Іншими словами, кожен варіант використання визначає деякий набір дій, який здійснюється складним об’єктом при діалозі з актором. При цьому, не йдеться про те, яким чином буде реалізовано взаємодію акторів із складним об’єктом [19].

Складовими діаграми такого типу є виконавець та прецедент. Виконавець (дійова особа, Actor) –множина логічно пов'язаних ролей, виконуваних при взаємодії з прецедентами або сутностями (складний об’єкт, система, підсистема або клас).

Прецедент–опис множини послідовних подій (включаючи варіанти), що виконуються складним об’єктом, які призводять до спостережуваного актором результату, і являє поведінку сутності, описуючи взаємодію між актором і складним об’єктом.Прецедент не показує, як досягається певний результат, а тільки зазначає які саме дії виконуються.

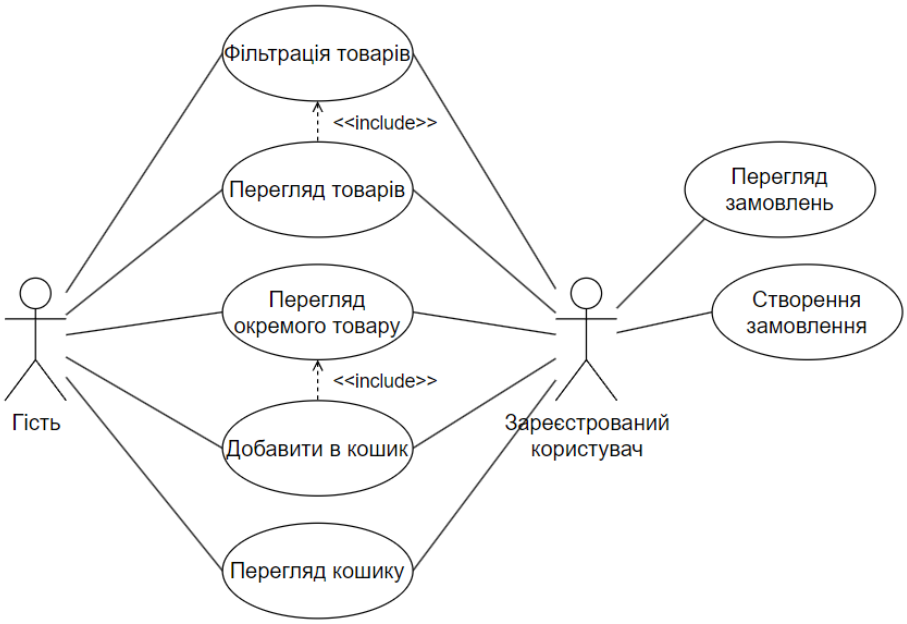

Рисунок 2.15 – діаграма прецедентів для гостя та зареєстрованого користувача

Рисунок 2.16 – діаграма прецедентів для постачальника

Розглянемо наступну складову шару доступу до даних, що відділяє взаємодію ORM з базою від серверної логіки програми - патерн репозиторій.

Репозиторій - посередник між шарами домену та представлення даних за допомогою інтерфейсу, подібного до колекції, для доступу до об’єктів домену(рис. 2.17)[20].

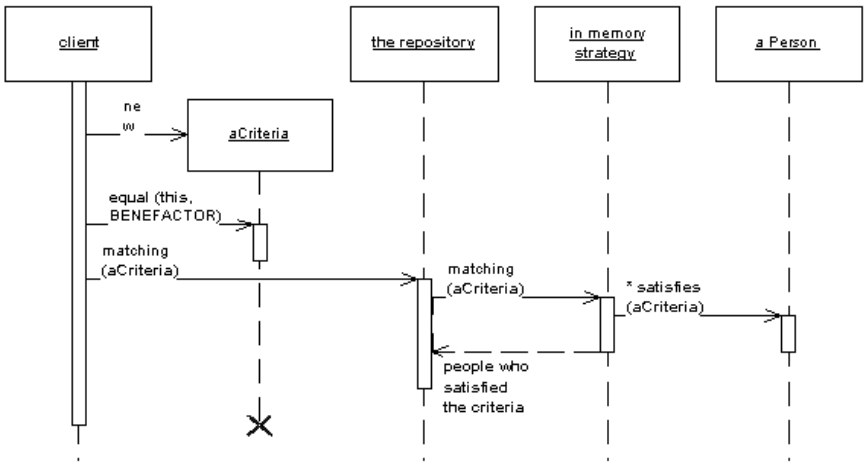

Рисунок 2.17 – демонстрація розміщення репозиторію для певної програми з використанням UML діаграми послідовності

Система зі складною моделлю предметної області часто виграє від рівня, такого як той, який забезпечує ORM, який ізолює об’єкти домену від деталей коду доступу до бази даних. У таких системах варто побудувати інший рівень абстракції поверх рівня представлення, де зосереджено код побудови запиту. Це стає більш важливим, коли існує велика кількість класів домену або інтенсивні запити. Зокрема, у цих випадках додавання цього рівня допомагає мінімізувати повторювану логіку запитів.

Репозиторій є посередником між шарами серверу та представлення даних, діючи як колекція об’єктів предметної області в пам’яті. Серверні об’єкти створюють специфікації запиту та надсилають їх до репозиторію для задоволення. Об’єкти можна додавати до сховища та видаляти з нього, як і з простої колекції об’єктів, а код представлення, ізольований репозиторієм, виконуватиме відповідні операції. Концептуально репозиторій ізолює, приховує набір об’єктів, що зберігаються в сховищі даних, і операції, що виконуються над ними, надаючи більш об’єктно-орієнтоване уявлення про рівень збереження. Репозиторій також підтримує мету досягнення чіткого розділення та односторонньої залежності між сервером та шаром представлення даних.

Для розробки шару доступу до даних створимо один загальний репозиторій, що буде реалізовувати базові операції над BaseEntity та будемо

розширяти його можливості в класах-нащадках.

BaseRepository<T> буде узагальненим класом, що прийматиме за T клас, який має базовий конструктор new() та є нащадком від класу BaseEntity. Реалізовуватиме методи для додавання оновлення видаленню пошуку окремої сутності чи групи сутностей.

BoardGameRepository є нащадком від BaseRepository<Boardgame> та матиме наступний додатковий функціонал : отримати ігри за фільтром, отримати ігри за постачальником.

OrderRepository є нащадком від BaseRepository<Order> та додатково матиме можливість пошуку замовлень окремого користувача.

Зобразимо отримані класи-репозиторії використовуючи UML діаграму класів(рис. 2.18).

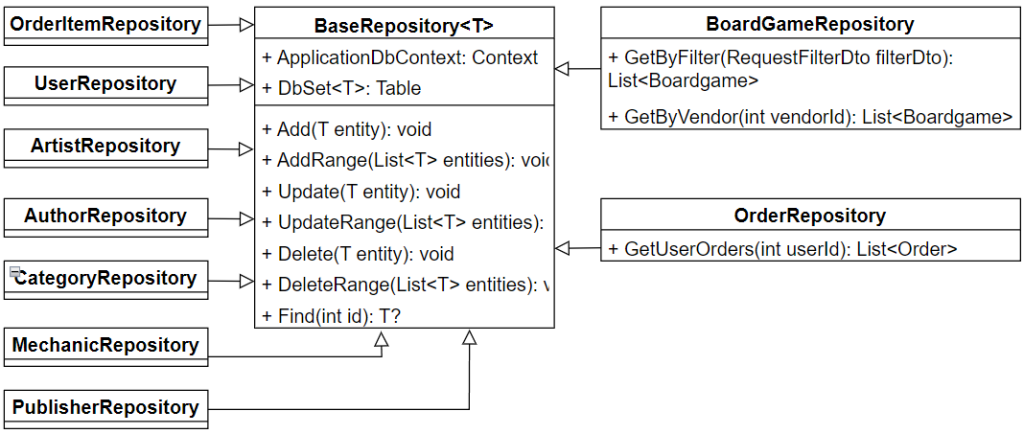

Рисунок 2.18 – UML діаграма класів-репозиторіїв шару доступу до даних

Далі розглянемо серверну логіку та взаємодію між клієнтом, сервером і шаром DAL. Для обміну ресурсами та для їх ідентифікації відповідно до REST архітектури використаємо використаємо прокол передачі даних HTTP.

Протокол передачі гіпертексту (HTTP) є основою Інтернету та використовується для передачі будь-яких ресурсів та сторінок за допомогою гіпертекстових посилань. HTTP — це протокол прикладного рівня, призначений для передачі інформації між мережевими пристроями та працює поверх інших рівнів стеку мережевих протоколів. Типовий потік через HTTP передбачає, що клієнтська машина надсилає запит серверу, який потім

надсилає відповідне повідомлення[21].

HTTP-запит — це спосіб, у який платформи інтернет-зв’язку, наприклад веб-браузери, запитують інформацію, необхідну для завантаження веб-сайту.

Кожен HTTP-запит, зроблений через Інтернет, несе в собі низку закодованих даних, які містять різні типи інформації. Типовий HTTP-запит містить:

- Тип версії HTTP
- URL
- метод HTTP
- заголовки запитів HTTP
- додаткове тіло HTTP.

Розглянемо важливі складові, що ми повинні враховувати та власноруч визначати вміст.

Метод HTTP, вказує на дію, яку HTTP-запит очікує від запитуваного сервера. Наприклад, двома найпоширенішими методами HTTP є “GET” і “POST”; запит “GET” очікує повернення інформації (зазвичай у формі веб-сайту), тоді як запит “POST” зазвичай вказує на те, що клієнт надсилає інформацію на веб-сервер (наприклад, інформацію форми, наприклад, подане ім’я користувача та пароль).

Тіло запиту – це частина, яка містить «тіло» інформації, яку запит передає. Тіло запиту HTTP містить будь-яку інформацію, яка надсилається на веб-сервер, наприклад ім’я користувача та пароль, або будь-які інші дані, введені у форму.

Відповідь HTTP – це те, що веб-клієнти (часто браузери) отримують від Інтернет-сервера у відповідь на запит HTTP. Ці відповіді передають цінну інформацію на основі запиту HTTP.

Типова відповідь HTTP містить:

- код статусу HTTP
- заголовки відповіді HTTP
- додаткове тіло HTTP

Коди статусу HTTP – це 3-значні коди, які найчастіше використовуються для вказівки, чи було успішно виконано запит HTTP. Коди стану розбиті на такі 5 блоків:

- 1xx Інформаційний
- 2xx Успіх
- 3xx Перенаправлення
- 4xx Помилка клієнта
- 5xx Помилка сервера

“xx” означає різні цифри від 00 до 99.

Успішні відповіді HTTP на запити «GET» зазвичай мають тіло, яке містить запитувану інформацію. У більшості веб-запитів це дані HTML, які веб-переглядач перетворює на веб-сторінку.

Також необхідно розглянути формат обміну інформацією між клієнтом та сервером. Для цього буде використано формат JSON та об’єкти передачі даних.

Об’єкт передачі даних (з англ. Data Transfer Object далі DTO)[22] - об’єкт,що потрібний для ізоляції даних і надсилання однієї підсистеми програми до іншої. DTO найчастіше використовуються на рівні служб у програмі з багатьма шарами для надсилання даних між підпрограмами та рівнем представлення даних. Основна перевага тут полягає в тому, що він зменшує обсяг даних, які потрібно пересилати в розподілених програмах.

Іншим використанням DTO може бути інкапсуляція параметрів для викликів методів. Це може бути корисно, якщо метод приймає більше чотирьох або п’яти параметрів.DTO не повинні містити ніякої логіки, але можуть містити механізми серіалізації/десереалізації для передавання даних.

Для реалізації серверної логіки створимо класи-контролери, що матимуть відкриті методи, кожен з яких матиме ідентифікатор у вигляді URL адресу. Ці методи будуть приймати запити від клієнта перевіряти на коректність запиту (десереалізація DTO, перевірка прав користувача), потів направлятимуть запити до репозиторіїв, на основі їх відповіді формуватимуть

відповідь клієнту, додатково створюючи потрібні DTO для клієнта.

UML діаграма послідовності (Sequence Diagram)[17] – це тип поведінкової діаграми, яка описує поведінкові аспекти складного об’єкта та враховує взаємодію об’єктів у часі.

Діаграма послідовності показує у вигляді паралельних вертикальних ліній (ліній життя) різні процеси або об’єкти, які живуть одночасно, а у вигляді горизонтальних стрілок — повідомлення, якими вони обмінюються в тому порядку, в якому вони відбуваються. Це дозволяє графічно описувати прості сценарії виконання.

- Діаграма системної послідовності повинна вказувати та показувати наступне:
- Зовнішні актори
- Повідомлення (методи), викликані цими акторами
- Повернути значення (за наявності), пов’язані з попередніми повідомленнями
- Вказівка будь-яких циклів або області ітерації

Цей тип діаграм дозволяє зобразити тимчасові особливості передачі і прийому повідомлень об'єктами, використовуючи уточнення діаграм прецедентів. Послідовні діаграми забезпечують розуміння взаємодії між шарами та об’єктами для коректного функціювання системи.

Переваги використання UML-діаграм:

- Відображення послідовності виконання операцій.
- Обмін повідомленнями між об’єктами у часі.
- Використання багатошарової архітектури.

Для демонстрації роботи програмного модулю для продажу ігор створимо UML діаграму послідовності на прикладі фільтрації ігор(рис. 2.19).

Спочатку користувач обирає параметри фільтра та відправляє їх в класові RequestFitlerDto, який включає масиви ідентифікаторів категорій, механік, авторів, акторів та виробників, що повинні містити в собі ігри. Також користувач передає при бажані певний ціновий діапазон, мінімальну та

максимальну кількість гравців, вікову категорію та номер сторінки. Потім контролер десеріалізує прикріплений до запиту файл JSON і відправляє DTO до BoardgameRepository. Він виконує запит до бази даних та повертає масив класів-сутностей List<Boardgame>. Після цього контролер створює відповідний масив класу DTO для надіслання інформації, що буде зображатись у користувача.

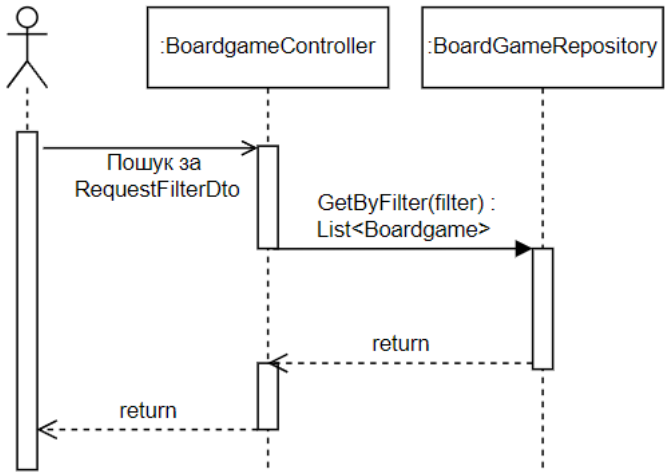

Рисунок 2.19 – UML діаграма послідовності фільтрації настільних ігор

2.3 Висновки

В даному розділі розглянуто п’ять архітектурних патернів, а саме багаторівневий патерн, клієнт-сервер патерн, REST патерн, брокер патерн та мікросервісний патерн. З них для реалізації був обраний REST патерн, так як він в ході розробці не створює надмірної комплексності та чудово підходить для розробки веб додатків.

Потім були розроблені UML діаграми для класів-сутностей, класів-репозиторіїв, були визначені типи користувачів та їх можливі дії і побудовано на основі цього UML діаграми активності. Наприкінці розділу була розроблена загальна схема взаємодії та продемонстровано на прикладі фільтрації настільних ігор та зображено з використанням UML діаграми послідовності.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛЯ ДЛЯ ПРОДАЖУ НАСТІЛЬНИХ ІГОР

3.1 Обґрунтування вибору мови програмування

Мова програмування виконує дві взаємопов'язані функції: забезпечує програмістові засоби для визначення дій, які виконує комп'ютер, і надає набір концепцій, що описують можливості програміста. З урахуванням вимог до системи можна обрати найбільш оптимальні інструменти для її реалізації.

C++ — це кросплатформна мова програмування для розробки високопродуктивних програм. Б'ярн Страуструп створив C++ як розширення мови програмування C. C++ дає програмістам велику владу над пам'яттю та ресурсами системи. C++ є однією з найпоширеніших мов програмування у світі. Сучасні операційні системи, графічні інтерфейси користувача та вбудовані системи містять його. Це об’єктно-орієнтована мова програмування, яка пропонує програмам логічну структуру та дозволяє повторно використовувати код, зменшуючи витрати на розробку. C++ — це портативна мова програмування, яка може використовуватися для створення програм, що працюють на різних системах.[24].

C++ було розроблено з урахуванням системного програмування та вбудованого програмного забезпечення з обмеженими ресурсами та великих систем, з продуктивністю, ефективністю та гнучкістю використання як основними моментами його дизайну[24].

C# — це мова програмування загального призначення високого рівня, яка підтримує кілька парадигм. C# охоплює статичну типізацію, сильну типізацію, лексичну область видимості, імперативну, декларативну, функціональну, загальну об’єктно-орієнтоване (на основі класів) і компонентно-орієнтоване програмування дисципліни[25].

Мова C# була створена Microsoft як частина програми .NET, яку координує Андерс Хейлсберг і його команда, і мова була авторизована

Європейською асоціацією виробників комп'ютерів (ECMA) і Міжнародною організацією стандартів (ISO) (ISO). Поточна версія C# є однією з мов для спільної мовної інфраструктури(Common Language Infrastructure, CLI).

Стандарт Ecma перераховує такі цілі дизайну для C#:

- Ця мова має бути простою, сучасною, об'єктно-орієнтованою мовою програмування загального призначення;
- Мова та її реалізації повинні забезпечувати підтримку принципів розробки програмного забезпечення, таких як сильна перевірка типів, перевірка меж масиву, виявлення спроб використання неініціалізованих змінних та автоматичне збирання сміття. Важливими є стійкість, міцність, довговічність програмного забезпечення і продуктивність програміста;
- Мова призначена для використання в розробці програмних компонентів, придатних для розгортання в розподілених середовищах;
- Портативність дуже важлива для вихідного коду та програмістів, особливо тих, хто вже знайомий із C та C++;
- Важлива підтримка інтернаціоналізації;
- C# призначений для написання додатків як для розміщених, так і для вбудованих систем, починаючи від дуже великих, які використовують складні операційні системи, і закінчуючи дуже маленькими, що мають спеціальні функції;
- Незважаючи на те, що додатки C# мають бути економічними щодо вимог до пам'яті та процесорної потужності, ця мова не була призначена для прямої конкуренції за продуктивністю та розміром із C або мовою асемблера.

Обидві мови програмування також підтримують роботу з потоками, з файловою системою, механізми синхронізації та блокування ресурсів і мають широкий бібліотечний функціонал. Порівняння мови програмування C++ та C# наведено в таблиці 3.1.

Таблиця 3.1 — Порівняння основних характеристик мов C++ і C#

Характеристика	C#	C++
Бінарні файли	Компілюється напряму в бінарний файл, малий розмір файлу.	Компілюється в проміжний код(CIL) з якого уже в бінарний, більший розмір файлу.
Типізація	Сувора типізація, потрібно вказувати тип даних при створенні змінної.	Гнучка, використовується як слабка так і сувора типізація.
Керування пам’яттю	Автоматичне керування пам’яттю (збирання сміття). Зменшує ризик витоків, але може впливати на продуктивність.	Здійснюється вручну. Якщо розробник створює об’єкт, він несе відповідальність за знищення цього об’єкта після завершення його виконання.
Попередження компілятора	Скомпілює будь-який синтаксично правильний код. Код може пошкодити операційну систему та обладнання.	C# надає попередження компілятора, якщо код може спричинити проблеми.

Тому код написаний на мові C# є зрозумілим і читабельним, що особливо важливо для командної розробки та підтримує код у довгостроковій перспективі. У той час як C++ має складніший синтаксис, але надає більшу гнучкість для досвідчених розробників.

Автоматичне керування пам’яттю в C#, з одного боку, полегшує життя розробника, але з іншого, може внести невизначеність у продуктивність. На противагу йому, ручне керування пам’яттю в C++ надає повний контроль, але, звичайно, вимагає відповідальності й важливості, щоб уникнути витоків.

C-Sharp виграє в галузях, де зручність і швидка розробка мають

першорядне значення. У веб-розробці з використанням ASP.NET, C# надає зручні інструменти для створення ефективних і масштабованих веб-додатків. Завдяки сучасним структурам даних і можливостям асинхронного програмування, C# також проявляє себе добре в обробці великої кількості одночасних запитів[26].

C++ же гарний у тих галузях, де критична висока продуктивність і управління ресурсами. Ігрова індустрія – один із яскравих напрямків, де цю мову використовують у розробці рушіїв, таких як Unreal Engine, забезпечуючи максимально можливу продуктивність для вимогливих до ресурсів ігор[24].

Виходячи з особливостей цих мов програмування та з розробленої структури програмного продукту для створення програмного модулю для продажу настільних ігор було обрано мову програмування C#, так як вона чудово підходить для ефективної розробки веб-додатків, зручна для реалізації різноманітних рішень без необхідності в низькорівневій взаємодії з апаратним забезпеченням.

Після вибору мови програмування розглянемо доступні фреймворки, що спрощують розробку різноманітних аспектів програми, а саме : Entity Framework Core для взаємодії з базою даних, ASP.NET Core для створення веб-додатку та Blazor для розробки клієнтської частини.

Entity Framework (EF) Core — це легка, розширювана кросплатформна версія популярної технології доступу до даних Entity Framework із відкритим кодом[27]. EF Core може служити об’єктно-реляційним відображенням (O/RM), який:

- Дозволяє розробникам .NET працювати з базою даних за допомогою об’єктів .NET.
- Усуває потребу в більшості коду доступу до даних, який зазвичай потрібно писати.

У EF Core доступ до даних здійснюється за допомогою моделі. Модель складається з класів сутностей і об’єкта контексту, який представляє сеанс із базою даних. Об’єкт контексту дозволяє запитувати та зберігати дані.

EF підтримує такі підходи до розробки моделей:

- Створіть модель з уже створеної бази даних.
- Ручне кодування моделі відповідно до бази даних.

Після створення моделі використовуйте EF Migrations, щоб створити базу даних із моделі. Міграції дозволяють розвивати базу даних у міру зміни моделі. Екземпляри класів сутностей витягуються з бази даних за допомогою мови інтегрованого запиту (LINQ). Дані створюються, видаляються та змінюються в базі даних за допомогою екземплярів класів сутностей.

ASP.NET Core[28] — це оновлений дизайн ASP.NET 4.x, включаючи зміни в архітектурі, що призвело до створення компактнішої та більш модульної структури.

ASP.NET Core надає такі переваги:

- Уніфікований інструментарій створення веб-інтерфейсу користувача та web API.
- Спроектовано для тестування.
- Razor Pages робить кодування сценаріїв, орієнтованих на сторінку, простішим і продуктивнішим.
- Blazor дозволяє використовувати C# у браузері разом із JavaScript. Спільний доступ до логіки додатків на стороні сервера та клієнта, написаних у .NET.
- Можливість розробки та запуску в Windows, macOS і Linux.
- З відкритим вихідним кодом і орієнтований на спільноту.
- Інтеграція сучасних клієнтських фреймворків і робочих процесів розробки.
- Підтримка розміщення служб Remote Procedure Call (RPC) за допомогою gRPC.
- Система налаштування на основі середовища, готова до роботи в хмарі.
- Вбудоване впровадження залежностей.
- Легкий, високопродуктивний і модульний конвеєр запитів HTTP.

- Паралельне створення версій.
- Інструменти, які спрощують сучасну веб-розробку.

ASP.NET Core MVC надає функції для створення Web-API і веб-програм:

- Шаблон Model-View-Controller (MVC) допомагає зробити ваші Web-API та веб-програми придатними для перевірки.
- Razor Pages — це модель програмування на основі сторінок, яка робить створення веб-інтерфейсу користувача продуктивнішим.
- Розмітка Razor забезпечує продуктивний синтаксис для переглядів Razor Pages і MVC.
- Помічники тегів дозволяють серверному коду брати участь у створенні та відображенні HTML-елементів у файлах Razor.
- Вбудована підтримка багатьох форматів даних і узгодження вмісту дозволяє вашим Web-API охоплювати широкий спектр клієнтів, включаючи браузери та мобільні пристрої.
- Зв’язування моделі автоматично відображає дані з HTTP-запитів на параметри методу дії.
- Перевірка моделі автоматично виконує перевірку на стороні клієнта та на стороні сервера.

ASP.NET Core використовує Blazor для створення багатого інтерактивного веб-інтерфейсу користувача.

Blazor — це фронтенд фреймворк .NET, який підтримує як рендеринг на стороні сервера, так і інтерактивність клієнта в одній моделі програмування[29]:

- Спільне використання логіки програм на стороні сервера та клієнта, написаною в .NET.
- Підтримка HTML і CSS для створення інтерфейсу користувача з підтримкою широкого спектру браузерів, у тому числі мобільних.

Використання .NET для веб-розробки на стороні клієнта дає такі переваги:

- Написання код на C# підвищує продуктивність у розробці та

обслуговуванні програм.

- Використання існуючої екосистему бібліотек .NET.
- Використання переваг продуктивності, надійності та безпеки .NET.

Програми Blazor засновані на компонентах. Компонент у Blazor — це елемент інтерфейсу користувача, наприклад сторінка, діалогове вікно або форма введення даних. Клас компонента зазвичай записується у формі сторінки розмітки Razor із розширенням файлу .razor[29].

3.2 Реалізація програмного модуля для продажу настільних ігор

Кожний програмний продукт, що будується з використанням мови програмування C# складається з рішень та проектів[30].У логічному сенсі проект містить усі файли, скомпільовані у виконуваний файл, бібліотеку або веб-сайт. Ці файли можуть містити вихідний код, значки, зображення, файли даних тощо. Проект міститься в рішенні. Незважаючи на свою назву, рішення не є “відповіддю”. Це просто контейнер для одного або кількох пов’язаних проектів разом із інформацією про збірку, параметрами та будь-якими різними файлами, які не пов’язані з конкретним проектом.

Виходячи з структури програми, що була розроблена в попередньому розділі, спочатку необхідно реалізувати DAL, так як він необхідний для коректної роботи інших шарів. Тому створимо проект бібліотеку класів з назвою BoadGameShop.DAL та рішення BoardGameShop. Наступним кроком створимо додаткові папки : EfStructures міститиме файли потрібні для роботи фреймворка EfCore в цілому; Entities міститиме класи-сутності та підпапку Configuration, що міститиме класи-конфігуратори сутностей; папка Repositories міститиме класи-репозиторії.

Також рішення BoadGameShop.DAL міститиме файл GlobalUsing.cs, що міститиме імпортування всіх необхідних просторів імен. Наступним кроком буде створення класів-сутностей та класів-конфігураторів для налаштування взаємозв’язків між сутностями.

Клас BaseEntity міститиме публічний параметр id ціле-чисельного типу int та публічний параметр TimeStamp у вигляді масиву байтів. Для ідентифікатора будуть використані атрибути Key та DatabaseGenerated(DatabaseGeneratedOption.Identity), які вказуватимуть, що параметр є ключем значення для якого буде генерувати СУБД як для поля-ідентифікатора. TimeStamp матиме відповідний атрибут Timestamp, що вкаже фреймворку необхідність налаштувати СУБД для перевірки паралелізму саме за допомогою цього поля(рис. 3.1).

```
namespace BoardGameShop.DAL.Entities
{
    10 references
    public class BaseEntity
    {
        [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        55 references
        public int Id { get; set; }
        [Timestamp]
        16 references
        public byte[] TimeStamp { get; set; }
    }
}
```

Рисунок 3.1 - лістинг коду класу BaseEntity

Клас Boardgame є дочірнім від класу BaseEntity та міститиме наступні поля :

- рядок Name, максимальна кількість символів 30, є обов’язковим полем;
- рядок Description, максимальна довжина 1000 символів, необов’язкове поле;
- Quantity з типом int за значення за замовчуванням 0. Відповідає за кількість гри доступної для продажу;
- MinPlayer та MaxPlayer з типом byte?, можуть не мати значення, відповідно це мінімальна та максимальна кількість гравців;
- MinPlayTime та MaxPlayTime з типом int?, можуть не мати значення, відповідно це мінімальний та максимальний час повної партії, вказується в хвилинах;

- Логічне(булеве) поле IsDeleted, що потрібне для реалізації м’якого видалення. Тобто щоб гра не відображалась у клієнтів, але існувала в базі даних(наприклад для коретного зберігання замовлень);
- Age з типом даних переліку всіх вікових категорій AgeEnum?, може бути відсутнім.
- Обов’язкове десяткове поле FullPrice, зберігатиме повну ціну гри;
- Discount з типом byte?, може не мати значення, зберігатиме скидку на гру;
- Обов’язкове цілочисельні поля PublisherId і VendorId, що слугують зовнішніми ключами для зв’язку з відповідно сутністю виробника та сутністю постачальника.

Також цей клас матиме додаткові поля, що слугуватимуть не для представлення певної характеристики настільної гри, а для доступу до інших сутностей через сутність настільної гри. Також в кожного такого поля буде вказаних атрибут, що вказуватиме на таке ж поле у сутності, до якої вказується зв’язок(рис. 3.2).

```
public List<Author> Authors { get; set; }
[InverseProperty(nameof(BoardgameToAuthor.BoardgameNavigation))]
7 references
public List<BoardgameToAuthor> BoardgameAuthors { get; set; }

[InverseProperty(nameof(Artist.Boardgames))]
8 references
public List<Artist> Artists { get; set; }
[InverseProperty(nameof(BoardgameToArtist.BoardgameNavigation))]
7 references
public List<BoardgameToArtist> BoardgameArtists { get; set; }

[Required]
6 references
public int PublisherId { get; set; }
[Required, ForeignKey(nameof(PublisherId))]
[InverseProperty(nameof(Publisher.BoardGames))]
12 references
public Publisher PublisherNavigation { get; set; }
4 references
public int VendorId { get; set; }
[InverseProperty(nameof(Vendor.Boardgames))]
5 references
public Vendor VendorNavigation { get; set; }
[InverseProperty(nameof(OrderItem.BoardgameNavigation))]
4 references
public List<OrderItem>? OrderItems { get; set; }
```

Рисунок 3.2 - фрагмент лістингу коду класу Boardgame

Класи-сутності автора(рис. 3.3) та художника(рис. 3.4) міститимуть їх повні імена з обмеженням в 100 символів і навігаційні поля до відповідних їм ігор та відображувачів відношення Б:Б, також класи є дочірніми від BaseEntity :

```
public class Author : Creator
{
    [InverseProperty(nameof(Boardgame.Authors))]
    2 references
    public List<Boardgame> Boardgames { get; set; }
    [InverseProperty(nameof(BoardgameToAuthor.AuthorNavigation))]
    1 reference
    public List<BoardgameToAuthor> BoardgameToAuthors { get; set; }
}
```

Рисунок 3.3 - лістинг коду класу Author

```
public class Artist : Creator
{
    [InverseProperty(nameof(Boardgame.Artists))]
    2 references
    public List<Boardgame> Boardgames { get; set; }
    [InverseProperty(nameof(BoardgameToArtist.ArtistNavigation))]
    1 reference
    public List<BoardgameToArtist> BoardgameArtists { get; set; }
}
```

Рисунок 3.4 - лістинг коду класу Artist

Класи-сутності категорії(рис. 3.5) та механіки(рис. 3.6) міститимуть рядкове обов’язкове поле імені та обмеження в 20 символів і необов’язкове рядкове поле опису, поля навігації до відповідних їм ігор та відображувачів відношення Б:Б. Дочірні від класу BaseEntity :

```
public class Category : BaseEntity
{
    [Required, StringLength(20)]
    5 references
    public string Name { get; set; } = string.Empty;
    [StringLength(200)]
    0 references
    public string? Description { get; set; } = string.Empty;
    [InverseProperty(nameof(Boardgame.Categories))]
    2 references
    public List<Boardgame> Boardgames { get; set; }
    [InverseProperty(nameof(BoardgameToCategory.CategoryNavigation))]
    1 reference
    public List<BoardgameToCategory> BoardgameCategories { get; set; }
}
```

Рисунок 3.5 - лістинг коду класу Category

```
public class Mechanic : BaseEntity
{
    [Required, StringLength(20)]
    5 references
    public string Name { get; set; } = string.Empty;
    [StringLength(200)]
    0 references
    public string? Description { get; set; } = string.Empty;
    [InverseProperty(nameof(Boardgame.Mechanics))]
    2 references
    public List<Boardgame> Boardgames { get; set; }
    [InverseProperty(nameof(BoardgameToMechanic.MechanicNavigation))]
    1 reference
    public List<BoardgameToMechanic> BoardgameMechanics { get; set; }
}
```

Рисунок 3.6 - лістинг коду класу Mechanic

Класи-сутності виробника та постачальника(рис. 3.7) міститимуть їх повні імена з обмеженням в 50 символів і навігаційні поля до відповідних їм ігор, також постачальник міститиме навігаційне поле до відповідних замовлень. Класи є дочірніми від BaseEntity :

```
public class Vendor : BaseEntity
{
    2 references
    public string Name { get; set; } = string.Empty;
    [InverseProperty(nameof(Boardgame.VendorNavigation))]
    2 references
    public List<Boardgame> Boardgames { get; set; }
    1 reference
    public List<Order> Orders { get; set; }
}
```

Рисунок 3.7 - лістинг коду класу Vendor

Клас користувача міститиме три рядкових необов’язкових поля для повного імені з обмеженням 50 символів, обов’язковий рядок логіну; поле для ролі з значенням за замовчуванням Role.User; рядок для адресу електронної почти та номеру телефону; навігаційне поле до замовлень та хеш пароля у вигляді масива байтів(рис. 3.8). Є дочірнім від BaseEntity :

```
public Role UserRole { get; set; } = Role.User;
[StringLength(50)]
[Required]
8 references
public string Username { get; set; } = string.Empty;
[StringLength(50)]
2 references
public string Email { get; set; } = string.Empty;
[StringLength(50)]
2 references
public string? FirstName { get; set; }
[StringLength(50)]
2 references
public string? SecondName { get; set; }
[StringLength(50)]
2 references
public string? LastName { get; set; }
[Required]
4 references
public byte[] PasswordHash { get; set; }
[Required]
4 references
public byte[] PasswordSalt { get; set; }
[StringLength(20)]
2 references
public string PhoneNumber { get; set; } = string.Empty;
1 reference
public List<Order> Orders { get; set; }
```

Рисунок 3.8 - лістинг коду класу User

Клас замовлення міститиме обов'язкові цілочисельні ідентифікатори користувача та постачальника з навігаційними полями; дату створення, оплати та завершення, дані користувача, статус замовлення і оплати(рис. 3.9).

```
[Required]
3 references
public int UserId { get; set; }
[Required]
3 references
public DateTime CreationDate { get; set; }
3 references
public DateTime? CompletionDate { get; set; }
1 reference
public DateTime? PaymentDate { get; set; }
[Required]
2 references
public int VendorId { get; set; }
3 references
public Vendor? VendorNavigation { get; set; }
2 references
public string? Firstname { get; set; }
2 references
public string? Lastname { get; set; }
2 references
public string? Secondname { get; set; }
2 references
public string? PhoneNumber { get; set; }
2 references
public string? DeliveryAddress { get; set; }
2 references
public string? MessageFromUser { get; set; }
[Required]
3 references
public OrderStatus OrderStatus { get; set; }
[Required]
3 references
public PaymentStatus PaymentStatus { get; set; }
[InverseProperty(nameof(OrderItem.OrderNavigation))]
3 references
public List<OrderItem> OrderItems { get; set; }
```

Рисунок 3.9 - лістинг коду класу Order

Далі створимо класи, що необхідні для реалізації відношень багато до багатьох. Ці класи не будуть мати батьківського класу, матимуть два обов'язкових цілочисельні поля ідентифікатори сутностей, що входять в відношення багато до багатьох. Міститимуть навігаційні поля до відповідних класів-сутностей, де використовуватиметься як зовнішній ключ відповідний ідентифікатор. Первинний ключ складений, складається з двох ідентифікаторів, налаштовуватиметься це в конфігурації класів-сутностей. Для прикладу наведено лістинг коду класу BoardgameToMechanic на рисунку 3.10; класи VendorEmloyee, BoardgameToCategory, BoardgameToAuthor і BoardgameToArtist мають такий же ш вигляд :

```
public class BoardgameToMechanic
{
    4 references
    public int BoardgameId { get; set; }
    [ForeignKey(nameof(BoardgameId)), Required]
    2 references
    public Boardgame? BoardgameNavigation { get; set; }
    5 references
    public int MechanicId { get; set; }
    [ForeignKey(nameof(MechanicId)), Required]
    2 references
    public Mechanic? MechanicNavigation { get; set; }
}
```

Рисунок 3.10 - лістинг коду класу BoardgameToMechanic

Клас OrderItem крім ідентифікаторів замовлень та ігор з відповідними навігаційними полями також міститиме кількість замовлених екземплярів цього товару в конкретному замовлені та ціну одного предмета в момент створення замовлення з урахуванням скидки(рис. 3.11).

```
[EntityTypeConfiguration(typeof(OrderItemConfiguration))]
public class OrderItem
{
    [Required]
    public int OrderId { get; set; }
    [Required]
    public int ItemId { get; set; }
    [Required]
    public int ItemCost { get; set; }
    [Required]
    public int Qty { get; set; }
    [Required]
    [ForeignKey(nameof(OrderId))]
    public Order OrderNavigation { get; set; }
    [ForeignKey(nameof(ItemId))]
    public Boardgame BoardgameNavigation { get; set; }

}
```

Рисунок 3.11 - лістинг коду класу OrderItem

Наступним кроком створимо класи-конфігуратори для додаткового налаштування класів-сутностей та створеної бази даних за допомогою EFCore.

У BoardgameConfiguration вкажимо назву таблиці для створення; вкажимо на поле первинний ключ; налаштуємо відношення до класів виробника та постачальника як один до багатьох і вкажемо зовнішні ключі; проведемо створення зв’язку багато до багатьох через додатковий клас, вкажемо відповідні навігаційні поля та створимо складений ключ для додаткових класів(таблиць).Налаштування багато для багатьох буде наведено лише для Boardgame - BoardgameToMechanic - Mechanic на рисункові 3.12 :

```
public class BoardgameConfiguration : IEntityTypeConfiguration<Boardgame>
{
    public void Configure(EntityTypeBuilder<Boardgame> builder)
    {
        builder.ToTable("BoardGames");
        builder.HasKey(x => x.Id);
        builder.HasOne(x => x.PublisherNavigation)
            .WithMany(y => y.BoardGames)
            .IsRequired()
            .HasForeignKey(x => x.PublisherId);

        builder.HasOne(x => x.VendorNavigation)
            .WithMany(y => y.Boardgames)
            .HasForeignKey(x => x.VendorId).IsRequired();

        builder.HasMany(x => x.Mechanics)
            .WithMany(y => y.Boardgames)
            .UsingEntity<BoardgameToMechanic>(
            j => j.HasOne(cd => cd.MechanicNavigation)
            .WithMany(cd => cd.BoardgameMechanics)
            .HasForeignKey(cd => cd.MechanicId)
            .OnDelete(DeleteBehavior.Cascade),
            j => j.HasOne(cd => cd.BoardgameNavigation)
            .WithMany(cd => cd.BoardgameMechanics)
            .HasForeignKey(cd => cd.BoardgameId)
            .OnDelete(DeleteBehavior.Cascade),
            j => j.HasKey(cd => new { cd.MechanicId, cd.BoardgameId }));
```

Рисунок 3.12 - фрагмент лістингу коду класу BoardgameConfiguration

Конфігурація замовлень(рис. 3.13) створюватиме відношення один до багатьох предметів замовлення; вкаже на значення за замовчуванням для полів-статусів та необхідність при додаванні нового запису в таблиці генерувати для поля CreationDate значення, що відповідатиме даті додавання замовлення. Конфігурація предмета замовлення (рис. 3.14) указуватиме на складений ключ, на створення відношення з класом замовлень і встановлюватиме правила видалення.

```
internal class OrderConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.HasMany(c => c.OrderItems)
            .WithOne(c => c.OrderNavigation)
            .HasForeignKey(c => c.OrderId);
        builder.Property(o => o.CreationDate).ValueGeneratedOnAdd().HasDefaultValueSql("getdate()");
        builder.Property(o => o.OrderStatus).HasDefaultValue(OrderStatus.Created);
        builder.Property(o => o.PaymentStatus).HasDefaultValue(PaymentStatus.NotPaid);
    }
}
```

Рисунок 3.13 - лістинг коду класу OrderConfiguration

```
internal class OrderItemConfiguration : IEntityTypeConfiguration<OrderItem>
{
    public void Configure(EntityTypeBuilder<OrderItem> builder)
    {
        builder.HasKey(i => new { i.OrderId, i.ItemId });
        builder.HasOne(i => i.BoardgameNavigation)
            .WithMany(g => g.OrderItems)
            .HasForeignKey(i => i.ItemId).OnDelete(DeleteBehavior.ClientCascade);
    }
}
```

Рисунок 3.14 - лістинг коду класу OrderItemConfiguration

Конфігурація постачальника(рис. 3.15) та користувача(рис. 3.16) будуть лише мати налаштування з класом замовлення для установки зв’язку один до багатьох.

```
internal class VendorConfiguration : IEntityTypeConfiguration<Vendor>
{
    public void Configure(EntityTypeBuilder<Vendor> builder)
    {
        builder.HasMany(c => c.Orders)
            .WithOne(o => o.VendorNavigation)
            .HasForeignKey(c => c.VendorId);
    }
}
```

Рисунок 3.15 - лістинг коду класу VendorConfiguration

```
internal class UserConfiguration : IEntityTypeConfiguration<User>
{
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.HasMany(u => u.Orders)
            .WithOne()
            .HasForeignKey(o => o.UserId);
    }
}
```

Рисунок 3.16 - лістинг коду класу UserConfiguration

Наступним кроком при використанні EFCore - створення класу дочірнього від DbContext. Створимо клас ApplicationDbContext в папці EfStructures. Через цей клас і відбуватиметься взаємодія з базою даних. Кожен клас, що повинен бути представлений в базі даних указується як заповнювач T для публічного поля типу DbSet<T> *PropertyName*. Таким чином зареєструємо всі класи-сутності та додаткові класи для створення відповідних таблиць в базі даних (рис. 3.17).

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    public DbSet<Boardgame> BoardGames { get; set; }
    public DbSet<BoardgameToCategory> BoardgameToCategory { get; set; }
    public DbSet<BoardgameToMechanic> BoardgameToMechanic { get; set; }
    public DbSet<BoardgameToArtist> BoardgameToArtist { get; set; }
    public DbSet<BoardgameToAuthor> BoardgameToAuthor { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Author> Authors { get; set; }
    public DbSet<Artist> Artists { get; set; }
    public DbSet<Mechanic> Mechanics { get; set; }
    public DbSet<Publisher> Publishers { get; set; }
    public DbSet<User> Users { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderItem> OrderItems { get; set; }
    public DbSet<Vendor> Vendors { get; set; }
    public DbSet<VendorEmployee> VendorEmployees { get; set; }
```

Рисунок 3.17 - реєстрація класів-сутностей в класові ApplicationDbContext

Налаштування класів і таблиць розташовується в перевизначеному методі OnModelCreating(ModelBuilder modelBuilder). За допомогою modelBulder і відбувається вказання на класи конфігурації (рис. 3.18), тому передамо потрібні сутності з цього об’єкта на налаштування в класи-конфігуратори.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    new BoardgameConfiguration().Configure(modelBuilder.Entity<Boardgame>());
    new OrderConfiguration().Configure(modelBuilder.Entity<Order>());
    new OrderItemConfiguration().Configure(modelBuilder.Entity<OrderItem>());
    new VendorConfiguration().Configure(modelBuilder.Entity<Vendor>());
    new UserConfiguration().Configure(modelBuilder.Entity<User>());
    new OrderItemConfiguration().Configure(modelBuilder.Entity<OrderItem>());
```

Рисунок 3.18 - використання класів конфігурації в методові OnModelCreating класу ApplicationDbContext

Останнім кроком для коректної роботи EFCore потрібно створити клас, що буде мати налаштування для створення DbContext класа : ApplicationDbContextFactory(рис. 3.19). В ньому буде лише один метод, що вказуватиме параметри для підключення та повертатиме створений DbContext об‘єкт.

```
public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
{
    public ApplicationDbContext CreateDbContext(string[] args)
    {
        var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
        var connectionString = @"server=.,4433;Database=BoardGameDb;User Id=sa;Password=Pasd55152;Encrypt=false";
        optionsBuilder.UseSqlServer(connectionString);
        return new ApplicationDbContext(optionsBuilder.Options);
    }
}
```

Рисунок 3.19 - лістинг коду класу ApplicationDbContextFactory

Далі необхідно встановити утиліту для терміналу, що дозволить взаємодіяти з схемою бази даних з отриманням налаштувань від класу ApplicationDbContextFactory. Для встановки утиліти потрібно виконати наступну команду в терміналі :

dotnet tool install --global dotnet-ef

Після цього створюємо міграцію з іменем Inititial та запускаємо виконання міграції, для цього виконаємо наступні команди в терміналі :

dotnet ef migration add Initial

dotnet ef database update

Далі створимо BaseRepository клас, що буде основою для інших репозиторіїв, які уже будуть дороблятись у випадку отримати специфічну інформацію з бази даних. Міститиме поле Context та поле Table типу DbSet<T>, де T це якийсь клас-нащадок від BaseEntity. Також методи, що змінюють записи, прийматимуть відповідний об’єкт типу T та булеве значення persiste = true. Для збереження змін в базі даних EFCore потребує після виконання дій виклик методу DbContext.SaveChanges(), що обновить значення в базі даних(рис. 3.20). До цього всі дії виконуватимуться з копією даних, а не з оригіналом. Таким чином отримуємо захист оригінальних даних до успішного виконання всієї групи дій.

```
public class BaseRepository<T> : IBaseRepo<T> where T : BaseEntity, new()
{
    private bool disposedValue;

    public BaseRepository(ApplicationDbContext applicationDbContext)...

    public ApplicationDbContext Context { get; }
    public DbSet<T> Table { get; }
    public async Task<int> Add(T entity, bool persiste = true)
    {
        await Table.AddAsync(entity);
        return persiste ? await SaveChanges() : 0;
    }

    public async Task<int> AddRange(IEnumerable<T> entities, bool persiste = true)...

    public async Task<int> Delete(int id, byte[] timeStamp, bool persiste = true)...

    public async void Delete(T entity, bool persiste = true)
    {
        Table.Remove(entity);
        if (persiste)
            await SaveChanges();
        return;
    }

    public async void DeleteRange(IEnumerable<T> entities, bool persiste = true)...
    public async Task<IEnumerable<T>> GetAll()...

    public async Task<T?> Find(int id)...
    public async Task<int> Update(T entity, bool persiste = true)...
    public async Task<int> UpdateRange(IEnumerable<T> entities, bool persiste = true)...
    public async Task<int> SaveChanges()...
```

Рисунок 3.20 - лістинг коду класу BaseRepository

Перейдемо до створення серверної частини. Створимо ASP.NET Core Web Api шаблон проекту та назвемо його BoardGameShop.Api. У проекті матимемо папку Controllers з контролерами, статичний клас DtoConversions, що міститиме методи для конвертації класа-сутності(сутностей) в певний Dto клас. Також проект містить файл Program.cs необхідний для конфігурації серверу та файд GlobalUsing.cs для глобальних імпортів бібліотек.

Перейдемо до створення контролерів. Першим створимо контролер для настільних ігор. Будь-який контролер повинен бути нащадком від класу ControllerBase і атрибут Route, що вказуватиме формат, за яким відбувається ідентифікація цього контролера і його ресурсів(рис. 3.21). Контролер міститиме два метода, що повертатимуть тип IActionResult, який містить в собі необов’язково серіалізований об’єкт відповідь сервера та обов’язково статус-

код виконання операції. В конструкторі клас приймає репозиторій, що використовуватиметься для подальшого надсилання запитів до бази даних.

```
[Route("api/[controller]")]
[ApiController]
public class BoardGameController : ControllerBase
{
    private readonly IBoardGameRepository gameRepository;

    public BoardGameController(IBoardGameRepository gameRepository)
    {
        this.gameRepository = gameRepository;
    }
```

Рисунок 3.21 - конструктор і поля класу BoardGameController

Для отримання всіх ігор доступних для покупки клієнт робить запит на BoadGameController та відправляє параметри фільтрації та пагінації. Після успішного серіалізації фільтру викликаємо відповідний метод у об’єкта репозиторія і отримуємо у відповідь масив ігор та загальну кількість можливих сторінок при переданій кількості ігор на одну сторінку. Після проводимо конвертацію масива ігор в gamesDto. Якщо в результаті запиту ігор не знайдено надсилаємо відповідну помилку на клієнт, інакше повертаємо статус код 200 з серіалізованими іграми(рис. 3.22).

```
[Produces("application/json")]
[HttpPost("Filter")]
public async Task<IActionResult> FindAll([FromBody] RequestFilterDto filter)
{
    (var games, int totalPage) = await gameRepository.GetByFilter(filter);
    var gamesDto = games.ConvertToDto();
    if (gamesDto == null)
    {
        return NotFound();
    }
    var result = new ProductsPageDto { Products = gamesDto, TotalPages = totalPage };
    return Ok(result);
}
```

Рисунок 3.22 - лістинг методу FindAll

Далі створимо метод для отримання деталей про окрему гру(рис. 3.23). Для цього клієнт буде передавати ідентифікатор гри в URL адрес. Викликаємо метод у репозиторія, якщо не знаходимо повертаємо помилку, інакше ж відправляємо шуканий об’єкт перетворивши в відповідне Dto.

```
[Produces("application/json")]
[HttpGet("{id:int}")]
public async Task<IActionResult> FindSingle(int id)
{
    var game = await gameRepository.GetById(id);
    if (game == null)
        return NotFound();
    return Ok(game.ConvertToDto());
}
```

Рисунок 3.23 - лістинг методу FindSingle

Метод Find від базового репозиторія в даному випадку не підходить, так як він повертає лише клас-сутність без заповнення полів навігації. Для цього потрібно при запитові до DbSet<T> додатково вказати які повинні класи також потрібно отримати з бази даних за допомогою метода Include, після чого EFCore створить запит для отримання даних про клас-сутність, потім створить додаткові запити на отримання даних з інших таблиць використовуючи зовнішні ключі, що вказані при конфігурації відношень(рис. 3.24).

```
public async Task<Boardgame> GetById(int id)
{
    var game = await Table.Include(cd => cd.PublisherNavigation)
        .Include(cd => cd.VendorNavigation)
        .Include(cd => cd.Mechanics)
        .Include(cd => cd.Categories)
        .Include(cd => cd.Artists)
        .Include(cd => cd.Authors)
        .FirstAsync(g => g.Id == id);
    return game ?? new Boardgame();
}
```

Рисунок 3.24 - лістинг методу GetById

Для подальшого функціоналу необхідна перевірка прав користувача на виконання дій, тому спочатку необхідно реалізувати авторизацію та аутентифікацію клієнта. Але відправляти при кожному запиті до сервера логін і пароль є небезпечним, також це є неоптимальне використання трафіку; перевірка потім валідності отриманих даних є затратним по обчислювальним ресурсам. Для вирішення цієї проблеми після проведення аутентифікації буде створений токен, що зберігатиме дані-стан необхідні для клієнтської і/або

серверної частини та є важким для підробки. Потім цей токен буде зберігатись в локальній пам’яті браузера і буде добавлений до кожного запиту від клієнта.

Тому розглянемо один з варіантів такого токену - JWT токен. Веб-токен JSON (англ. JSON Web Token, далі JWT) — це відкритий стандарт (RFC 7519), який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами як об’єкт JSON. Цю інформацію можна перевірити та довіряти їй, оскільки вона має цифровий підпис. JWT можна підписати за допомогою секрету (з алгоритмом HMAC) або пари відкритих/приватних ключів за допомогою RSA або ECDSA[31]. Стан передається за допомогою пар ключ:значення, ці пари також називають клеймами.

Хоча JWT можна зашифрувати, щоб також забезпечити секретність між сторонами, ми зосередимося на підписаних токенах. Підписані токени можуть підтвердити цілісність клеймів, що містяться в ньому, тоді як зашифровані токени приховують ці клейми від інших сторін. Коли токени підписуються за допомогою пар відкритих/приватних ключів, підпис також засвідчує, що лише сторона, яка володіє закритим ключем, підписала його.

У своїй компактній формі JWT токени складаються з трьох частин, розділених крапками (.):

- Заголовок
- Корисне навантаження
- Підпис

Таким чином, JWT зазвичай має вигляд xxxxx.yyyyy.zzzzz

Заголовок зазвичай складається з двох частин: типу токену, яким є JWT, і використовуваного алгоритму підпису, наприклад HMAC SHA256 або RSA. Тоді цей JSON кодується Base64Url, щоб сформувати першу частину JWT.

Друга частина токена — це корисне навантаження, яке містить претензії. Претензії(клейми) – це дані про сутність (як правило, користувача) і додаткові дані. Існує три види клеймів :

- Зареєстровані клейми - набір попередньо визначених клеймів, які не є обов’язковими, але рекомендовані. Деякі з них: exp (термін дії), sub

(тема), aud (аудиторія);

- Публічні клейми - клейми, які можуть бути визначені за бажанням тими, хто використовує JWT. Але щоб уникнути колізії, їх слід визначити в реєстрі веб-токенів IANA JSON або визначити як URI, який містить простір імен, стійкий до колізій;
- Приватні клейми - спеціальні клейми, створені для обміну інформацією між сторонами, які домовилися про їх використання, і не є ні зареєстрованими, ні публічними.

Корисне навантаження потім кодується Base64Url для формування другої частини веб-токена JSON.

Щоб створити частину підпису, вам потрібно взяти закодований заголовок, закодовану корисну інформацію, секрет, алгоритм, указаний у заголовку і підписати це.

Підпис використовується для перевірки того, що повідомлення не було змінено на шляху і, у випадку токенів, підписаних закритим ключем, він також може підтвердити, що відправник JWT є тим, ким він себе іменує.

Результатом є три рядки Base64-URL, розділені крапками, які можна легко передати в середовищах HTML і HTTP, але при цьому є більш компактними порівняно зі стандартами на основі XML, такими як SAML.

Щоразу, коли користувач хоче отримати доступ до захищеного маршруту або ресурсу, клієнт користувача повинен надіслати JWT, як правило, у заголовку авторизації за допомогою Bearer схеми. Вміст заголовка має виглядати так: Authorization: Bearer <token>.

За рахунок простоти роботи, достатньої захищеності та компактного розміру чудово підходить для реалізації аутентифікації та подальшої авторизації.

Спочатку необхідно добавити додатковий NuGet пакет Microsoft.AspNetCore.Authentication.JwtBearer до серверного проекту. Цей пакет додає проміжну підпрограму ASP.NET Core, яке дозволяє програмі отримувати Bearer OpenID Connect токен.

Потім у файлі Program.cs налаштувати раніше добавлену проміжну підпрограму для аутентифікації, в ній вказати що буде використовуватись Bearer схема для токена та вказати тип шифрування(в даному випадку симетричне) і вказати на секрет, що буде використовуватись при підписі, що зберігається в налаштуваннях проекту(рис. 3.25) :

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8
                .GetBytes(builder.Configuration.GetSection("AppSettings:Token").Value)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

Рисунок 3.25 - налаштування підпрограми аутентифікації

Після цього створимо контролер AuthController, що керуватиме взаємодією з обліковим записом користувача. Для коректної роботи контролеру необхідний UserRepository і IConfiguration через який буде отримуватись конфігурація проекту, в якому зберігається секрет. Також добавимо простір імен System.Security.Cryptography для підпису JWT токена та для хешування паролю користувача для подальшого збереження в базі даних та Microsoft.AspNetCore.Authorization для додавання клеймів в токен(рис. 3.26).

```
using Microsoft.AspNetCore.Authorization;
using System.Security.Cryptography;

namespace BoardGameShop.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private readonly IConfiguration _configuration;
        private readonly IUserRepository _userRepository;

        public AuthController(IConfiguration configuration, IUserRepository userRepository)
        {
            _configuration = configuration;
            _userRepository = userRepository;
        }
```

Рисунок 3.26 - лістинг фрагменту класу AuthController

Для реєстрації користувача приймаємо логін, адрес електронної пошти та пароль. Потім перевіряємо чи коректні дані направив користувач - чи існує користувач з таким логіном і/або поштою. формат електронної пошти, довжину логіну і паролю і т.д.. Після створюємо новий об’єкт користувача, переносимо логін і пошту в відповідні поля, після чого обчислюємо хеш паролю і зберігаємо залишок і хеш в відповідні поля. Додаємо зареєстрованого користувача до бази даних з збереженням і відправляємо користувачу код 200(рис. 3.27).

```
[HttpPost("registration")]
public async Task<IActionResult> UserRegistration(UserRegistrationDto request)
{
    bool isGood = await CheckRegistrationInputs(request);
    if (!isGood)
    {
        return BadRequest("Уже існує або неправильний пароль чи e-mail");
    }
    var user = new User
    {
        Email = request.Email,
        Username = request.Username
    };
    (user.PasswordHash, user.PasswordSalt) = CreatePasswordHash(request.Password);
    await _userRepository.Add(user);
    return Ok();
}
```

Рисунок 3.27 - лістинг методу UserRegistration

При аутентифікації клієнт відправлятиме лише логін і пароль, після чого проводимо пошук в базі даних користувача. Якщо не знаходимо жодного або більше одного - повертаємо помилку клієнтові. Інакше, використовуючи хеш і залишок з бази даних, перевіряємо пароль; потім створюємо Json веб-токен у якого в клеймах буде зберігатись логін користувача, його права (роль) та ідентифікатор(рис. 3.28). Підписуємо використовуючи такий же ш алгоритм і секрет з файлу конфігурації проекту, що вказали у проміжній підпрограмі аутентифікації і повертаємо користувачу токен, який буде зберігатись в локальній пам’яті браузера(рис. 3.29).

```
private string CreateToken(User user)
{
    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, user.Username),
        new Claim(ClaimTypes.Role, user.UserRole.ToString()),
        new Claim(ClaimTypes.Sid, user.Id.ToString())
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
        _configuration.GetSection("AppSettings:Token").Value));

    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

    var token = new JwtSecurityToken(
        claims: claims,
        expires: DateTime.Now.AddDays(1),
        signingCredentials: creds);

    var jwt = new JwtSecurityTokenHandler().WriteToken(token);

    return jwt;
}
```

Рисунок 3.28 - лістинг методу CreateToken

```
[HttpPost("login")]
public async Task<ActionResult<string>> Login(UserLoginDto request)
{
    var users = _userRepository.GetUserByUsername(request.Username)
        .Select(cd => new User
        {
            Id = cd.Id,
            PasswordHash = cd.PasswordHash,
            PasswordSalt = cd.PasswordSalt,
            Username = cd.Username,
            UserRole = cd.UserRole
        });
    if (!users.Any() || users.Count() != 1)
    {
        return Unauthorized("Неправильний логін або пароль");
    }
    var user = await users.FirstAsync();
    bool verifycation = VerifyUser(request.Password, user.PasswordHash, user.PasswordSalt);
    if (!verifycation)
    {
        return Unauthorized("Неправильний логін або пароль");
    }
    var token = CreateToken(user);
    return Ok(token);
}
```

Рисунок 3.29 - лістинг методу Login

Далі перейдемо до реалізації контролеру замовлень OrderController. У ньому реалізуємо два методи - на створення замовлень та отримання всіх замовлень користувача.

При створенні замовлень необхідно враховувати, що користувач може одночасно оформити декілька замовлень, так як добавить в кошик предмети від різних постачальників, а будь-яке замовлення відноситься лише до одного певного постачальника. В окремому Dto об’єкті для створення замовлень буде

міститись масив предметів замовлення, адрес доставки і повідомлення користувача, інформація про користувача, що містить його ПІБ та номер телефону. Відповідний метод CreateOrders прийматиме масив таких Dto об’єктів для створення замовлень та повертатиме статус код виконання.

Спочатку перевіримо чи прийшло нам щось насправді від користувача. Далі з клеймів потрібно отримати ідентифікатор та логін користувача для перевірки існування такого користувача. Потім робимо подібну перевірку з іграми - отримуємо з бази даних записи про ігри, що вказав користувач і перевіряємо чи всі вказані ним ігри існують чи збігаються в часові мітки і чи є потрібна кількість предметів в продажі. Після цих перевірок змінюємо кількість ігор в продажі у відповідних сутностей, створюємо об’єкт замовлення. Потім викликаємо додавання запису замовлення і оновлення записів настільних ігор, так як змінили кількість для продажу. Якщо ці дві дії відбуваються успішно - зберігаємо зміни в базу даних і повертаємо статус код 200 на клієнтську частину(рис. 3.30).

```
[HttpPost("create")]
[Authorize(Roles = nameof(Role.User))]
public async Task<IActionResult> CreateOrders(IEnumerable<CreateOrderDto> request)
{
    try{
        if (request == null || request.Count() <= 0)
            return BadRequest();
        bool isParse = Int32.TryParse(User.FindFirstValue(ClaimTypes.Sid), out int id);
        if (!isParse)
            return BadRequest("Не вдалось зчитати айді з токена");
        string username = User.FindFirstValue(ClaimTypes.NameIdentifier);
        var user = await userRepo.Find(id);
        if (user.Username != username)
            return Unauthorized("Користувача не знайдено");
        foreach (var order in request){
            var games = await gameRepo.GetByVendor(order.VendorId)
                .Where(g => order.Items.Select(i => i.Id).Contains(g.Id))
                .ToListAsync();
            foreach (var item in order.Items){
                var game = games.SingleOrDefault(g => g.Id == item.Id);
                if (game == null)
                    return BadRequest("Один з товарів не існує");
                if (!game.TimeStamp.SequenceEqual(item.TimeStamp))
                    return BadRequest($"Неправильна часова мітка у {game.Name}, дані не актуальні");
                if (game.Quantity < item.Qty)
                    return BadRequest($"Недостатньо {game.Name} на складі");
                games[games.IndexOf(game)].Quantity -= item.Qty;
            }
            var orderItems = ConvertToEntity(order.Items).ToList();
            orderItems.ForEach(oi => oi.ItemCost = GetPrice(games.Single(g => g.Id == oi.ItemId), oi.Qty));
            var orderEntity = new Order...;
            await orderRepo.Add(orderEntity, persiste: false);
            await gameRepo.UpdateRange(games, persiste: false);
        }
        await orderRepo.SaveChanges();
        return Ok();
    }
```

Рисунок 3.30 - лістинг методу CreateOrders

Для отримання замовлень користувача перевіряємо його ідентифікатор та логін в токені як в попередньому методі, після чого повертаємо масив замовлень(рис. 3.31).

```
[HttpGet("userOrders")]
[Authorize(Roles = nameof(Role.User))]
public async Task<ActionResult<IEnumerable<OrderDetailsDto>>> GetUserOrders()
{
    try
    {
        bool isParse = Int32.TryParse(User.FindFirstValue(ClaimTypes.Sid), out int id);
        if (!isParse)
            return BadRequest("Не вдалось зчитати айді з токена");
        string username = User.FindFirstValue(ClaimTypes.NameIdentifier);
        var user = await userRepo.Find(id);
        if (user.Username != username)
            return Unauthorized("Користувача не знайдено");
        var quryableOrder = orderRepo.GetUserOrders(id);
        var result = await quryableOrder.Select(qo => new OrderDetailsDto...).ToListAsync();
        return Ok(result);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message + ex.InnerException?.Message);
    }
}
```

Рисунок 3.31 - лістинг методу GetUserOrders

Для створення панелі постачальника, через яку він буде керувати замовленнями і пов’язаними з ним іграми, створимо контролер VendorController, що прийматиме всі запити від панелі постачальника.

Для отримання статистики про ігри постачальника створимо метод FindGamesByVendor.

Спочатку отримаємо ідентифікатор користувача з токену і перевіримо чи існує такий користувач, має відповідні права взаємодії з даними цього постачальника. Потім отримаємо всі настільні ігри постачальника, також обрахувавши кількість проданих копій кожної гри і на яку суму, шукаючи ці ігри в замовленнях, в яких уже є дата завершення замовлення; і повертаємо відповідний масив Dto об’єктів, що представлятимуть список ігор(рис. 3.32).

```
[HttpGet("games")]
[Produces("application/json")]
public async Task<ActionResult<List<BoardgameStatDto>>> FindGamesByVendor(){
    try{
        int userId = int.Parse(User.FindFirstValue(ClaimTypes.Sid) ?? "");
        int id = await vendorEmpRepo.GetVendorId(userId);
        var querybleOrder = gameRepo.GetOrderItemsByVendor(id).Include(g => g.PublisherNavigation)
            .Select(g => new BoardgameStatDto
            {
                Id = g.Id, Name = g.Name, TimeStamp = g.TimeStamp,
                PublisherName = g.PublisherNavigation.Name,Discount = g.Discount,
                FullPrice = g.FullPrice,IsDeleted = g.IsDeleted,Quantity = g.Quantity,
                ItemSold = g.OrderItems.Where(oi => oi.OrderNavigation.CompletionDate != null).Sum(oi => oi.Qty),
                Earned = g.OrderItems.Where(oi => oi.OrderNavigation.CompletionDate != null).Sum(oi => oi.ItemCost)
            });
        return Ok(await querybleOrder.ToListAsync());
    }
    catch (InvalidCastException ex){
        return Unauthorized("Неможливо отримати ідентифікатор користувача : " + ex.Message);
    }
    catch (UnauthorizedAccessException ex){
        return Unauthorized("Недостатньо прав в користувача : " + ex.Message);
    }
    catch (Exception ex){
        return BadRequest("Помилка : " + ex.Message + ex.InnerException?.Message);
    }
}
```

Рисунок 3.32 - лістинг методу FindGamesByVendor

Для реалізації клієнтської частини створимо BlazorWebAssembly проект і додамо наступні NuGet пакети : Blazored.LocalStorage для взаємодії з пам’ятю браузера, AspNetCore.Components.Authorization і IdentityModel.Tokens.Jwt для зчитування клеймів з JWT токена і коректної роботи аутентифікації та авторизації.

Для відправлення запитів і отримування їх результатів в фреймоврокі BlazorWebAssebly використовується клас HttpClient, взаємодію з яким доцільно винести в окремі класи-сервіри. Таким чином ми отримаємо сторінки з їх логікою роботи, відображенням даних і класи-сервіси, що відправлятимуть запити з подальшим опрацювання результату і відправленням об’єктів або коректного повідомлення про помилку. Також в класах-сервісах буде знаходитись взаємодія з локальною пам’яттю.

Створимо сервіс для отримання даних про настільні ігри ProductService та реалізуємо отримання інформації про окрему гру(рис. 3.33). Для цього ми приймемо від клієнта лише ідентифікатор гри та передамо його в URL за яким ідентифікуємо потрібний метод та контролер в REST API сервері. Якщо отримуємо в відповіді код 200, то зчитуємо і повертаємо вміст JSON файлу; інакше зчитуємо рядок помилки і повертаємо без доповнення.

```
public async Task<ProductDetailsDto> GetProductById(int id)
{
    try
    {
        var result = await httpClient.GetAsync($"api/BoardGame/{id}");
        if (result.IsSuccessStatusCode)
        {
            return await result.Content.ReadFromJsonAsync<ProductDetailsDto>()
                ?? new ProductDetailsDto();
        }
        string message = await result.Content.ReadAsStringAsync();
        throw new Exception(message);
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
```

Рисунок 3.33 - лістинг методу GetProductById

Для коректної роботи авторизації і аутентифікації на клієнській частині потрібно реалізувати абстрактний клас AuthenticationStateProvider і його метод GetAuthenticationStateAsync. Він потрібен для визначення стану аутентифікації та повідомлення іншим частинам клієнтської частини, що стан змінився і надати їм доступ до інформації стану.

В методі GetAuthenticationStateAsync отримуємо JWT токен з пам’яті (якщо користувач увійшов в свій обліковий запис), після чого зчитуємо його клейми, створюємо користувача і його стан, повідомляємо клієнтську частину про завершення аутентифікації, додаємо токен в заголовок всіх HTTP запитів від клієнтської частини.

```
public class CustomAuthStateProvider : AuthenticationStateProvider, IDisposable
{
    public string Username { get; set; } = string.Empty;
    private readonly ILocalStorageService localStorage;
    private readonly HttpClient http;
    public CustomAuthStateProvider(ILocalStorageService localStorage, HttpClient http)...
    public async override Task<AuthenticationState> GetAuthenticationStateAsync(){
        string token = await localStorage.GetItemAsync<string>("token") ?? "";
        var identity = new ClaimsIdentity();
        var jwtTokenHandler = new JwtSecurityTokenHandler();
        http.DefaultRequestHeaders.Authorization = null;
        if (jwtTokenHandler.CanReadToken(token)){
            var jwtToken = jwtTokenHandler.ReadJwtToken(token);
            identity = new ClaimsIdentity(jwtToken.Claims, "boardgameShop");
            http.DefaultRequestHeaders.Authorization =
                new AuthenticationHeaderValue("Bearer", token.Replace("\"", ""));
        }
        var user = new ClaimsPrincipal(identity);
        var state = new AuthenticationState(user);
        NotifyAuthenticationStateChanged(Task.FromResult(state));
        return state;
    }
```

Рисунок 3.34 - лістинг фрагменту класу GetAuthenticationStateProvider

3.3 Тестування програмного модуля для продажу настільних ігор

Після запуску на сторінці веб-додатку представлена сітка різних настільних ігор видимих для покупки з загальною про них інформацією(рис.3.35). Користувач може використати кнопку на картці кожного товару(або назвою чи зображенням), щоб перейти на сторінку з детальним описом цього конкретного товару.

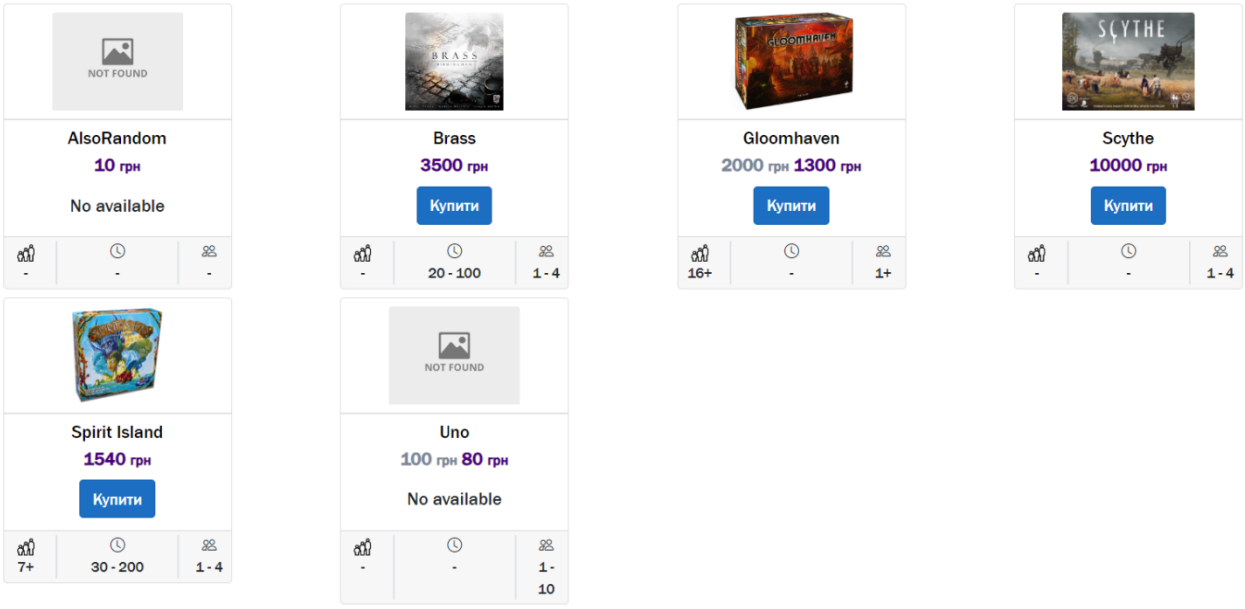

Рисунок 3.35 – сітка настільних ігор

Зверху знаходиться меню навігації, що дозволяє переключатись між вікнами сайту. В анонімного користувача там будуть кнопки “Настільні ігри”, “Кошик”, “Увійти”. “Зареєструватись”(рис. 3.36).

Рисунок 3.36 – меню навігації анонімного користувача

Між сіткою ігор та меню навігації на екрані настільних ігор також знаходиться вибір параметрів фільтрації за кількістю гравців, ціною, виробником, категоріями, механіками, авторами та дизайнерами(рис. 3.37).

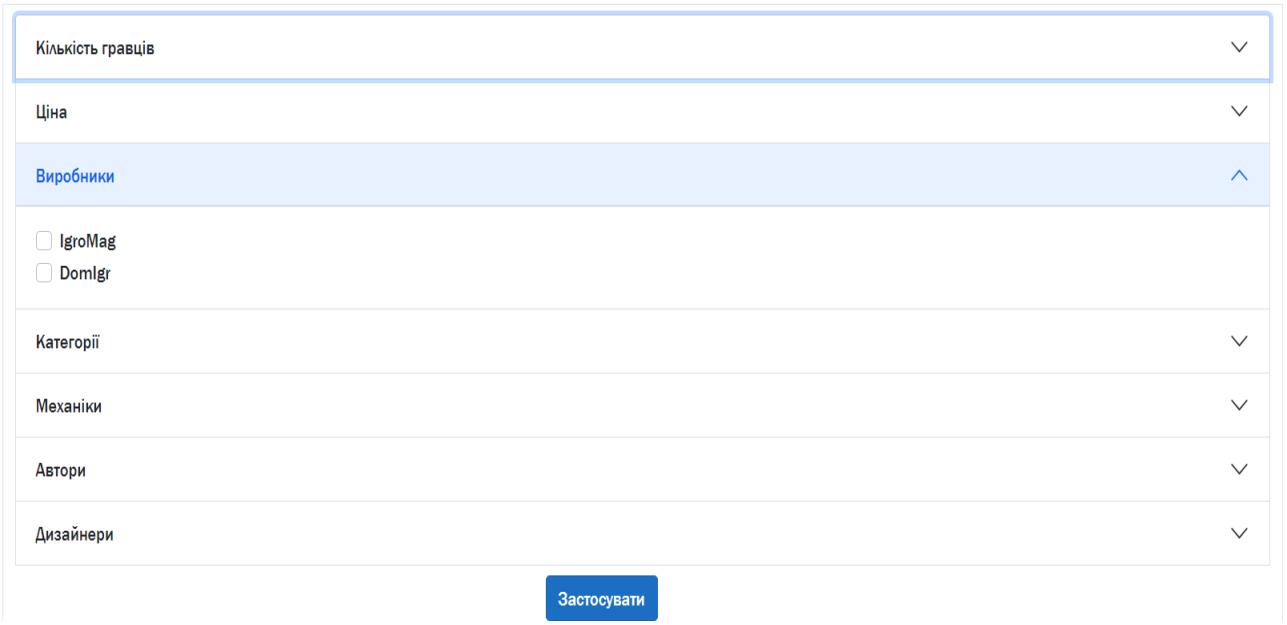

Рисунок 3.37 – фільтри для пошуку настільних ігор

Натиснемо на один з товарів і перейдемо до його детального опису. На цьому екранові користувач може переглянути всі добавлені зображення товару, тривалість гри, кількість гравців, вікову категорію, виробника, автора, художника, категорії механіки та його опис(рис. 3.38). Також натиснувши кнопку “Купити” добавити товар у кошик.

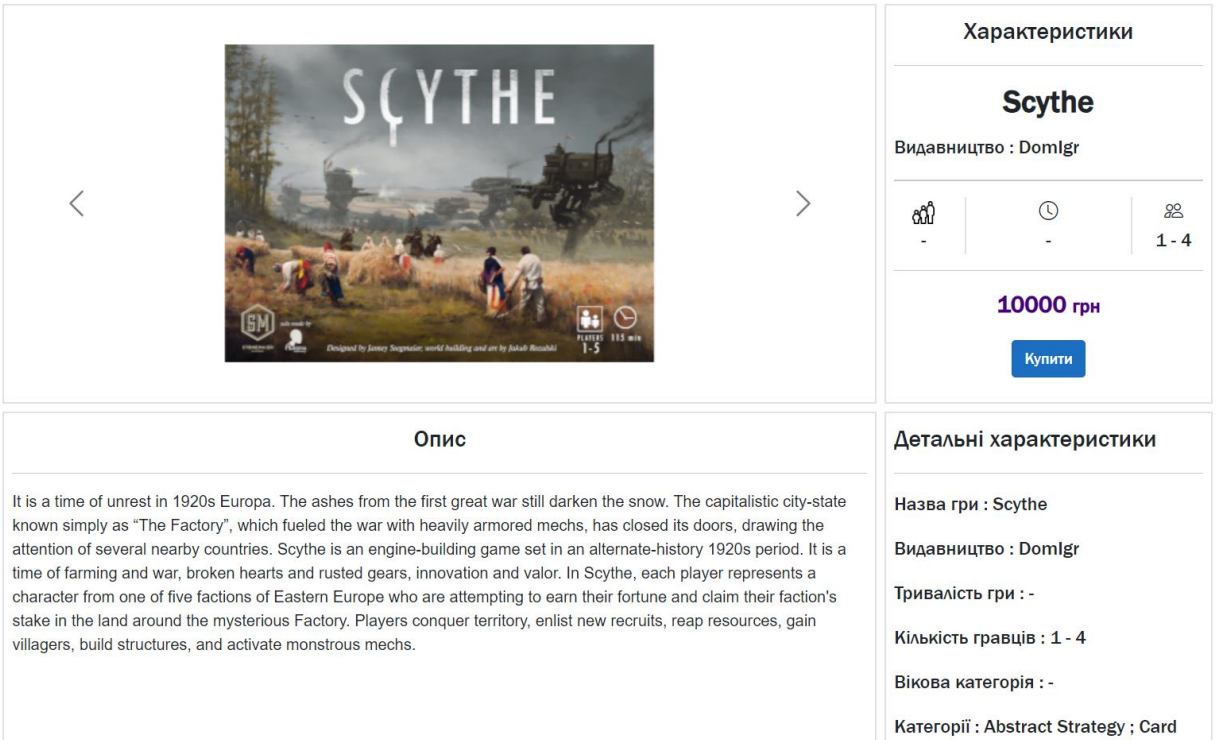

Рисунок 3.38 – детальний опис настільної гри Scythe

Натиснемо кнопку “Купити” і перейдемо в кошик. В цьому вікні побачимо всі додані настільні ігри в кошик, можливість змінювати їх кількість або видаляти за допомогою відповідних кнопок, ціну окремої гри та загальну ціну; кнопки для видалення вмісту кошику і дві кнопки навігації - до каталогу настільних ігор та до форми входу в обліковий запис(рис. 3.39)

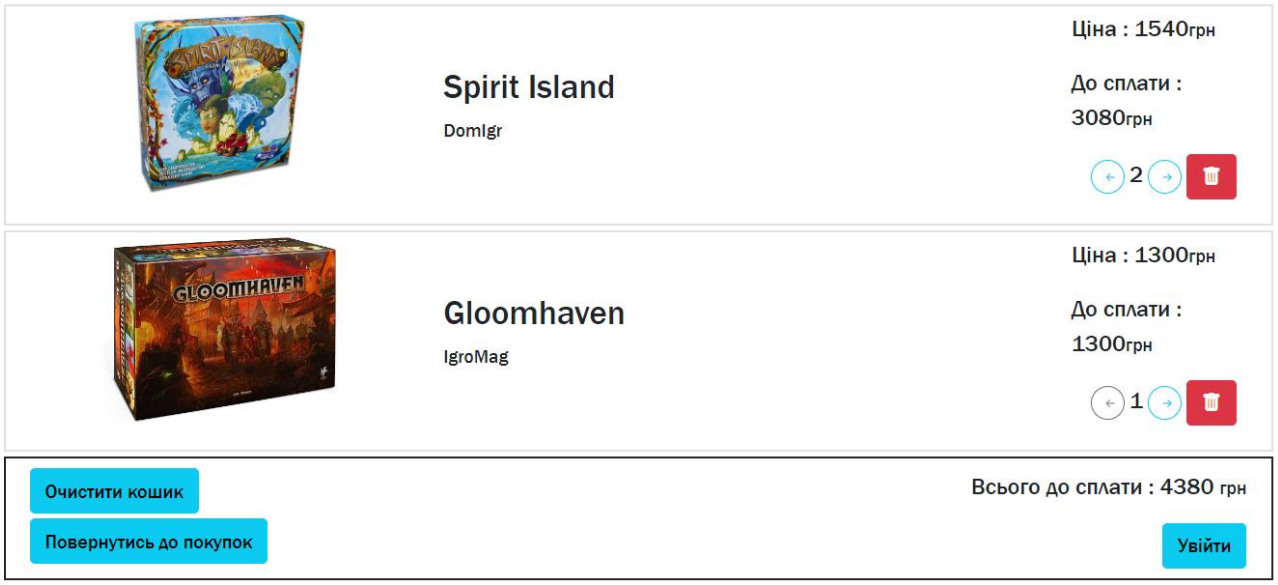

Рисунок 3.39 – кошик користувача

Перед тим як увійти в обліковий запис необхідно його створити, тому перейдемо до форми реєстрації(рис. 3.40). Вона містить поле для електронної пошти, логіну користувача, паролю та для повторного його введення. Форма для входу в обліковий запис має поля для логіну користувача та для паролю(рис. 3.41).

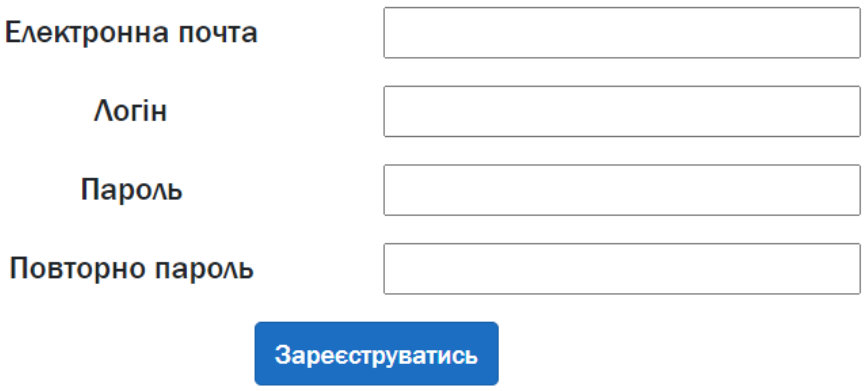

Рисунок 3.40 – форма реєстрації облікового запису

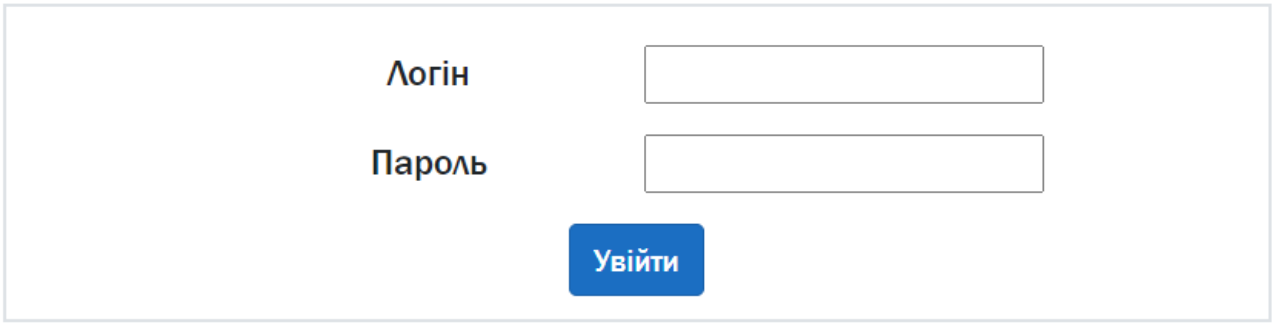

Рисунок 3.41 – форма входу в обліковий запис

Після того як увійдемо в обліковий запис, перейдемо в кошик і натиснемо кнопку “Оформити замовлення”. Після цього ми перейдемо на екран створення замовлення, де буде знаходитись форма для інформації про користувача.Цю інформацію про себе користувач може окремо ввести в додатковій інформації про свій обліковий запис. Для цього на панелі навігації потрібно натиснути на свій логін і тоді відкриється форма як на рисунку 3.42. Також на екранові замовлень знаходяться відповідні замовлення до різних постачальників як на рисункові 3.43, де маємо замовлення до постачальника IgroDom і до постачальника BGG.

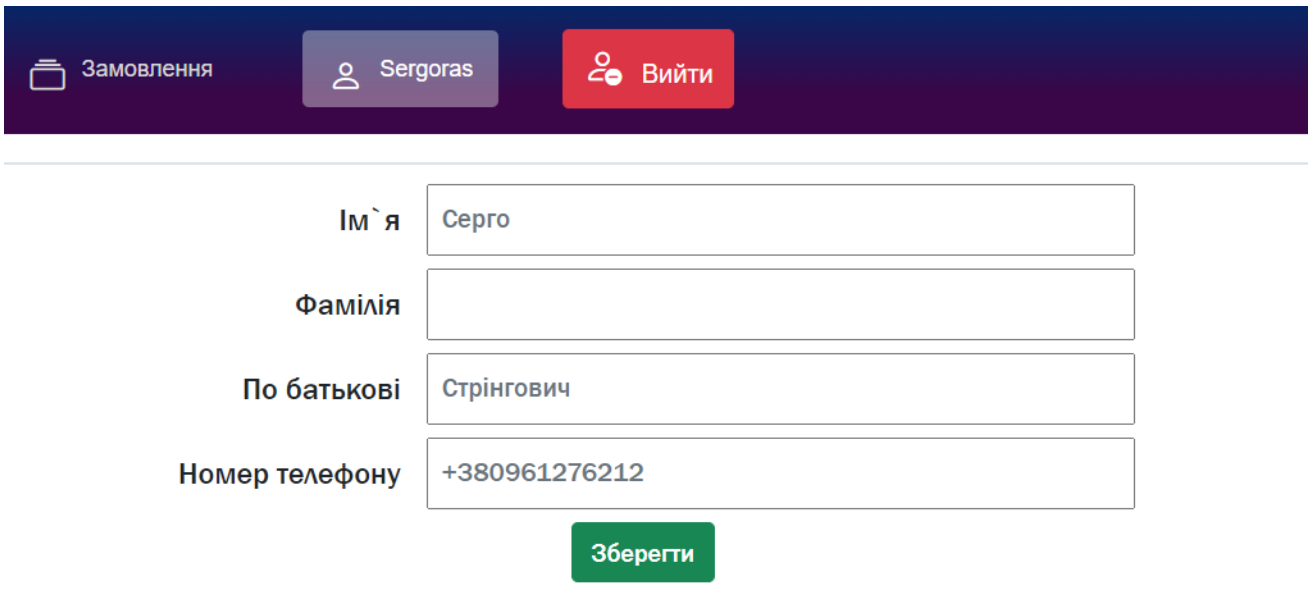

Рисунок 3.42 - додаткова інформація про користувача

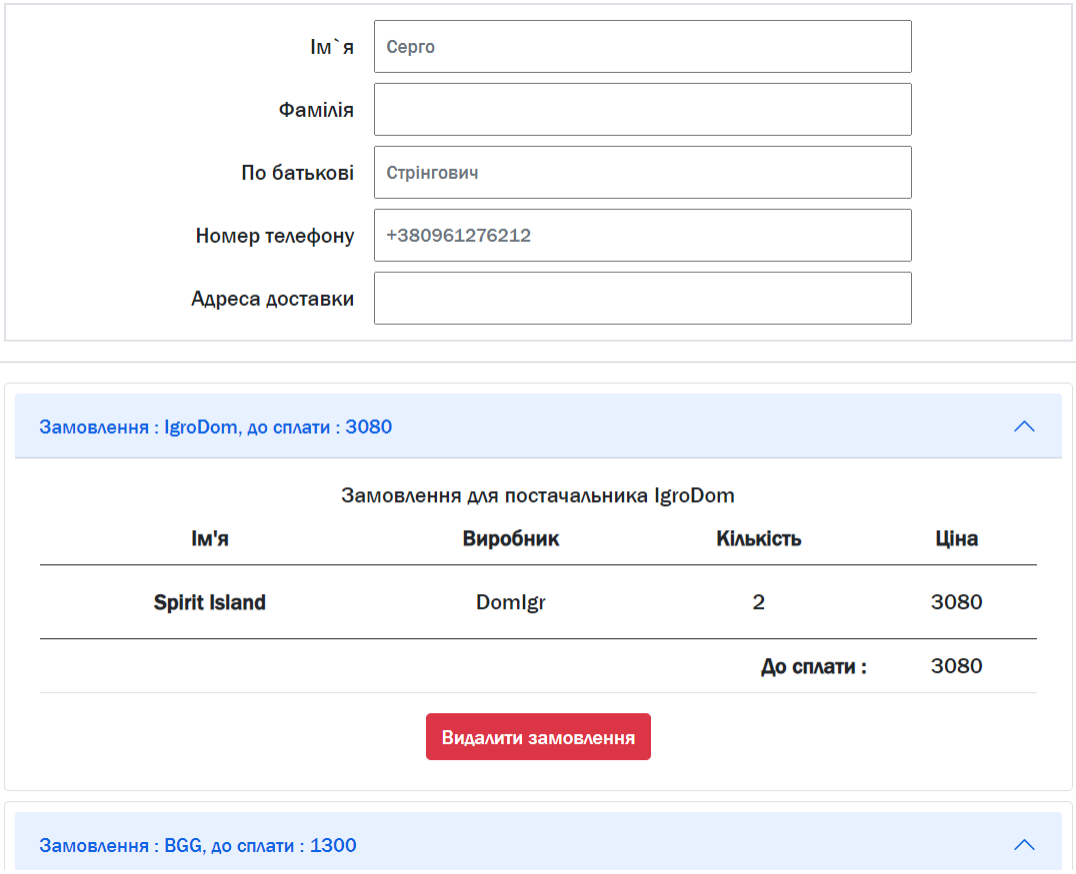

Рисунок 3.43 - форма створення замовлень

Далі користувач при виборі відповідного пункту меню в навігаційній панелі отримає список створених ним замовлень і детальну інформацію про предмети замовлення загальну суму і статус(рис. 3.44).

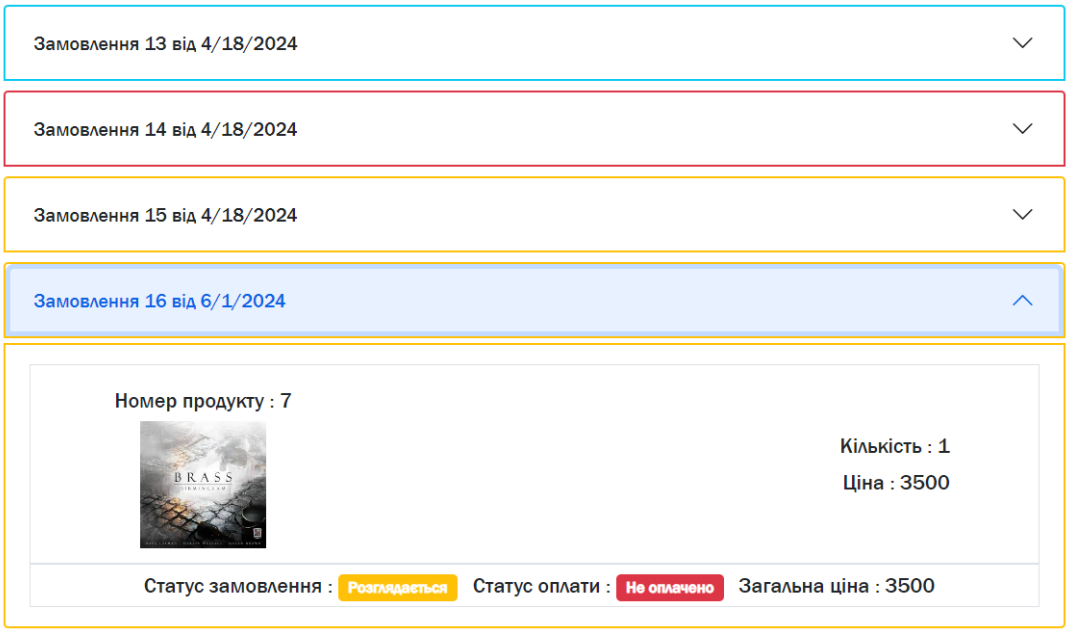

Рисунок 3.44 - список замовлень користувача

Далі увійдемо в обліковий запис з правами постачальника та перейдемо до статистики настільних ігор. На цьому екрані ми можемо побачити деяку інформацію, що стосується саме продажу і кнопки для припинення продажу гри та для відкриття форми редагування гри(рис 3.45).

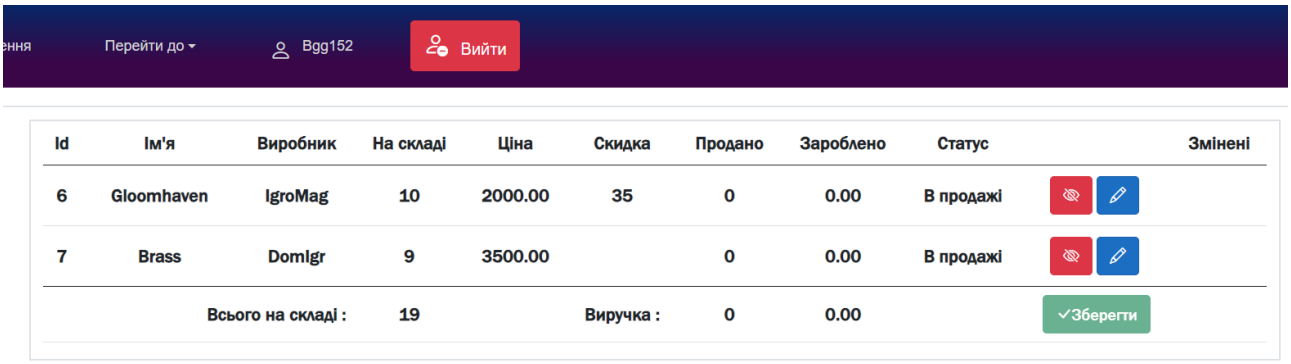

Рисунок 3.45 - список настільних ігор з облікового запису постачальника

В результаті розробки було досягнуто поставленої мети за рахунок реалізації функцій(табл. 3.2), що відсутні в програмах аналогах, а саме :

- Наявність повного детального опису настільних ігор
- Наявність широкого вибору параметрів фільтрації настільних ігор

Таблиця 3.2 – Таблиця порівняння наявності функцій в програмах

	Зручний інтерфейс	Повний детальний опис гри	Комплексна фільтрація	Якісний асортимент товару
Розроблений продукт	+	+	+	+
Rozetka	+	-	-	-
Дім Ігор	+	-	+	+
Ігромаг	-	-	-	+

За результатами таблиці можна побачити, що розроблений програмний засіб має покращений функціонал, принаймні на 2 можливості у порівнянні з програмами-аналогами.

3.4 Висновки

У цьому розділі було виконано розробку та тестування веб-ресурсу і також було проведено обґрунтування вибраних інструментів розробки.

Для реалізації серверної частини ресурсу була обрана мова програмування C#, оскільки вона підтримує об'єктно-орієнтоване програмування, відома своєю надійністю та зручністю. З метою забезпечення ефективної розробки веб-додатків, було прийнято рішення використовувати фреймворк ASP.NET Core, який надає широкий спектр модулів для веб-розробки. Для розробки клієнтської частини був використаний фреймворк BlazorWebAssembly, що є зручним для розробки з ASP.NET за рахунок відповідних програмних модулів та єдиної мови програмування, що гарантує безпеку типів. Для взаємодії з базою даних був використаний фреймворк Entity Framework Core, що надає зручний і простий доступ до бази даних за рахунок абстракції сутностей і запитів від архітектури СУБД.

Здійснено тестування розробленого сервісу, яке підтвердило його працездатність та довело, що розроблений програмний засіб має покращений функціонал, принаймні на 2 можливості у порівнянні з програмами-аналогами.

ВИСНОВКИ

У результаті виконання бакалаврської дипломної роботи було розроблено програмний модуль для продажу настільних ігор.Всі завдання поставлені перед бакалаврською дипломною роботою виконані.

Було проаналізовано предметну область, проаналізовано етапи процесу розробки веб-додатків, визначено актуальні підходи до веб-розробки та обґрунтовано доцільність їх використання під час розробки веб-додатків.Було розглянуто аналогічні програмні рішення і виявлено їх переваги та недоліки.

Було розглянуто відомі архітектурні патерни, зокрема REST патерн, що в ході розробці не створює надмірної комплексності та чудово підходить для розробки веб додатків. Потім були розроблені UML діаграми для класів-сутностей, класів-репозиторіїв, були визначені типи користувачів та їх можливі дії. Була розроблена загальна схема взаємодії користувача з модулем.Потім було виконано розробку та тестування веб-ресурсу і також було проведено обґрунтування вибраних інструментів розробки.

Мета дослідження у вигляді розширення функціональних можливостей та інформативності програмного модуля для продажу настільних ігор було досягнуто за рахунок того, що у порівнянні з розглянутими виробами-аналогами, розроблений програмний модуль надає збільшені можливості фільтрації та більшу кількість характеристик окремої гри.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Web Application Development: Process, Tools, & Examples [Електронний ресурс] – режим доступу : https://www.browserstack.com/guide/web-application-development-guide

2. Rozetka про нас [Електронний ресурс] – режим доступу : https://rozetka.com.ua/ua/pages/about/

3. Дім ігр про нас [Електронний ресурс] – режим доступу : https://domigr.com.ua/ua/about_us.php

4. Ігромаг [Електронний ресурс] – режим доступу : https://desktopgames.com.ua/ua/

5. Geekach [Електронний ресурс] – режим доступу : https://geekach.com.ua

6. Gameland [Електронний ресурс] – режим доступу : https://gameland.com.ua/o-nas/

7. 5 essential patterns of software architecture [Електронний ресурс] – режим доступу : https://www.redhat.com/architect/5-essential-patterns-software-architecture#client-server

8. Software Architecture Patterns [Електронний ресурс] – режим доступу : https://www.spiritofsoft.com/software-architecture-patterns/

9. Software Architecture Patterns [Електронний ресурс] – режим доступу: https://medium.com/@mahmoudIbrahimAbbas/software-architecture-patterns-558486f4c3aa

10. Open Universiteit. Architectural patterns [Електронний ресурс] – режим доступу : https://www.ou.nl/documents/40554/791670/IM0203_03.pdf

11. REST Architecture [Електронний ресурс] – режим доступу : https://www.redhat.com/en/blog/rest-architecture

12. Unified Modeling Language [Електронний ресурс] – режим доступу :https://en.wikipedia.org/wiki/Unified_Modeling_Language

13. Об'єктно-реляційне відображення [Електронний ресурс] – режим доступу : https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82

%D0%BD%D0%BE-%D1%80%D0%B5%D0%BB%D1%8F%D1%86%D1%96%D0%B9%D0%BD%D0%B5_%D0%B2%D1%96%D0%B4%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F

14. What is object-relational mapping (ORM)? [Електронний ресурс] – режим доступу : https://www.theserverside.com/definition/object-relational-mapping-ORM

15. Object-Relational Mapping Tools: Pros, Cons, and When to Use [Електронний ресурс] – режим доступу : https://www.altexsoft.com/blog/object-relational-mapping-tools/

16. Timestamp Based Concurrency Control [Електронний ресурс] – режим доступу : https://www.prepbytes.com/blog/dbms/timestamp-based-concurrency-control/

17. Handling Concurrency Conflicts [Електронний ресурс] – режим доступу : https://learn.microsoft.com/en-us/ef/core/saving/concurrency

18. UML Class Diagram Tutorial [Електронний ресурс] – режим доступу : https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

19. Куліков В.М., Рябцев В.В., Паршуков С.С. Об’єктно-орієнтоване програмування для фахівців з кібербезпеки: навч. посіб. / ІСЗЗІ КПІ ім. Ігоря Сікорського. Київ: КПІ ім. Ігоря Сікорського, 2023. – 365 с.

20. Repository [Електронний ресурс] – режим доступу : https://martinfowler.com/eaaCatalog/repository.html

21. What is HTTP? [Електронний ресурс] – режим доступу : https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

22. DTO [Електронний ресурс] – режим доступу : https://uk.wikipedia.org/wiki/DTO

23. Sequence diagram [Електронний ресурс] – режим доступу : https://en.wikipedia.org/wiki/Sequence_diagram

24. C++ [Електронний ресурс] – режим доступу :

https://en.wikipedia.org/wiki/C%2B%2B

25. C Sharp (programming language) [Електронний ресурс] – режим доступу : https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

26. Що краще C++ чи C#? [Електронний ресурс] – режим доступу : https://foxminded.ua/c-sharp-vs-c-plus-plus/

27. Entity Framework Core [Електронний ресурс] – режим доступу : https://learn.microsoft.com/en-us/ef/core/

28. ASP.NET overview [Електронний ресурс] – режим доступу : https://learn.microsoft.com/en-us/aspnet/overview

29. ASP.NET Core Blazor [Електронний ресурс] – режим доступу : https://learn.microsoft.com/en-us/aspnet/core/blazor

30. What are solutions and projects in Visual Studio? [Електронний ресурс] – режим доступу : https://learn.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio

31. What is JSON Web Token? [Електронний ресурс] – режим доступу : https://jwt.io/introduction

ДОДАТКИ

ДОДАТОК А
ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Розробка програмного модуля для продажу настільних ігор

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра комп'ютерних наук, ФІІТА
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 87,1% Схожість 12,9%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку Озеранський В.С.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи Миронюк Д.А.

Керівник роботи Денисюк В.О.

ДОДАТОК Б
ІНСТРУКЦІЯ КОРИСТУВАЧА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ПРОДАЖУ НАСТІЛЬНИХ ІГОР

Для переходу до списку настільних ігор необхідно натиснути на пункт меню “Настільні ігри” в навігаційній панелі, яка зображена на рисункові Б.1.

Рисунок Б.1 - меню навігації

Після чого буде отримана сітка різних настільних ігор видимих для покупки з загальною про них інформацією як на рисункові Б.2.

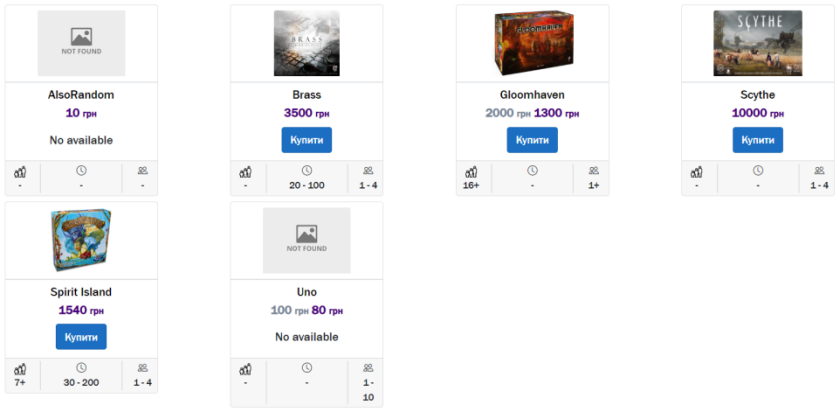

Рисунок Б.2 - сітка товарів

Натиснувши на кнопку “Купити” або на зображення товару відкриється сторінка опису окремого товару, що зображена на рисункові Б.3. На цій сторінці можемо ознайомитись з наявною інформацією про вікову категорію, час партії, необхідну кількість гравців,категорії і механіки гри;автори і художники; постачальник і виробник гри. Після натискання в цьому вікні кнопки “Купити” перейдемо до оформлення замовлення, що зображено на рисункові Б.4.

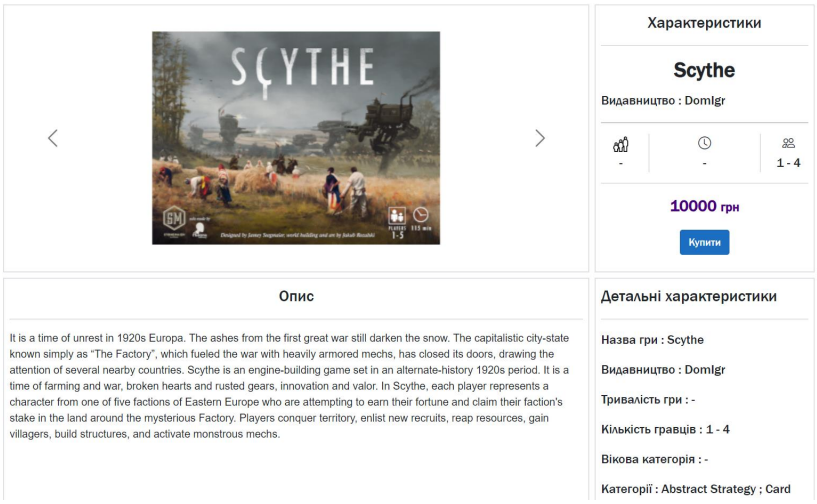

Рисунок Б.3 - детальних опис гри “Scythe”

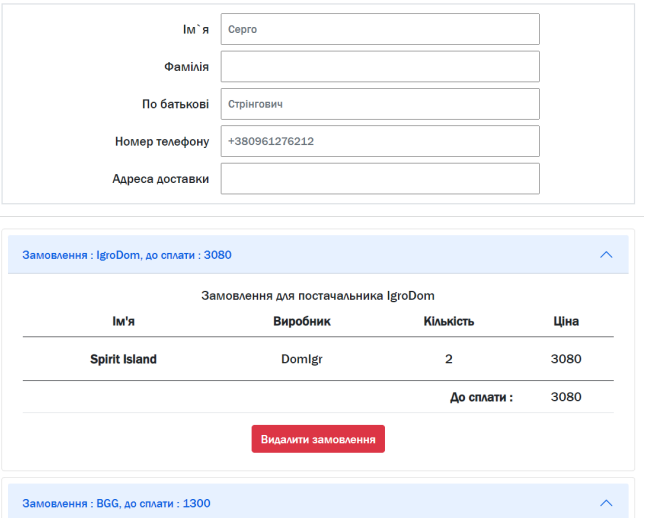

Рисунок Б.4 - оформлення замовлення

Для перегляду замовлень перейдемо в пункт меню “Замовлення”, що відкриє сторінку як на рисункові Б.5, що містить перелік замовлень користувача, продукти в замовлені та статус виконання і оплати замовлення.

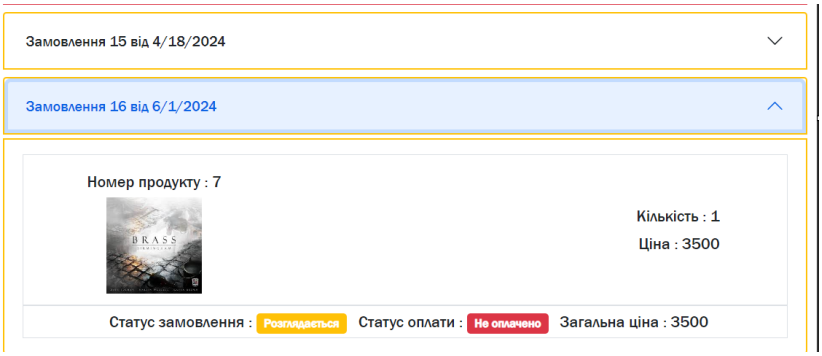

Рисунок Б.5 - список замовлень користувача

ДОДАТОК В

ЛІСТИНГ ПРОГРАМИ

BaseEntity.cs :

```
public class BaseEntity
{
   [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
   public int Id { get; set; }
   [Timestamp]
   public byte[] TimeStamp { get; set; }
}
```

BoardGame.cs :

```
[EntityTypeConfiguration(typeof(BoardgameConfiguration))]
public class Boardgame : BaseEntity
{
   [Required]
   [MaxLength(30)]
   public string Name { geit; set; } = string.Empty;
   [MaxLength(1000)]
   public string? Description { get; set; } = string.Empty;
   [DefaultValue(0)]
   public int Quantity { get; set; } = 0;
   public byte? MinPlayer { get; set; }
   public byte? MaxPlayer { get; set; }
   public int? MinPlayTime { get; set; }
   public int? MaxPlayTime { get; set; }
   [DefaultValue(false)]
   public bool IsDeleted { get; set; }
   public AgeEnum? Age { get; set; }
   [Required]
   [Precision(20, 2)]
   public decimal FullPrice { get; set; }
   public byte? Discount { get; set; }
   [InverseProperty(nameof(Category.Boardgames))]
   public List<Category> Categories { get; set; }
   [InverseProperty(nameof(BoardgameToCategory.BoardgameNavigation))]
   public List<BoardgameToCategory> BoardgameCategories { get; set; }
   [InverseProperty(nameof(BoardgameToMechanic.BoardgameNavigation))]
   public List<BoardgameToMechanic> BoardgameMechanics { get; set; }
   [InverseProperty(nameof(Mechanic.Boardgames))]
   public List<Mechanic> Mechanics { get; set; }
   [InverseProperty(nameof(Author.Boardgames))]
   public List<Author> Authors { get; set; }
   [InverseProperty(nameof(BoardgameToAuthor.BoardgameNavigation))]
   public List<BoardgameToAuthor> BoardgameAuthors { get; set; }
   [InverseProperty(nameof(Artist.Boardgames))]
   public List<Artist> Artists { get; set; }
   [InverseProperty(nameof(BoardgameToArtist.BoardgameNavigation))]
   public List<BoardgameToArtist> BoardgameArtists { get; set; }
   [Required]
   public int PublisherId { get; set; }
   [Required, ForeignKey(nameof(PublisherId))]
   [InverseProperty(nameof(Publisher.BoardGames))]
   public Publisher PublisherNavigation { get; set; }
   public int VendorId { get; set; }
   [InverseProperty(nameof(Vendor.Boardgames))]
   public Vendor VendorNavigation { get; set; }
   [InverseProperty(nameof(OrderItem.BoardgameNavigation))]
   public List<OrderItem>? OrderItems { get; set; }
}
```

Category.cs :

```
[Table("Categories")]
public class Category : BaseEntity
{
   [Required, StringLength(20)]
   public string Name { get; set; } = string.Empty;
   [StringLength(200)]
   public string? Description { get; set; } = string.Empty;
   [InverseProperty(nameof(Boardgame.Categories))]
   public List<Boardgame> Boardgames { get; set; }
   [InverseProperty(nameof(BoardgameToCategory.CategoryNavigation))]
   public List<BoardgameToCategory> BoardgameCategories { get; set; }
}
```

Mechanic.cs :

```
public class Mechanic : BaseEntity
{
   [Required, StringLength(20)]
   public string Name { get; set; } = string.Empty;
   [StringLength(200)]
   public string? Description { get; set; } = string.Empty;
   [InverseProperty(nameof(Boardgame.Mechanics))]
   public List<Boardgame> Boardgames { get; set; }
   [InverseProperty(nameof(BoardgameToMechanic.MechanicNavigation))]
   public List<BoardgameToMechanic> BoardgameMechanics { get; set; }
}
```

Artist.cs :

```
public class Artist : Creator
{
   [InverseProperty(nameof(Boardgame.Artists))]
   public List<Boardgame> Boardgames { get; set; }
   [InverseProperty(nameof(BoardgameToArtist.ArtistNavigation))]
   public List<BoardgameToArtist> BoardgameArtists { get; set; }
}
```

Author.cs

```
public class Author : Creator
{
   [InverseProperty(nameof(Boardgame.Authors))]
   public List<Boardgame> Boardgames { get; set; }
   [InverseProperty(nameof(BoardgameToAuthor.AuthorNavigation))]
   public List<BoardgameToAuthor> BoardgameToAuthors { get; set; }
}
```

Creator.cs :

```
public class Creator : BaseEntity
{
   [Required]
   [StringLength(100)]
   public string FullName { get; set; } = string.Empty;
}
```

Publisher.cs :

```
public class Publisher : BaseEntity
{
   [Required]
   [StringLength(50)]
   public string Name { get; set; } = string.Empty;
   [InverseProperty(nameof(Boardgame.PublisherNavigation))]
       public List<Boardgame> BoardGames { get; set; }}
```

Vendor.cs :

```
public class Vendor : BaseEntity
{
    public string Name { get; set; } = string.Empty;
    [InverseProperty(nameof(Boardgame.VendorNavigation))]
    public List<Boardgame> Boardgames { get; set; }
    public List<Order> Orders { get; set; }
}
```

VendorEmployee.cs :

```
public class VendorEmployee
{
    [Required]
    public int UserId { get; set; }
    [ForeignKey(nameof(UserId))]
    public User UserNavigation { get; set; }
    [Required]
    public int VendorId { get; set; }
    [ForeignKey(nameof(VendorId))]
    public Vendor VendorNavigation { get; set; }
}
```

Order.cs :

```
[EntityTypeConfiguration(typeof(OrderConfiguration))]
public class Order : BaseEntity
{
    [Required]
    public int UserId { get; set; }
    [Required]
    public DateTime CreationDate { get; set; }
    public DateTime? CompletionDate { get; set; }
    public DateTime? PaymentDate { get; set; }
    [Required]
    public int VendorId { get; set; }
    public Vendor? VendorNavigation { get; set; }
    public string? Firstname { get; set; }
    public string? Lastname { get; set; }
    public string? Secondname { get; set; }
    public string? PhoneNumber { get; set; }
    public string? DeliveryAddress { get; set; }
    public string? MessageFromUser { get; set; }
    [Required]
    public OrderStatus OrderStatus { get; set; }
    [Required]
    public PaymentStatus PaymentStatus { get; set; }
    [InverseProperty(nameof(OrderItem.OrderNavigation))]
    public List<OrderItem> OrderItems { get; set; }
}
```

OrderItem.cs :

```
[EntityTypeConfiguration(typeof(OrderItemConfiguration))]
public class OrderItem
{
    [Required]
    public int OrderId { get; set; }
    [Required]
    public int ItemId { get; set; }
    [Required]
    public int ItemCost { get; set; }
    [Required]
    public int Qty { get; set; }
    [Required]
    [ForeignKey(nameof(OrderId))]
    public Order OrderNavigation { get; set; }
    [ForeignKey(nameof(ItemId))]
    public Boardgame BoardgameNavigation { get; set; }
}
```

BaseRepository.cs :

```
public class BaseRepository<T> : IBaseRepo<T> where T : BaseEntity, new()
{
    private bool disposedValue;
    public BaseRepository(ApplicationDbContext applicationDbContext)
    {
        Context = applicationDbContext;
        Table = Context.Set<T>();
        disposedValue = false;
    }
    public ApplicationDbContext Context { get; }
    public DbSet<T> Table { get; }
    public async Task<int> Add(T entity, bool persiste = true)
    {
        await Table.AddAsync(entity);
        return persiste ? await SaveChanges() : 0;
    }
    public async Task<int> AddRange(IEnumerable<T> entities, bool persiste = true)
    {
        await Table.AddRangeAsync(entities);
        return persiste ? await SaveChanges() : 0;
    }
    public async Task<int> Delete(int id, byte[] timeStamp, bool persiste = true)
    {
        T entity = new T { Id = id, TimeStamp = timeStamp };
        Context.Entry(entity).State = EntityState.Deleted;
        return persiste ? await SaveChanges() : 0;
    }
    public async void Delete(T entity, bool persiste = true)
    {
        Table.Remove(entity);
        if (persiste)
            await SaveChanges();
        return;
    }
    public async void DeleteRange(IEnumerable<T> entities, bool persiste = true)
    {
        Table.RemoveRange(entities);
        if (persiste)
            await SaveChanges();
        return;
    }
    public async Task<IEnumerable<T>> GetAll()
    {
        return await Table.ToListAsync();
    }
    public async Task<T?> Find(int id)
    {
        return await Table.FindAsync(id);
    }
    public async Task<int> Update(T entity, bool persiste = true)
    {
        Table.Update(entity);
        return persiste ? await SaveChanges() : 0;
    }
    public async Task<int> UpdateRange(IEnumerable<T> entities, bool persiste = true)
    {
        Table.UpdateRange(entities);
        return persiste ? await SaveChanges() : 0;
    }
    public async Task<int> SaveChanges()
    {
        try
        {
            return await Context.SaveChangesAsync();
        }
```

```
        catch (Exception ex)
        {
            throw new Exception("In base repo" + ex.Message, ex);
        }
    }
    private bool _IsDisposed;
    protected virtual void Dispose(bool disposing)
    {
        if (_IsDisposed)
            return;
        if (disposing)
        {
            if (disposedValue)
            {
                Context.Dispose();
            }
            _IsDisposed = true;
        }
    }

    ~BaseRepository()
    {
        //     // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method
        Dispose(disposing: false);
    }

    public void Dispose()
    {
        // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }
}
```

BoardGameRepository.cs :

```
public class BoardGameRepository : BaseRepository<Boardgame>, IBoardGameRepository
{
    public BoardGameRepository(ApplicationDbContext applicationDbContext) : base(applicationDbContext) { }
    public async Task<int> CountPages(int itemPerPage)
    {
        int item = await Table.CountAsync();
        return (int)Math.Ceiling((double)(item / itemPerPage));
    }
    public async Task<(IEnumerable<Boardgame>, int)> GetByFilter(RequestFilterDto filterDto)
    {
        int totalItems = 0;
        var request = Table.AsQueryable();
        request = FilterByPublishers(request, filterDto.PublisherIds);
        request = FilterByCategories(request, filterDto.CategoryIds);
        request = FilterByMechanics(request, filterDto.MechanicIds);
        request = FilterByAuthors(request, filterDto.AuthorsIds);
        request = FilterByArtist(request, filterDto.ArtistIds);
        (request, totalItems) = await CountTotalPage(request);
        request = request.OrderBy(x => x.Name);
        request = Pagination(request, filterDto.CurrentPage, filterDto.ItemsPerPage);
        request = request.Select(cd => new Boardgame
        {
            Age = cd.Age,Discount = cd.Discount,FullPrice = cd.FullPrice,Id = cd.Id,
            MaxPlayer = cd.MaxPlayer,MinPlayer = cd.MinPlayer,
            MaxPlayTime = cd.MaxPlayTime,MinPlayTime = cd.MinPlayTime,
            Name = cd.Name,Quantity = cd.Quantity
        });
        return (request.ToList(), (int)Math.Ceiling(((double)totalItems / filterDto.ItemsPerPage)));
    }
    private IQueryable<Boardgame> Pagination(IQueryable<Boardgame> games, int currentPage, int itemPerPage)
    {
        return games.Skip(currentPage * itemPerPage).Take(itemPerPage);
```

```
    }
    private IQueryable<Boardgame> FilterByCategories(IQueryable<Boardgame> games, IEnumerable<int> categoriesIds)
    {
        if (categoriesIds.Count() == 0 || categoriesIds == null) return games;
        return games.Include(x => x.BoardgameCategories)
            .Where(g => g.BoardgameCategories.Where(bg => categoriesIds.Contains(bg.CategoryId)).Any());
    }
    private IQueryable<Boardgame> FilterByMechanics(IQueryable<Boardgame> games, IEnumerable<int> mechanicIds)
    {
        if (mechanicIds.Count() == 0 || mechanicIds == null) return games;
        return games.Include(g => g.BoardgameMechanics)
            .Where(g => g.BoardgameMechanics.Where(bm => mechanicIds.Contains(bm.MechanicId)).Any());
    }
    private IQueryable<Boardgame> FilterByAuthors(IQueryable<Boardgame> boardgames, IEnumerable<int> authorIds)
    {
        if (authorIds.Count() == 0 || authorIds == null) return boardgames;
        return boardgames.Include(g => g.BoardgameAuthors)
            .Where(g => g.BoardgameAuthors.Where(ga => authorIds.Contains(ga.AuthorId)).Any());
    }
    private IQueryable<Boardgame> FilterByArtist(IQueryable<Boardgame> boardgames, IEnumerable<int> artistIds)
    {
        if (artistIds.Count() == 0 || artistIds == null) return boardgames;
        return boardgames.Include(g => g.BoardgameArtists)
            .Where(g => g.BoardgameArtists.Where(gar => artistIds.Contains(gar.ArtistId)).Any());
    }
    private IQueryable<Boardgame> FilterByPublishers(IQueryable<Boardgame> boardgames, IEnumerable<int>
publisherIds)
    {
        if (publisherIds.Count() == 0 || publisherIds == null) return boardgames;
        return boardgames.Where(g => publisherIds.Contains(g.PublisherId));
    }
    private async Task<(IQueryable<Boardgame>, int)> CountTotalPage(IQueryable<Boardgame> boardgames)
    {
        int totalPage = await boardgames.CountAsync();
        return (boardgames, totalPage);
    }
    public async Task<Boardgame> GetById(int id)
    {
        var game = await Table.Include(cd => cd.PublisherNavigation)
            .Include(cd => cd.VendorNavigation).Include(cd => cd.Mechanics)
            .Include(cd => cd.Categories).Include(cd => cd.Artists)
            .Include(cd => cd.Authors).FirstAsync(g => g.Id == id);
        return game ?? new Boardgame();
    }
    public IQueryable<Boardgame> GetByVendor(int vendorId)
    {
        return Table.Where(g => g.VendorId == vendorId);
    }
    public IQueryable<Boardgame> GetOrderItemsByVendor(int vendorId)
    {
        return GetByVendor(vendorId).Include(g => g.OrderItems).ThenInclude(oi => oi.OrderNavigation).AsQueryable();
    }
}
```

Program.cs :

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.OpenApi.Models;
using Swashbuckle.AspNetCore.Filters;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(options =>
{
    options.AddSecurityDefinition("oauth2", new OpenApiSecurityScheme
    {
        Description = "Standard Authorization header using the Bearer scheme (\"bearer {token}\")",
```

```
        In = ParameterLocation.Header,
        Name = "Authorization",
        Type = SecuritySchemeType.ApiKey
    });
    options.OperationFilter<SecurityRequirementsOperationFilter>();
});
builder.Services.AddDbContextPool<ApplicationDbContext>(options =>
{
options.UseSqlServer(builder.Configuration.GetConnectionString("BoardGameDbConnection"));
});
builder.Services.AddScoped<IBoardGameRepository, BoardGameRepository>();
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<IMechanicRepository, MechanicRepository>();
builder.Services.AddScoped<IArtistRepository, ArtistRepository>();
builder.Services.AddScoped<IAuthorRepository, AuthorRepository>();
builder.Services.AddScoped<IPublisherRepository, PublisherRepository>();
builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IVendorEmployeeRepository, VendorEmployeeRepository>();
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8
                .GetBytes(builder.Configuration.GetSection("AppSettings:Token").Value)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
    app.UseWebAssemblyDebugging();
}
app.UseCors(options =>
{
    options.AllowAnyHeader();
    options.AllowAnyMethod();
});
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
app.UseBlazorFrameworkFiles();
app.UseStaticFiles();
app.MapFallbackToFile("index.html");
app.Run();
```

BoardGameController.cs :

```
[Route("api/[controller]")]
[ApiController]
public class BoardGameController : ControllerBase
{
    private readonly IBoardGameRepository gameRepository;
    public BoardGameController(IBoardGameRepository gameRepository)
    {
        this.gameRepository = gameRepository;
    }
    [Produces("application/json")]
    [HttpPost("Filter")]
    public async Task<IActionResult> FindAll([FromBody] RequestFilterDto filter)
```

```
    {
        (var games, int totalPage) = await gameRepository.GetByFilter(filter);
        var gamesDto = games.ConvertToDto();
        if (gamesDto == null)
        {
            return NotFound();
        }
        var result = new ProductsPageDto { Products = gamesDto, TotalPages = totalPage };
        return Ok(result);
    }
    [Produces("application/json")]
    [HttpGet("{id:int}")]
    public async Task<IActionResult> FindSingle(int id)
    {
        var game = await gameRepository.GetById(id);
        if (game == null)
            return NotFound();
        return Ok(game.ConvertToDto());
    }
}
```

AuthControlles.cs :

```
[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
    private readonly IConfiguration _configuration;
    private readonly IUserRepository _userRepository;
    public AuthController(IConfiguration configuration, IUserRepository userRepository)
    {
        _configuration = configuration;
        _userRepository = userRepository;
    }
    [HttpPost("registration")]
    public async Task<IActionResult> UserRegistration(UserRegistrationDto request)
    {
        bool isGood = await CheckRegistrationInputs(request);
        if (!isGood)
        {
            return BadRequest("Уже існує або неправильний пароль чи e-mail");
        }
        var user = new User
        {
            Email = request.Email,
            Username = request.Username
        };
        (user.PasswordHash, user.PasswordSalt) = CreatePasswordHash(request.Password);
        await _userRepository.Add(user);
        return Ok();
    }
    [HttpPost("login")]
    public async Task<ActionResult<string>> Login(UserLoginDto request)
    {
        var users = _userRepository.GetUserByUsername(request.Username)
            .Select(cd => new User
            {
                Id = cd.Id,
                PasswordHash = cd.PasswordHash,
                PasswordSalt = cd.PasswordSalt,
                Username = cd.Username,
                UserRole = cd.UserRole
            });
        if (!users.Any() || users.Count() != 1)
        {
            return Unauthorized("Неправильний логін або пароль");
        }
        var user = await users.FirstAsync();
```

```
    bool verifycation = VerifyUser(request.Password, user.PasswordHash, user.PasswordSalt);
    if (!verifycation)
    {
        return Unauthorized("Неправильний логін або пароль");
    }
    var token = CreateToken(user);
    return Ok(token);
}
[HttpGet("userData"), Authorize]
public async Task<ActionResult<UserPersonalInfoDto>> GetUserData()
{
    string Username = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var user = await _userRepository.GetUserByUsername(Username)
        .Select(u => new UserPersonalInfoDto
        {
            Firstname = u.FirstName,
            Lastname = u.LastName,
            Secondname = u.SecondName,
            PhoneNumber = u.PhoneNumber,
        }).SingleAsync();
    return user;
}
[HttpPost("updateUser"), Authorize]
public async Task<IActionResult> UpdateUser(UserPersonalInfoDto request)
{
    bool isParse = Int32.TryParse(User.FindFirstValue(ClaimTypes.Sid), out int id);
    if (!isParse)
    {
        return BadRequest("Не вдалось зчитати айді з токена");
    }
    var user = await _userRepository.Find(id);
    if (!string.IsNullOrEmpty(request.Firstname))
        user.FirstName = request.Firstname;
    if (!string.IsNullOrEmpty(request.Secondname))
        user.SecondName = request.Secondname;
    if (!string.IsNullOrEmpty(request.Lastname))
        user.LastName = request.Lastname;
    if (!string.IsNullOrEmpty(request.PhoneNumber))
        user.PhoneNumber = request.PhoneNumber;
    await _userRepository.Update(user);
    return Ok("Зміни збережо успішно");
}
private async Task<bool> CheckRegistrationInputs(UserRegistrationDto registrationDto)
{
    bool isExist = await _userRepository.GetUserQueryable()
    .Where(cd => cd.Username == registrationDto.Username || cd.Email == registrationDto.Email)
        .AnyAsync();
    if (isExist)
        return false;
    if (registrationDto.Password.Length <= 5)
        return false;
    if (registrationDto.Username.Length <= 2)
        return false;
    if (!IsValidEmailAddress(registrationDto.Email))
        return false;
    return true;
}
private bool IsValidEmailAddress(string address)
    => address != null && new EmailAddressAttribute().IsValid(address);
private string CreateToken(User user)
{
    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, user.Username),
        new Claim(ClaimTypes.Role, user.UserRole.ToString()),
        new Claim(ClaimTypes.Sid, user.Id.ToString())
```

```
        };
        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
            _configuration.GetSection("AppSettings:Token").Value));
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);
        var token = new JwtSecurityToken(
            claims: claims,
            expires: DateTime.Now.AddDays(1),
            signingCredentials: creds);
        var jwt = new JwtSecurityTokenHandler().WriteToken(token);
        return jwt;
    }
    bool VerifyUser(string password, byte[] passwordHash, byte[] passwordSalt)
    {
        using (var hmac = new HMACSHA512(passwordSalt))
        {
            var computedHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
            return computedHash.SequenceEqual(passwordHash);
        }
    }
    (byte[] passwordHash, byte[] passwordSalt) CreatePasswordHash(string password)
    {
        byte[] passwordSalt;
        byte[] passwordHash;
        using (var hmac = new HMACSHA512())
        {
            passwordSalt = hmac.Key;
            passwordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
        }
        return (passwordHash, passwordSalt);
    }
}
```

OrderController.cs :

```
[Route("api/[controller]")]
[ApiController]
public class OrderController : ControllerBase
{
    private readonly IBoardGameRepository gameRepo;
    private readonly IOrderRepository orderRepo;
    private readonly IUserRepository userRepo;
    public OrderController(IBoardGameRepository gameRepo, IOrderRepository orderRepo, IUserRepository userRepo)
    {
        this.gameRepo = gameRepo;
        this.orderRepo = orderRepo;
        this.userRepo = userRepo;
    }
    [HttpPost("create")]
    [Authorize(Roles = nameof(Role.User))]
    public async Task<IActionResult> CreateOrders(IEnumerable<CreateOrderDto> request)
    {
        try
        {
            if (request == null || request.Count() <= 0)
                return BadRequest();
            bool isParse = Int32.TryParse(User.FindFirstValue(ClaimTypes.Sid), out int id);
            if (!isParse)
                return BadRequest("Не вдалось зчитати айді з токена");
            string username = User.FindFirstValue(ClaimTypes.NameIdentifier);
            var user = await userRepo.Find(id);
            if (user.Username != username)
                return Unauthorized("Користувача не знайдено");
            foreach (var order in request)
            {
                var games = await gameRepo.GetByVendor(order.VendorId)
                    .Where(g => order.Items.Select(i => i.Id).Contains(g.Id))
                    .ToListAsync();
                foreach (var item in order.Items)
```

```
            {
                var game = games.SingleOrDefault(g => g.Id == item.Id);
                if (game == null)
                    return BadRequest("Один з товарів не існує");
                if (!game.TimeStamp.SequenceEqual(item.TimeStamp))
                    return BadRequest($"Неправильна часова мітка у {game.Name}, дані не актуальні");
                if (game.Quantity < item.Qty)
                    return BadRequest($"Недостатньо {game.Name} на складі");
                games[games.IndexOf(game)].Quantity -= item.Qty;
            }
            var orderItems = ConvertToEntity(order.Items).ToList();
            orderItems.ForEach(oi => oi.ItemCost = GetPrice(games.Single(g => g.Id == oi.ItemId), oi.Qty));
            var orderEntity = new Order
            {
                UserId = id,VendorId = order.VendorId,
                DeliveryAddress = order.DeliveryAddress,
                Firstname = order.User.Firstname,Lastname = order.User.Lastname,
                Secondname = order.User.Secondname,OrderItems = orderItems,
                PhoneNumber = order.User.PhoneNumber,
                PaymentStatus = PaymentStatus.NotPaid,
                CreationDate = DateTime.UtcNow,
                MessageFromUser = order.MessageFromUser ?? "",
                OrderStatus = OrderStatus.Created
            };
            await orderRepo.Add(orderEntity, persiste: false);
            await gameRepo.UpdateRange(games, persiste: false);
        }
        await orderRepo.SaveChanges();
        return Ok();
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message + ex.InnerException?.Message);
    }
}
[HttpGet("userOrders")]
[Authorize(Roles = nameof(Role.User))]
public async Task<ActionResult<IEnumerable<OrderDetailsDto>>> GetUserOrders()
{
    try
    {
        bool isParse = Int32.TryParse(User.FindFirstValue(ClaimTypes.Sid), out int id);
        if (!isParse)
            return BadRequest("Не вдалось зчитати айді з токена");
        string username = User.FindFirstValue(ClaimTypes.NameIdentifier);
        var user = await userRepo.Find(id);
        if (user.Username != username)
            return Unauthorized("Користувача не знайдено");
        var quryableOrder = orderRepo.GetUserOrders(id);
        var result = await quryableOrder.Select(qo => new OrderDetailsDto
        {
            OrderId = qo.Id,
            VendorName = qo.VendorNavigation.Name,
            DeliveryAddress = qo.DeliveryAddress,
            MessageFromUser = qo.MessageFromUser,
            Items = qo.OrderItems.ConvertToDto().ToList(),
            User = new UserPersonalInfoDto
            {
                Firstname = qo.Firstname,Lastname = qo.Lastname,
                PhoneNumber = qo.PhoneNumber,Secondname = qo.Secondname
            },
            CompletionDate = qo.CompletionDate,CreationDate = qo.CreationDate,
            OrderStatus = qo.OrderStatus,PaymentDate = qo.PaymentDate,
            PaymentStatus = qo.PaymentStatus}).ToListAsync();
        return Ok(result);
    }
```

```
        catch (Exception ex)
        {
            return BadRequest(ex.Message + ex.InnerException?.Message);
        }
    }
    private int GetPrice(Boardgame game, int qty)
    {
        if (game.Discount != null)
            return (int)Math.Ceiling(game.FullPrice * (1 - ((decimal)game.Discount / 100)) * qty);
        return (int)Math.Ceiling(game.FullPrice * qty);
    }
    private IEnumerable<OrderItem> ConvertToEntity(IEnumerable<OrderItemDto> orders)
        => orders.Select(o => new OrderItem { ItemId = o.Id, Qty = o.Qty });
}
```

ДОДАТОК Г
ГРАФІЧНА ЧАСТИНА

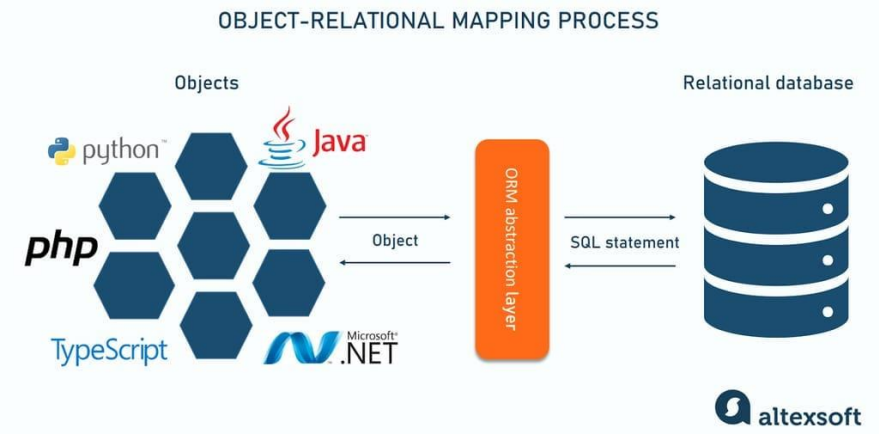

Рисунок Г.1 – Рівень абстракції ORM між об’єктами та таблицями в реляційній базі даних

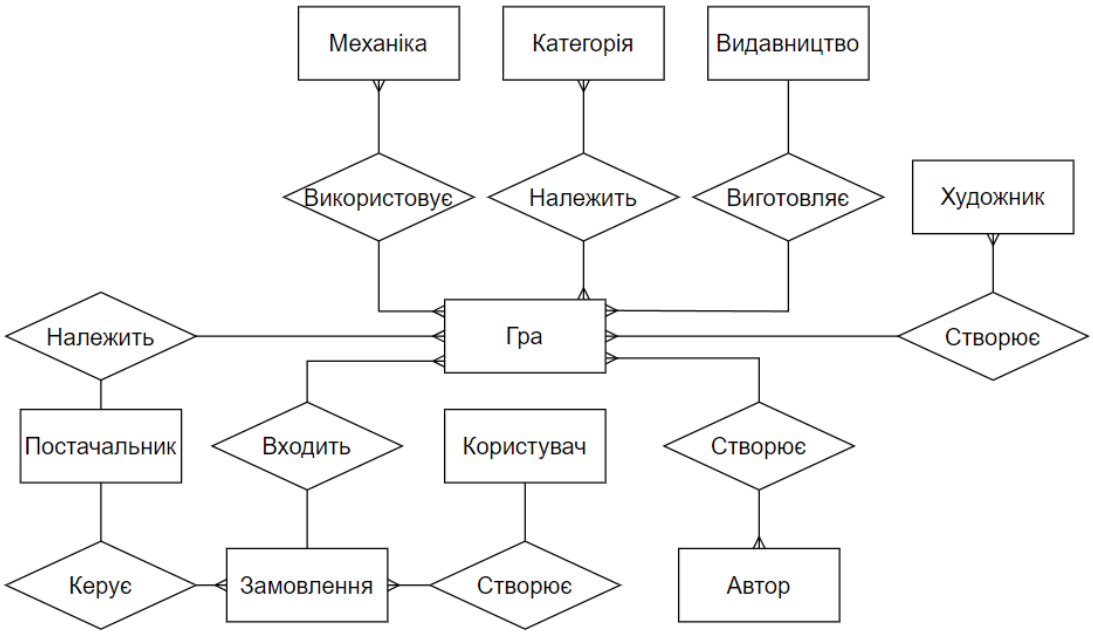

Рисунок Г.2 – ER-модель програмного модулю для продажу настільних ігор

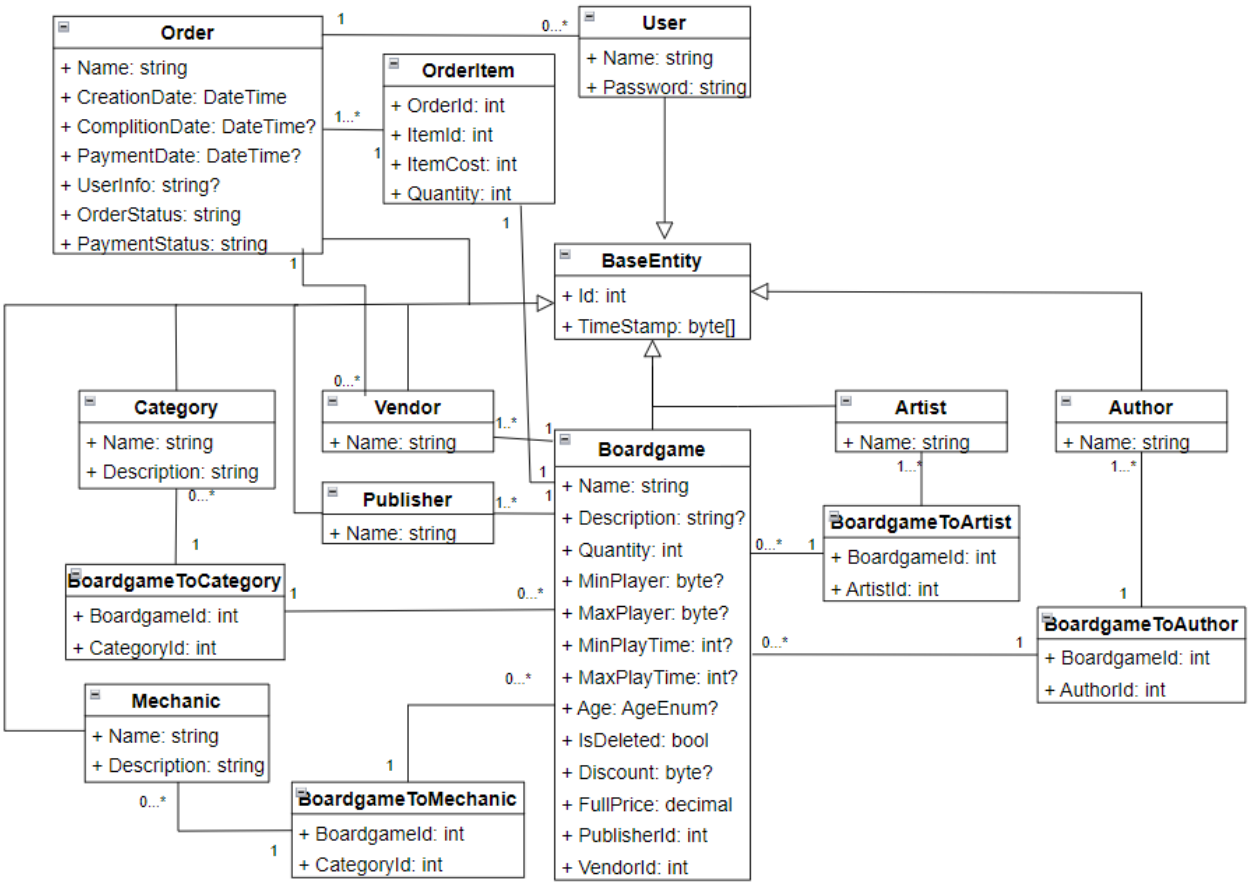

Рисунок Г.3 – UML діаграма класів програмного модуля для продажу настільних ігор

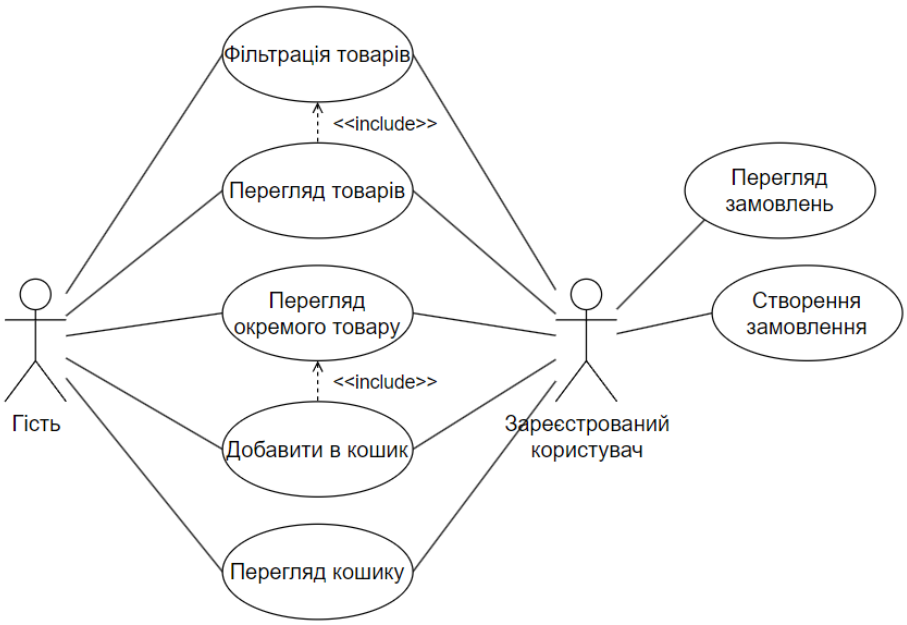

Рисунок Г.4 – діаграма прецедентів для гостя та зареєстрованого користувача

Рисунок Г.5 – діаграма прецедентів для постачальника

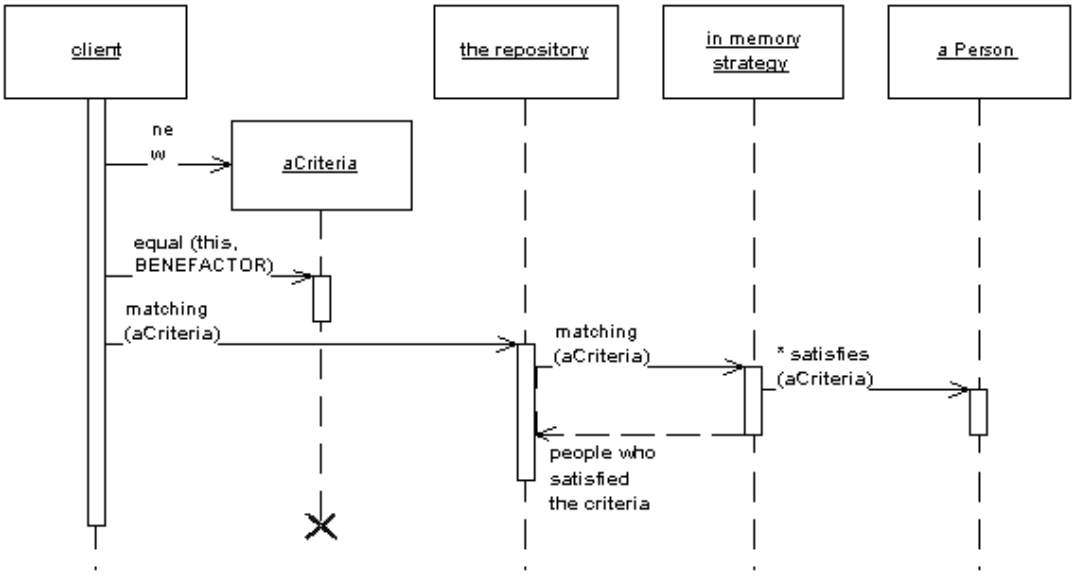

Рисунок Г.6 – демонстрація розміщення репозиторію для певної програми з використанням UML діаграми послідовності

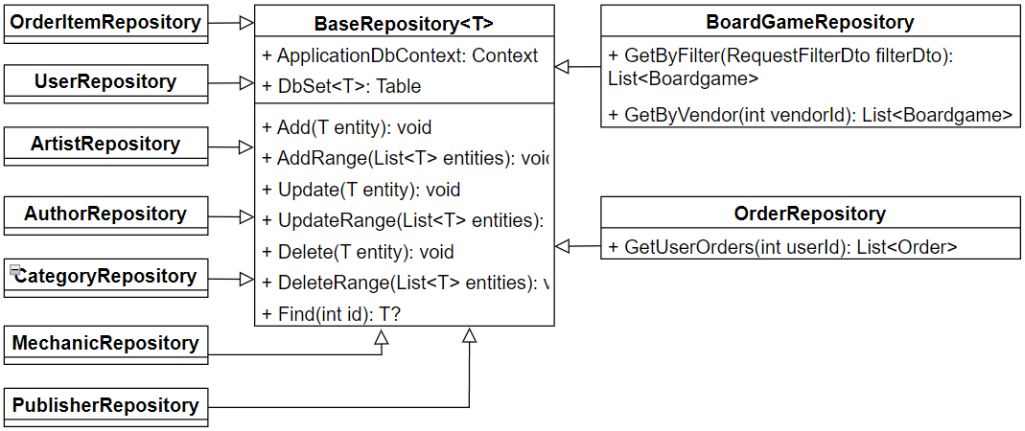

Рисунок Г.7 – UML діаграма класів-репозиторіїв шару доступу до даних

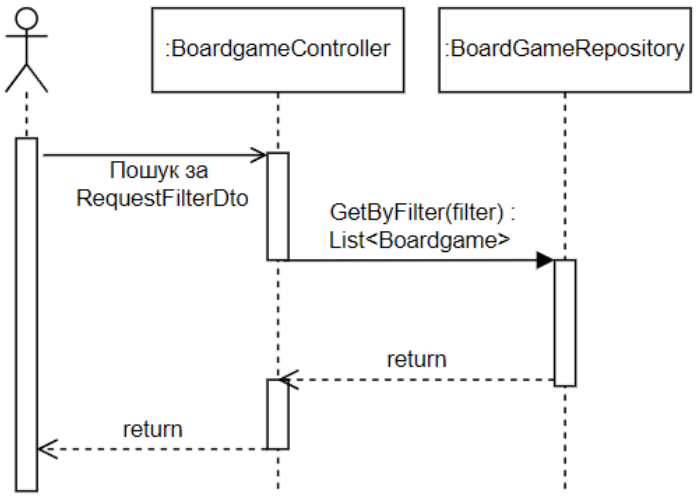

Рисунок Г.8 – UML діаграма послідовності фільтрації настільних ігор

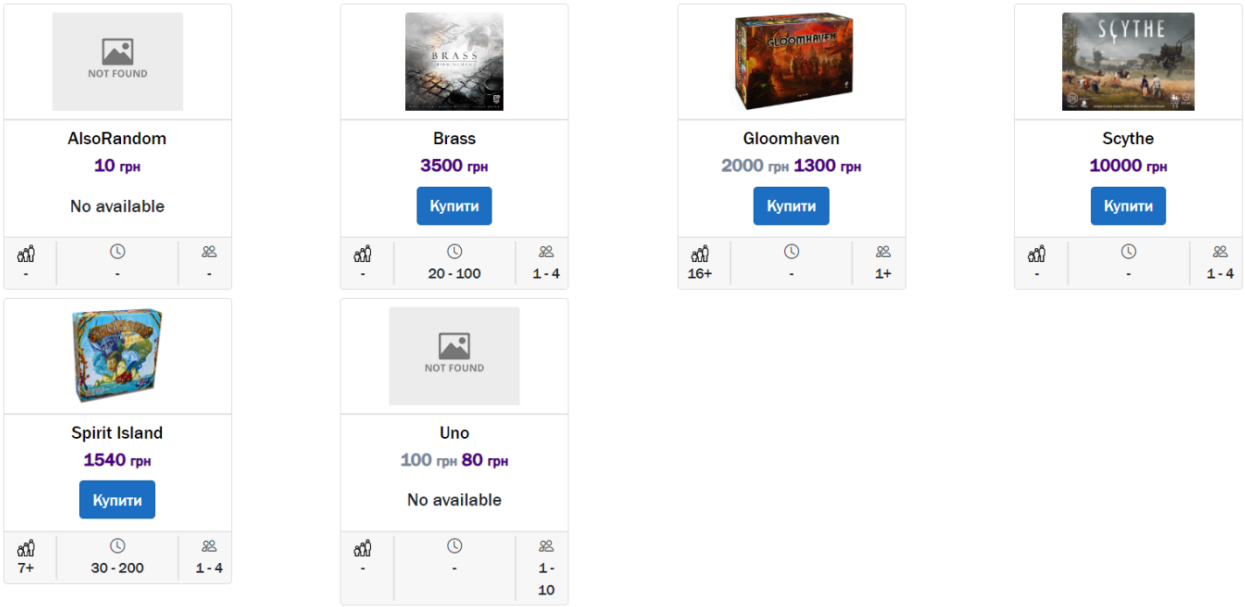

Рисунок Г.9– сітка настільних ігор

Рисунок Г.10 – меню навігації анонімного користувача

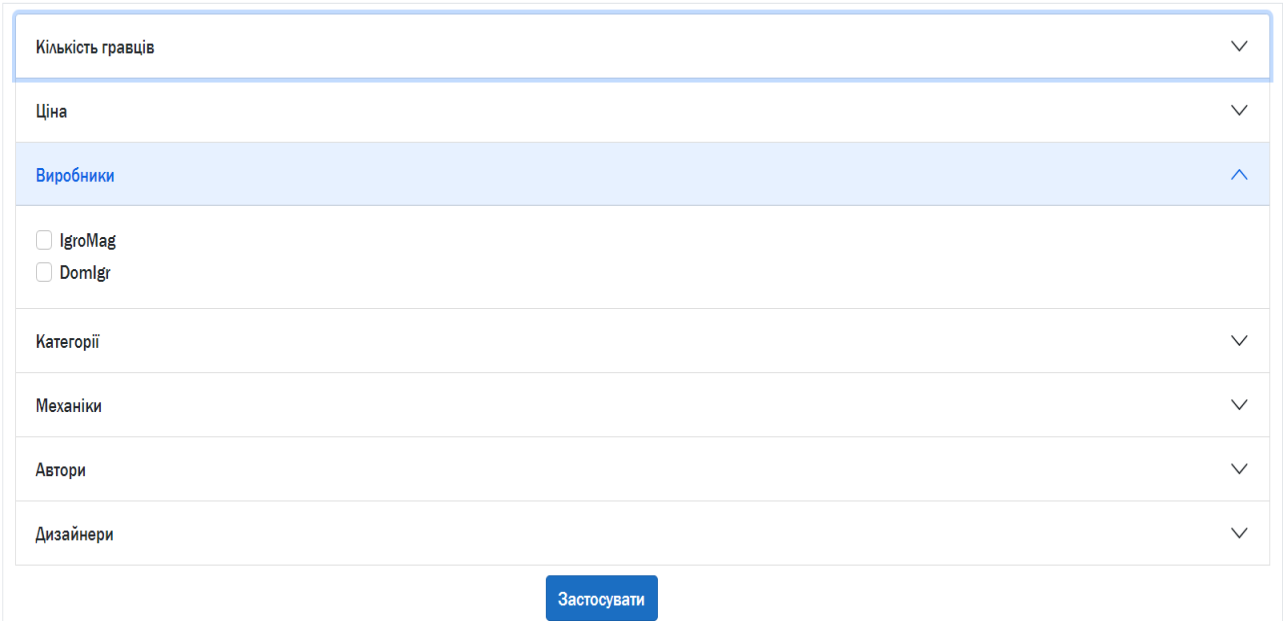

Рисунок Г.11 – фільтри для пошуку настільних ігор

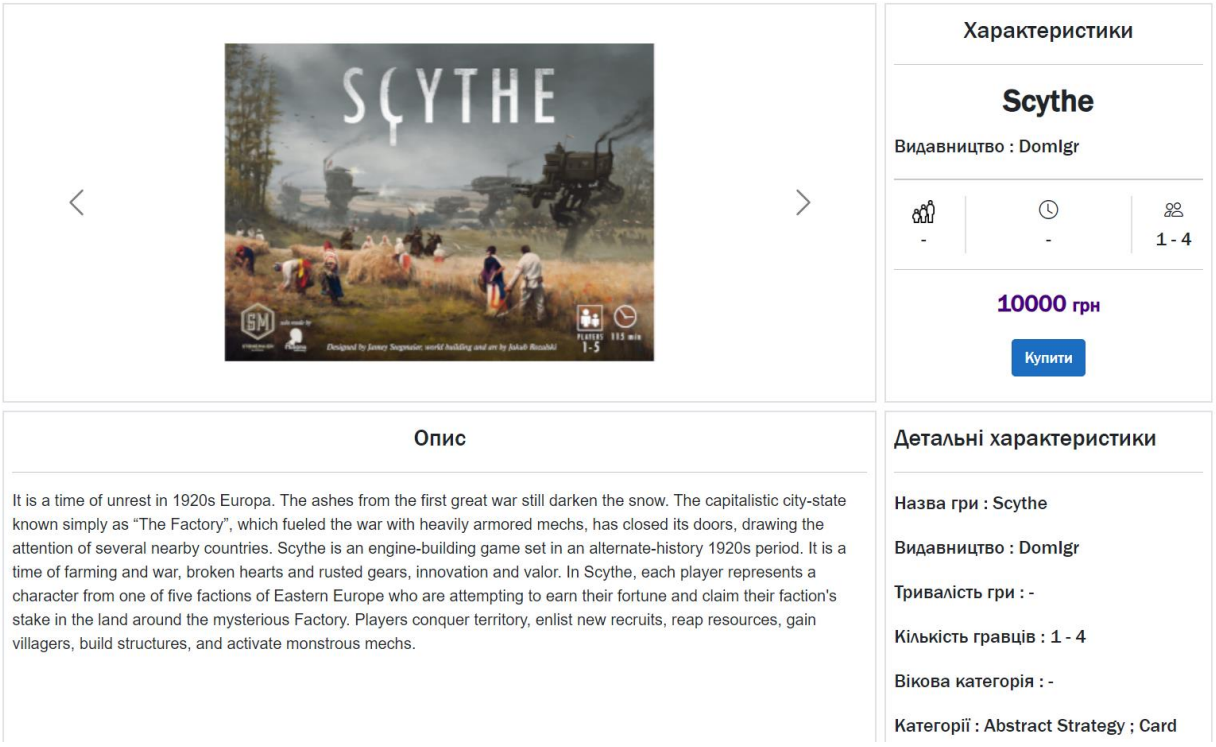

Рисунок Г.12 – детальний опис настільної гри Scythe

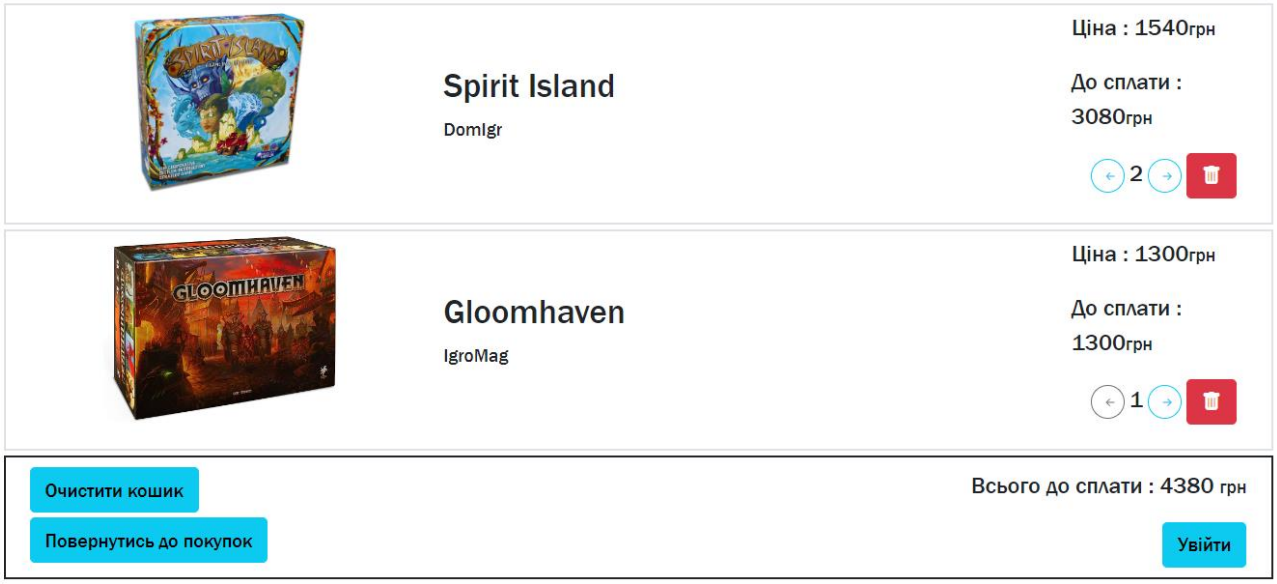

Рисунок Г.13 – кошик користувача

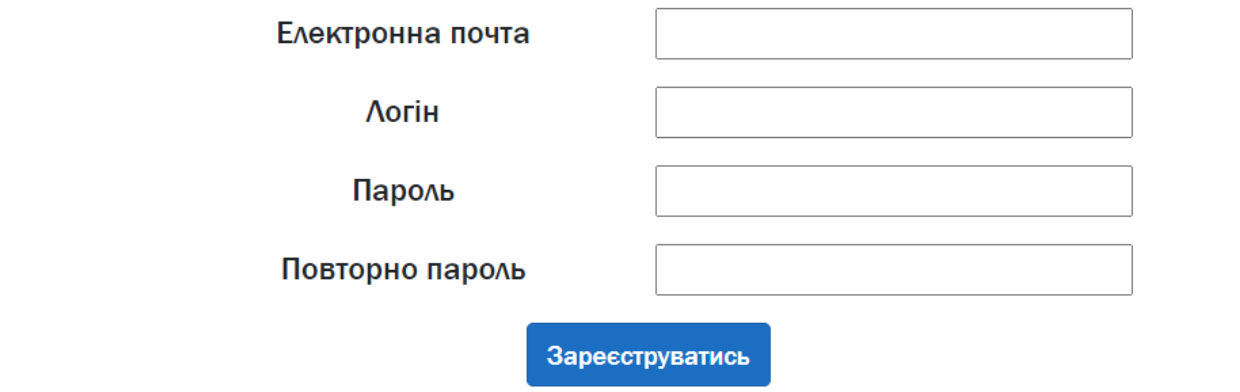

Рисунок Г.14 – форма реєстрації облікового запису

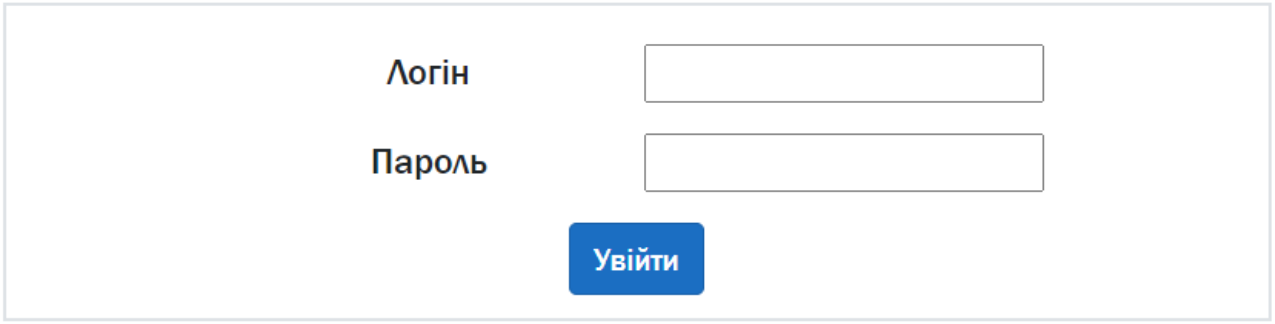

Рисунок Г.15 – форма входу в обліковий запис

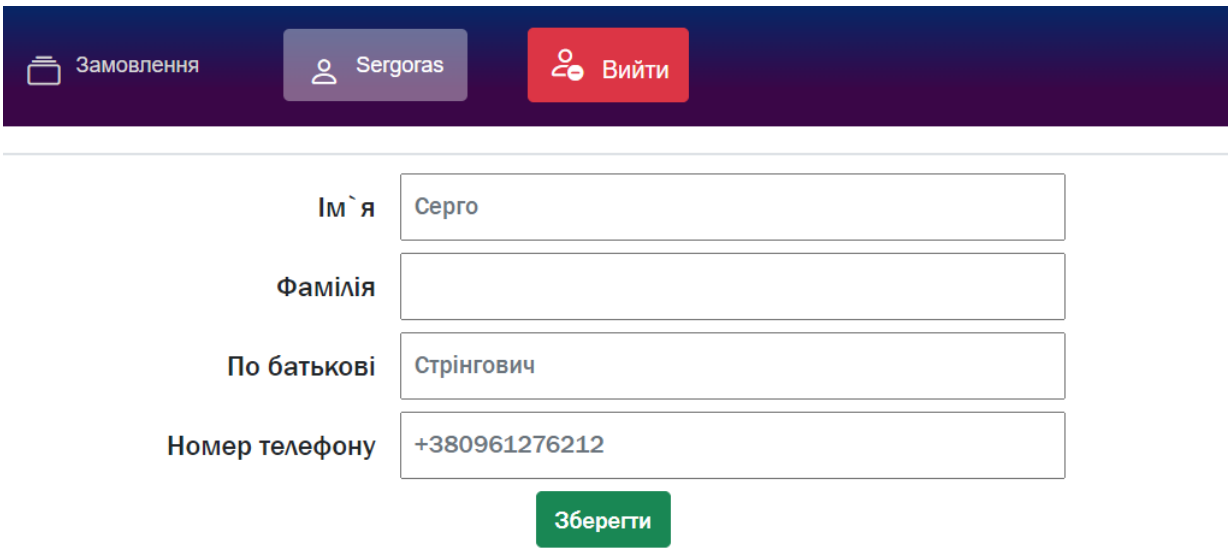

Рисунок Г.16 - додаткова інформація про користувача

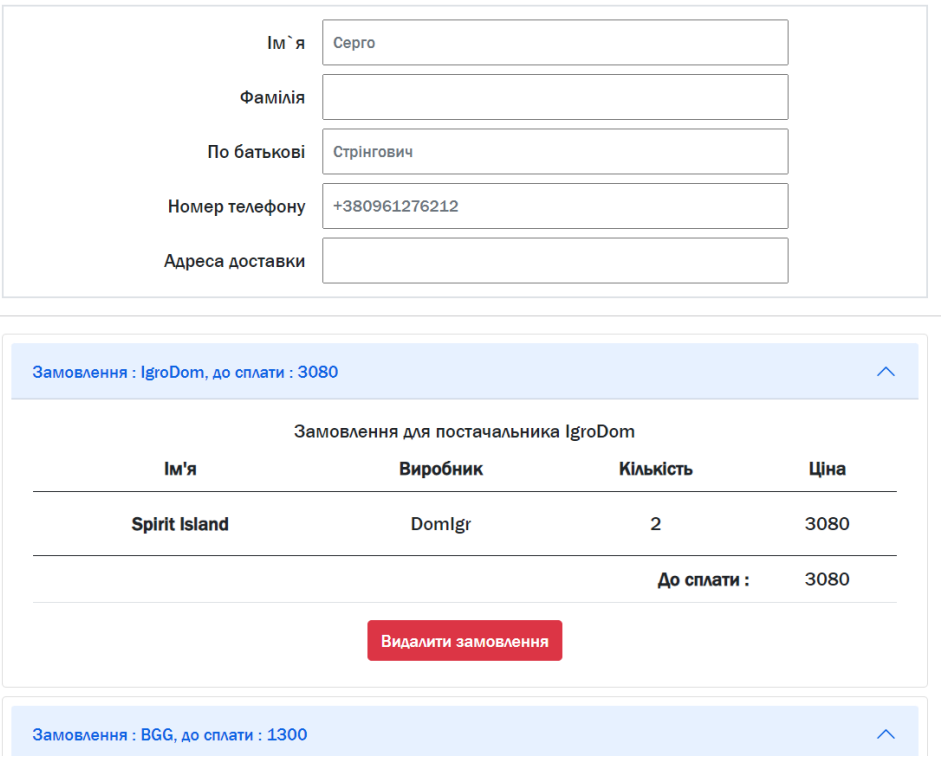

Рисунок Г.17 - форма створення замовлень

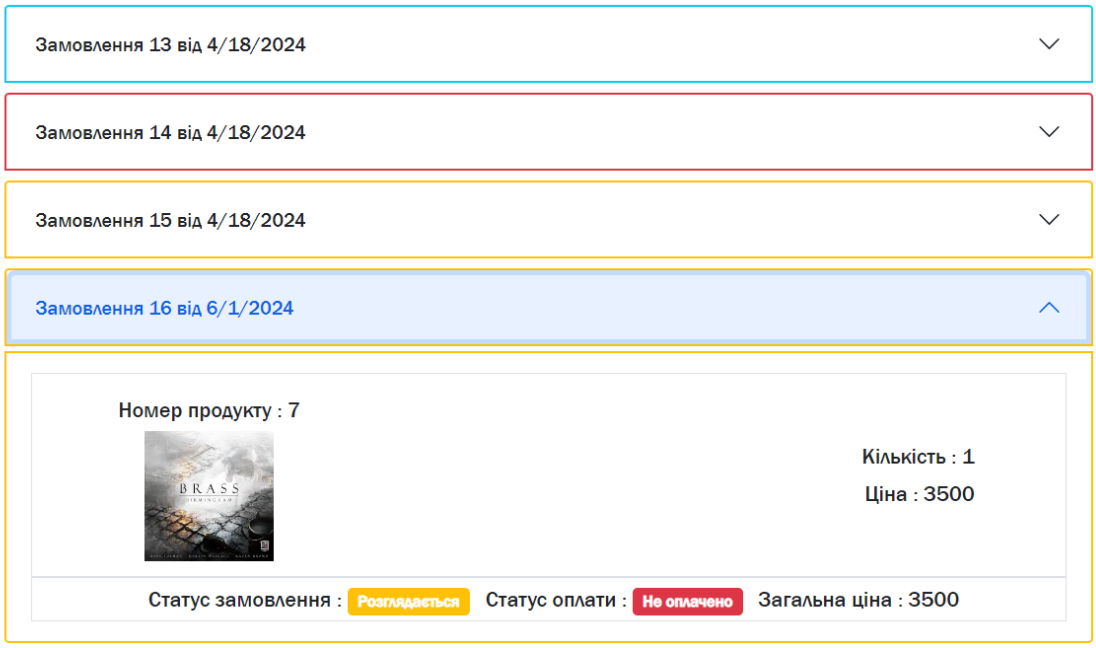

Рисунок Г.18 - список замовлень користувача

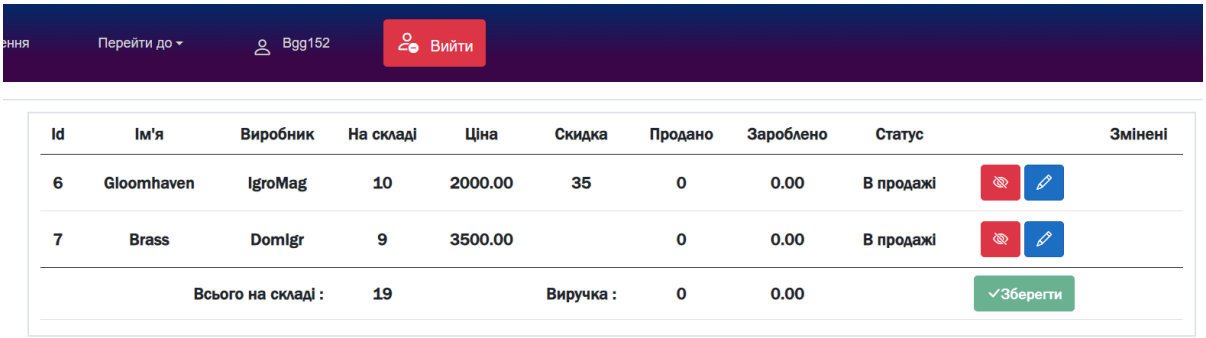

Рисунок Г.19 - список настільних ігор з облікового запису постачальник