Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

**РОЗРОБКА ІНТЕРНЕТ-МАГАЗИНУ ДЛЯ ПРОДАЖУ ІГОР**
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-19-1

Гончаров О.О.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Кіношенко Д.К.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2023 р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Гончарову Олексію Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка інтернет-магазину для продажу ігор

затверджена наказом університету від 15 травня 2023 року № 474 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 22 травня 2023 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література, дані інтернет-мережі, бібліотека з відкритим кодом React.

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналітичний огляд мов програмування та бібліотек для розробки застосунку інтернет-магазину для продажу ігор.

2. Моделювання застосунку інтернет-магазину для продажу ігор.

3. Програмна реалізація застосунку.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми інтернет-магазину для продажу ігор, постановка задачі, тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

| Найменування розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| Консультант з дотримання діючих стандартів та норм | Доцент Творошенко І.С. | | |

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 10.04.2023 | |
| 2 | Аналіз завдання, підбір літератури | 11.04.23-17.04.23 | |
| 3 | Аналіз літератури з досліджуваної проблеми | 18.04.23-20.04.23 | |
| 4 | Аналіз технічних засобів | 21.04.23-30.04.23 | |
| 5 | Моделювання застосунку | 01.05.23-14.05.23 | |
| 6 | Програмна реалізація | 15.05.23-23.05.23 | |
| 7 | Оформлення пояснювальної записки | 24.05.23-26.05.23 | |
| 8 | Перевірка на плагіат | 27.05.23 | |
| 9 | Рецензування | 28.05.23 | |
| 10 | Підготовка презентації та доповіді | 29.05.23-30.05.23 | |
| 11 | Занесення роботи в електронний архів | 31.05.23 | |
| 12 | Попередній захист кваліфікаційної роботи | 31.05.23 | |

Дата видачі завдання 10 квітня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Кіношенко Д.К.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 72 с., 10 рис., 2 дод., 30 джерел.

ВЕБСАЙТ, ОТОЧЕННЯ DOCKER ТА DOCKER COMPOSE, МОВА ГІПЕРТЕКСТОВОЇ РОЗМІТКИ HTML5, КАСКАДНІ ТАБЛИЦІ СТИЛІВ CSS, МОВА ПРОГРАМУВАННЯ JAVASCRIPT, JAVASCRIPT-БІБЛІОТЕКА REACT, JAVASCRIPT ОТОЧЕННЯ NODE.JS, РЕЛЯЦІЙНА БАЗА ДАНИХ POSTGRESQL.

Об’єктом роботи є створення вебсайту інтернет-магазину для продажу комп’ютерних ігор. Буде використовуватися Docker, Docker Compose, JavaScript, Node.Js, Express, Prisma ORM, React, Redux та PostgreSQL для розробки системи.

Метою роботи є створення надійного та ефективного магазину, що дозволяє користувачам легко та швидко знайти та замовити бажану гру.

Сайт має 2 рівні доступу: незареєстрований та звичайний користувач. Незареєстрований користувач може переглядати каталог і вміст кожного товару, зареєструватися або увійти. Звичайний користувач може переглядати каталог, вміст кожного товару, додавати товари в кошик, здійснювати покупки.

У результаті роботи створено вебсайт для продажу ігор з усіма базовими функціями притаманними інтернет-магазинам.

WEBSITE, DOCKER ENVIRONMENT AND DOCKER-COMPOSE, HYPERTEXT MARKUP LANGUAGE HTML5, CASCADING STYLE SHEETS CSS, PROGRAMMING LANGUAGE JAVASCRIPT, JAVASCRIPT LIBRARY REACT, JAVASCRIPT ENVIRONMENT NODE.JS, RELATIVE DATABASE POSTGRESQL.

The object of work is to create a website for an online store selling computer games. Will use Docker, Docker Compose, JavaScript, Node.Js, Express, Prisma ORM, React, Redux, and PostgreSQL to develop the system.

The aim of the work is to create a reliable and efficient store that allows users to easily and quickly find and order the desired game.

The site has 2 levels of access: unregistered and regular user. The unregistered user can browse the catalog and the content of each product, register or log in. The regular user can view the catalog, the content of each product, add products to the cart, make purchases.

As a result, a website for the sale of games was created with all the basic functions of a general online store.

# ЗМІСТ

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

ІМ – інтернет-магазин

ВЗ – вебзастосунок

HTML – HyperText Markup Language (мова розмітки гіпертексту)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

DOM – Document Object Model (об’єктна модель документа)

MVC – Model-View-Controller (модель-вид-контролер)

UDF – Unidirectional Data Flow (односпрямований потік даних)

CBP – Component-Based Programming (компонентно-орієнтоване програмування)

CBFS – Component-Based Folder Structure (структура папок на основі компонентів)

REST – Representational State Transfer (передача репрезентативного стану)

# ВСТУП

Електронна комерція – це процес продажу фізичних і нефізичних товарів за допомогою доступних їм спеціалізованих електронних майданчиків це дає можливість зробити замовлення дистанційно.

Інтернет торгівля дуже добре розвиваються в Україні за даними українського ринку електронної комерції, обсяг продажів стабільно зростає, як і кількість ІМ.

ІМ вже можуть замінити традиційні магазини, у деяких сферах торгівлі відсутність ІМ є стратегічним недоліком. Вебсайт дозволяє компаніям представити інформацію про товари та послуги стисло або розгорнуто. Також застосунок може повідомляти про новини, зміни в ціні або режимі роботи, містити відгуки вдячних клієнтів. Актуальність розробки ВЗ пояснюється наступними факторами:

- швидкість передачі інформації великому колу людей;
- поліпшення іміджу магазину та підвищення популярності;
- наявність зворотного зв’язку з клієнтами;
- реклама і залучення покупців і клієнтів.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1 Огляд останніх досліджень і публікацій

У сучасний час сайти стали масовими. Майже кожна компанія, в тому числі й маленькі фірми, має свою «віртуальну вітрину», а автори нових продуктів чи технологій можуть розраховувати на успіх завдяки рекламі в мережі. З посиленням конкуренції та розвитком сучасних технологій магазинам необхідна побудова інформаційних систем з необхідним інструментарієм для успішного управління бізнесом в сучасних умовах. Активний розвиток Інтернету привів до необхідності створення сайтів для надання різноманітної інформації. Сайти можуть бути досить різноманітними – від невеликих, на яких розміщена коротка інформація про компанію та її послуги, до великих інтернет-каталогів з детальними характеристиками товарів, їх зображеннями та цінами. Зазвичай такий інтернет-каталог створюється, щоб користувач міг знайти докладний опис та зображення товару, тобто він виконує функцію рекламного каталогу товарів. ІМ – це прикладна система, побудована з використанням технології системи електронної торгівлі. Подібно до звичайного магазину, електронний магазин реалізує наступні основні функції: представлення товарів (послуг) покупцю, обробку замовлень, продаж та доставку товарів. ІМ поєднує елементи прямого маркетингу з традиційним магазином. У порівнянні зі звичайною формою транзакцій відмінною рисою ІМ є те, що вони можуть надати велику кількість інформації про товари та послуги, доступні для покупки в будь-який час, незалежно від місця знаходження покупця. Крім того, вони можуть забезпечувати більш ефективний процес продажу та забезпечувати зручний спосіб оплати та доставки товарів. Загалом, сайти та ІМ стали невід’ємною частиною бізнесу в сучасних умовах, допомагаючи компаніям розширювати свій ринок та збільшувати обсяги продажів.

Однією з головних проблем, що виникає при створенні ІМ, є поєднання традиційної торгівлі з Інтернет-технологіями. У звичайній торгівлі покупці можуть оцінити продукцію візуально, але в електронній комерції такої можливості немає. Хоча візуальна інформація зазвичай достатня, психологічні та емоційні фактори можуть грати важливу роль. Проблеми також можуть виникнути з доставкою товару, особливо якщо його ціна досить низька. Однак з кожним роком кількість ІМ збільшується, оскільки це дійсно зручно та вигідно для покупців. ІМ працюють цілодобово і деякі товари можуть продаватися автоматично без участі продавця. Крім того, не потрібно заздалегідь купувати товари, що значно економить місце для зберігання. Відносно продажу ігор можна сказати, що ця ніша ще не дуже заповнена, що робить онлайн платформу для продажу цієї продукції досить конкурентоздатною. Оскільки дана продукція цікава і жінкам, і чоловікам, створення ІМ є актуальним завданням. Метою роботи є розробка ІМ з продажу ігор, який буде використовуватися для комерційної діяльності певного підприємства.

## 1.2 Вибір засобів реалізації

Зараз у веброзробника стоїть безліч завдань, починаючи від створення інтерактивних сайтів для розваг до серйозних бізнес-проєктів, які потребують підвищеної надійності та захисту від несанкціонованого доступу. Для їх виконання потрібні правильно підібрані інструменти мови програмування, фреймворки або системи управління контентом, які стають все більш актуальними.

На сьогодні доступно багато мов програмування, але перевага кожної залежить від контексту завдання. Вибір мови залежить від рівня знань програміста та достатності їх для виконання проєкту.

### 1.2.1 Загальний огляд мов програмування

Мова програмування – це формальна система символів, яка використовується для створення комп'ютерних програм. Кожна мова програмування має свій власний набір команд, синтаксис та семантику, які визначають зовнішній вигляд програми та дії, які виконує комп'ютер. На наведеному графіку показана популярність мов програмування для розробки вебсайтів (рис. 1.1).

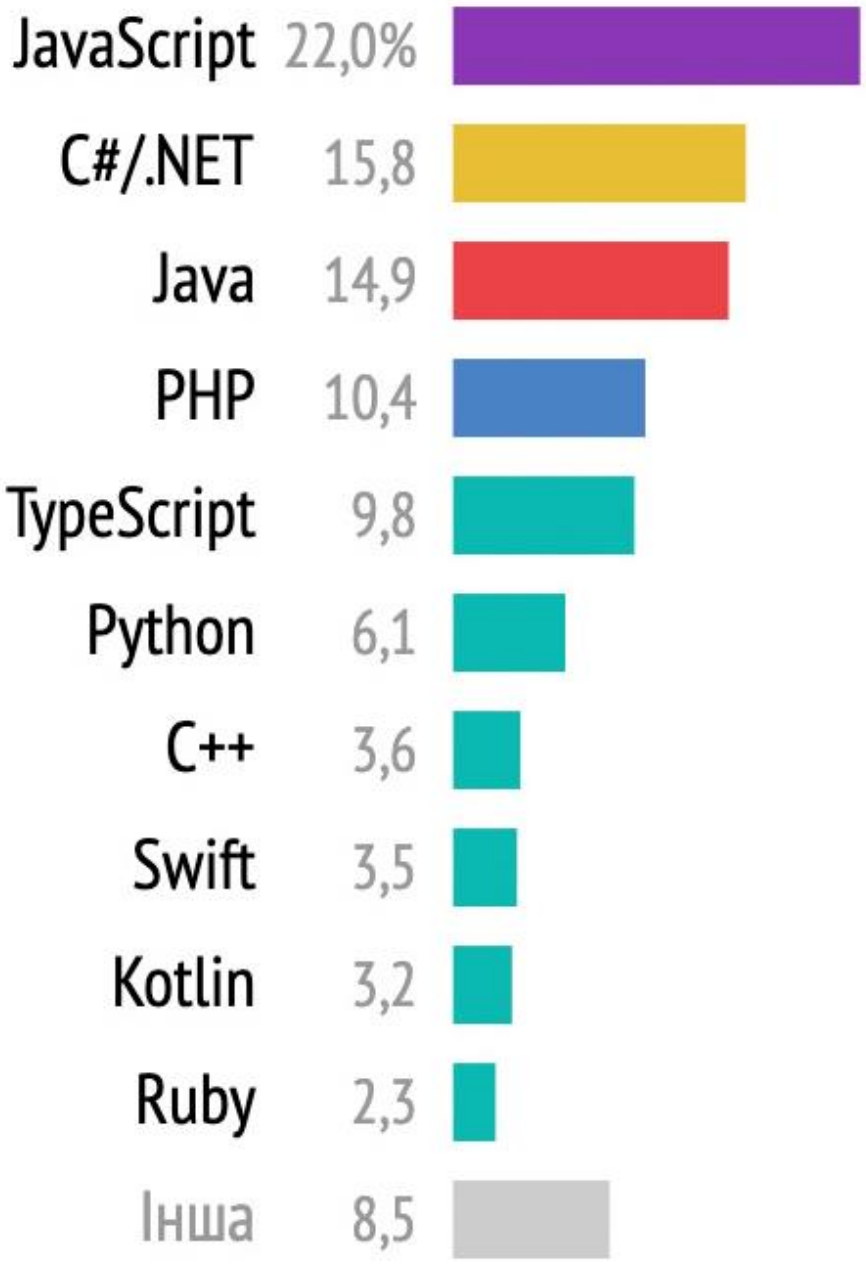


Рисунок 1.1 – Графік популярності мов програмування для веброзробки [1]

Можна розбити всі мови вебпрограмування на дві категорії: клієнтські та серверні мови. Мова клієнта використовується для створення програм, що працюють на клієнтській стороні, тобто у браузері, а серверна мова для програм, що працюють на сервері.

Єдина мова програмування для клієнтської частини вебсторінок – JavaScript. Вона використовується для керування сценаріями відображення вебсторінок. Особливість JavaScript полягає в тому, що елементи вебсередовища можуть змінюватись без перезавантаження сторінки, що дозволяє змінювати колір фону, зображення, створювати нові вікна або відображати повідомлення.

Крім того, розробники використовують мови розмітки HTML і CSS, які необхідні під час створення інтерфейсу. HTML є мовою розмітки гіпертексту, для створення вебсайтів, а CSS – формальна мова, яка використовується для визначення зовнішнього вигляду документів, написаних мовою розмітки. CSS використовується для встановлення шрифтів та макетів на сторінці та інших важливих аспектів вигляду сторінок.

До серверної мови вебпрограмування також відноситься Node.js – це серверна мова програмування, яка базується на JavaScript і користується великою популярністю в сфері веброзробки. Одна з головних особливостей Node.js полягає в тому, що вона працює на стороні сервера, що дозволяє писати код на Back-end частині за допомогою JavaScript.

Це забезпечує єдність мови та коду між фронтендом та бекендом, що спрощує процес розробки. Node.js також є дуже швидкою та ефективною мовою програмування, що забезпечує високу продуктивність ВЗ, які запущені на ній. Це робить Node.js чудовим вибором для розробки великих, масштабованих застосунків та систем, що працюють з великим обсягом даних.

Порівняно з іншими мовами програмування для створення серверних застосунків, Python є новачком у цій галузі. Ця мова є дуже простою для вивчення та динамічною, що дозволяє використовувати її у багатьох галузях.

Синтаксис мови є легким для читання, простим та інтуїтивно зрозумілим, що робить її популярною серед розробників.

Мова програмування Java була створена у 90-х роках і досі є надзвичайно популярною. Вона є стандартом у всіх сферах веброзробки по всьому світу. Java базується на об’єктно-орієнтованому підході і може функціонувати на будь-якій платформі, що робить її дуже функціональною. Java є однією з найбільш популярних мов програмування в світі IT. Вона має широкі можливості та використовується в багатьох сферах, включаючи веброзробку, розробку мобільних застосунків, ігор, наукові дослідження та багато іншого.

C# є однією з найбільш потужних, швидко зростаючих та популярних мов програмування в сфері IT. На сьогоднішній день на C# написано широкий спектр програм, від невеликих застосунків для робочого столу до масштабних вебпорталів та вебсервісів, які обслуговують мільйони користувачів щодня. Одним з найбільших переваг C# є його потужність та гнучкість. Ця мова програмування підтримує різноманітні парадигми програмування, включаючи об’єктно-орієнтоване, функціональне та асинхронне програмування. Також вона підтримує різні платформи, включаючи Windows, macOS та Linux, що дозволяє розробникам створювати програми для різних операційних систем.

PHP – це одна з найпопулярніших мов програмування в IT-індустрії. Вона використовується для створення різноманітних ВЗ, від невеликих сайтів до великих вебплатформ з мільйонами користувачів. PHP проста у вивченні та має зрозумілий синтаксис, що дозволяє швидко розробляти програми. Крім того, PHP – це відкрите програмне забезпечення, що означає, що вона є безкоштовною та доступною для всіх. Ці якості роблять PHP привабливою мовою для розробників, які шукають ефективний та простий спосіб створення ВЗ.

TypeScript – це мова програмування, яка заснована на JavaScript і надає додаткові функції для підтримки більших проєктів та більшої

розширюваності коду. TypeScript є популярним вибором для розробки ВЗ, застосунків для мобільних пристроїв та настільних застосунків. Він має багато інструментів для підтримки розробки, включаючи сильну типізацію, поліморфізм та інші властивості, які дозволяють створювати більш безпечний та стабільний код.

Інші мови не дуже підходять під розробку ІМ через специфічне використання цих мов для веброзробки або. Існує декілька причин, чому інші мови програмування не є найкращим вибором для розробки ІМ. По-перше, ці мови мають складну синтаксис та вимагають від розробника високої кваліфікації, що може збільшити час розробки та зробити процес більш витратним. Крім того, вони не забезпечують такого рівня підтримки як мови віще.

Для ІМ найоптимальніша мова програмування буде JavaScript. Ця мова є надзвичайно популярною та використовується веброзробниками всіх країн.

По-перше, це дуже проста мова програмування для вивчення з легким синтаксисом та мінімальною кількістю коду, необхідного для створення функціональних програм. Це дозволяє швидко розпочати розробку власних ВЗ.

По-друге, JavaScript має велику кількість ресурсів для навчання та документації завдяки відкритому коду, а також багато фреймворків та бібліотек, таких як React та Angular, побудованих на JavaScript, що забезпечує зручний та ефективний розвиток високоякісних ВЗ.

По-третє, JavaScript підтримує асинхронне програмування, що дозволяє розробникам створювати високопродуктивні ВЗ, які можуть обробляти багато запитів одночасно. Це особливо важливо для ВЗ, які мають багато взаємодій з користувачем.

Таким чином, JavaScript разом з Node.js є гарним варіантом, оскільки ці мови мають простий синтаксис, велику підтримку та можливості асинхронного програмування. Це дозволить нам швидко створити високоякісні ВЗ та досягти успіху у цьому.

### 1.2.2 Загальний огляд бібліотек та фреймворків

Фреймворк – це набір бібліотек, який надає розробникам можливість ефективно створювати програми за допомогою заздалегідь визначених структур та алгоритмів. Кожен фреймворк має свій власний набір функцій та інтерфейсів, які допомагають зменшити кількість коду, необхідного для розробки програми. На наведеному графіку показана популярність фреймворків для розробки вебсайтів, що дозволяє розробникам вибрати той, який найбільш підходить для їхнього проєкту.

Фреймворки можуть бути написані для різних мов програмування, таких як Python, Java, Ruby, PHP та інші, тому вони дозволяють розробникам працювати з різними технологіями та мовами програмування. Це зробило фреймворки дуже популярними серед розробників ВЗ та вебсайтів. Одним з основних переваг використання фреймворків є швидкість розробки. Завдяки готовим структурам та інтерфейсам, розробникам не потрібно витрачати час на написання коду з нуля.

Крім того, фреймворки зазвичай мають високу надійність та безпеку, оскільки вони підтримуються та оновлюються розробниками. В будь-якому випадку, використання фреймворків може допомогти розробникам зосередитися на розв’язанні конкретних завдань, замість витрачення часу на написання загальних функцій та алгоритмів з нуля.

На відміну від мов програмування, бібліотеки є набором готових функцій та інструментів, які можна використовувати у програмуванні, щоб скоротити час розробки та полегшити процес програмування. Бібліотеки надають можливість розробникам використовувати готові рішення та функції для вирішення певних завдань, замість написання коду з нуля. На даний час існує багато різних бібліотек для різних мов програмування, які дозволяють розробникам створювати більш складні та потужні програми з меншими затратами часу та зусиль.

На наведеному графіку показана популярність мов бібліотек та фреймворків для розробки клієнтської частини вебсайтів (рис. 1.2).

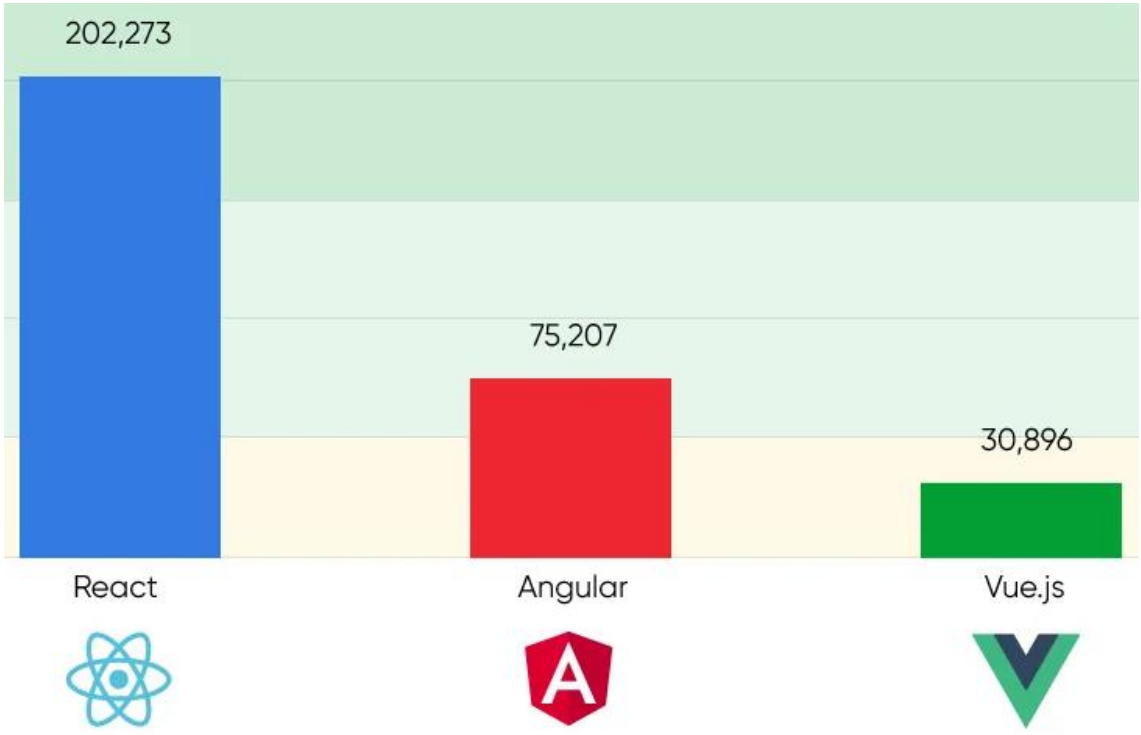


Рисунок 1.2 – Графік популярності бібліотек та фреймворків для розробки клієнтської частини вебсайтів [2]

React є однією з найбільш потужних та популярних бібліотек для розробки вебінтерфейсів у сфері IT. Її використання дозволяє розробникам створювати динамічні та взаємодіючі інтерфейси швидко та ефективно. За допомогою React можна створювати складні компоненти з низкою залежністю, що дозволяє розробникам легко перевикористовувати код та підтримувати його. React також має велику спільноту, яка активно працює над розвитком бібліотеки та надає багато корисних «інструментів для розробників» [3–9].

Angular є одним з найбільш потужних та популярних фреймворків для розробки вебінтерфейсів у сфері IT. Використання Angular дозволяє розробникам створювати динамічні та складні ВЗ швидко та ефективно. Фреймворк має великий набір інструментів для створення компонентів та

модулів, що дозволяє розробникам легко перевикористовувати код та підтримувати його. Angular також забезпечує високий рівень безпеки, що є особливо важливим для розробки ВЗ.

Vue.js є зручною та легкою у використанні бібліотекою для розробки вебінтерфейсів. Вона надає можливість швидко створювати інтерактивні інтерфейси та компоненти, що дозволяє ефективно взаємодіяти з користувачем. За допомогою Vue.js можна створювати невеликі та середні проєкти, а також складні застосунки з великою кількістю даних. Бібліотека є дуже популярною серед розробників, а також має велику та активну спільноту, яка постійно працює над розвитком та покращенням Vue.js.

React є одним з найпопулярніших фреймворків для розробки ВЗ на JavaScript. Його основна перевага полягає у тому, що він дозволяє розробникам створювати складні інтерфейси ВЗ, зберігаючи при цьому простоту та читабельність коду.

Також React використовує концепцію компонентів, що дозволяє розбивати складні інтерфейси на менші блоки, які можна повторно використовувати в інших частинах застосунку. Це дозволяє розробникам зосередитись на створенні функціональної частини застосунку, замість витрачення часу на розробку складних інтерфейсів.

Ще однією перевагою React є те, що він використовує віртуальний DOM. Це означає, що React не маніпулює реальним DOM напряму, а замість цього працює з віртуальною копією DOM, що дозволяє зменшити кількість звернень до реального DOM та зберегти час рендерингу.

React також має велику спільноту розробників та багато ресурсів для навчання та документації. Більшість великих компаній використовують React для своїх ВЗ, що свідчить про його популярність та ефективність.

Таким чином React стає хорошим варіантом для ІМ. Він дозволяє створювати переваги розбитих на компоненти застосунків та використовувати віртуальний DOM для зменшення часу рендерингу.

## 1.3 Постановка задачі

Основою функціональності магазину є взаємопов’язані компоненти, які ретельно розроблені та розподілені на окремі функціональні блоки. ІМ надає користувачам можливість перегляду широкого асортименту комп’ютерних ігор, замовлення та оплати їх в зручний для них спосіб. Будуть використовуватися сучасні технології, щоб забезпечити клієнтам швидку та безпечну доставку гри.

Об’єктом роботи є створення вебсайту інтернет-магазину для продажу комп’ютерних ігор. Буде використовуватися Docker, Docker Compose, JavaScript, Node.Js, Express, Prisma ORM, React, Redux та PostgreSQL для розробки системи.

Метою роботи є створення надійного та ефективного магазину, що дозволяє користувачам легко та швидко знайти та замовити бажану гру.

Для досягнення мети необхідно вирішити такі завдання:

- авторизація;
- каталог товарів;
- категорії товарів;
- жанри товарів;
- пошук товару: по категоріям, по жанрам, по назві;
- кошик;
- оформлення замовлення.

# 2 ОБҐРУНТУВАННЯ ВИБРАНИХ МЕТОДІВ

## 2.1 Методика контейнеризації

Метод контейнеризації ВЗ зазвичай включає в себе використання платформи контейнерів, таких як Docker. Контейнер – це віртуальне середовище, яке включає в себе всі необхідні залежності та бібліотеки, необхідні для «роботи програми» [10–12].

Кроки для контейнеризації ІМ будуть включають наступне:

Крок 1. Визначте необхідні компоненти та сервіси: перш ніж створювати контейнер, потрібно визначити, які компоненти та сервіси потрібні для роботи ВЗ. Це може включати вебсервери, бази даних, програмні бібліотеки та інші компоненти.

Крок 2. Створення Dockerfile: Dockerfile – це текстовий файл, який містить інструкції для створення образу контейнера. Він містить список залежностей, налаштування та інші деталі, необхідні для створення контейнера.

Крок 3. Створення образу контейнера: після створення Dockerfile, цей файл використовується для створення образу контейнера. Образ – це статичний файл, який містить всі необхідні залежності та конфігурацію для створення та запуску контейнера.

Крок 4. Запуск контейнера: після створення образу контейнера, можна запустити його на будь-якій підтримуваній платформі, яка має встановлений Docker.

Крок 5. Налаштування мережі та обмін даними: після запуску контейнера, може бути налаштовано мережу та обмін даними з ВЗ.

Крок 6. Оновлення та масштабування: при необхідності, можна оновити та масштабувати ВЗ, додавши нові компоненти або змінюючи налаштування.

Загалом, контейнеризація є потужним інструментом для розгортання та керування інтернет-магазином, який дозволяє забезпечити швидке та ефективне розгортання, зменшити ризик виникнення проблем, забезпечити безпеку та масштабованість системи.

Для ВЗ гарним варіантом буде стек з трьох контейнерів, а саме для Front-end, Back-end и бази даних.

Розберемо кожен з контейнерів.

Першим контейнером буде Front-end (React). Він відповідає за відображення інтерфейсу користувача через браузер. Даний контейнер заснований на фреймворку React і дозволяє створювати динамічні інтерфейси. Взаємодія контейнера Front-end (React) з контейнером Back-end (Node.js) дозволяє отримувати дані для відображення на інтерфейсі користувача.

Використання контейнера Front-end (React) дозволяє нам отримувати швидкий доступ до інформації та комфортно взаємодіяти з сайтом. Розробники можуть ефективно створювати складні інтерфейси за допомогою React з високим рівнем інтерактивності, що поліпшує досвід роботи з магазином для користувача. Використання контейнера Front-end (React) дозволяє ІМ бути більш гнучким і адаптованим до потреб користувачів, що забезпечує зростання продажів та задоволення клієнтів.

Контейнер Back-end (Node.js) – це обчислювана частина ІМ та обробляє запити від користувачів. Він використовує мову програмування JavaScript та платформу Node.js для обробки запитів, зв’язку з базою даних та формування відповідей для Front-end (React). Контейнер Back-end (Node.js) взаємодіє з контейнером Бази даних (PostgreSQL), щоб отримувати та зберігати дані, необхідні для функціонування інтернет-магазину.

Контейнер Бази даних (PostgreSQL) є ключовим компонентом ІМ, який забезпечує зберігання та оновлення даних. Він використовує систему управління базами даних PostgreSQL, що забезпечує швидкий доступ до даних та їх безпеку. Контейнер Бази даних (PostgreSQL) взаємодіє з

контейнером Back-end (Node.js), щоб забезпечити отримання та зберігання необхідних даних для функціонування ІМ. Використання контейнера Бази даних (PostgreSQL) дозволяє ІМ бути ефективним та надійним, що підвищує задоволеність користувачів та збільшує продажі.

Один із способів організації контейнерів та їх взаємодії це використання Docker Compose. Docker Compose – це інструмент, який дозволяє описати та запустити багатоконтейнерні застосунки Він дозволяє легко налаштувати взаємодію між контейнерами та забезпечує їх одночасний запуск. Конфігурацію можна зробити у файлі docker-compose.yml, що зручно для розгортання та керування контейнерами.

Крім того, він забезпечує безпеку застосунку, дозволяючи розділити його на декілька контейнерів та ізолювати їх один від одного. Це дає можливість запускати окремі контейнери з різними сервісами та налаштуваннями, а також забезпечує легкість управління та масштабування контейнерів. Ще Docker Compose підтримує масштабування застосунків, що дозволяє легко збільшувати кількість контейнерів за потребою. На наведеному рисунку показано різницю між окремим та кількома контейнерами (рис. 2.1).

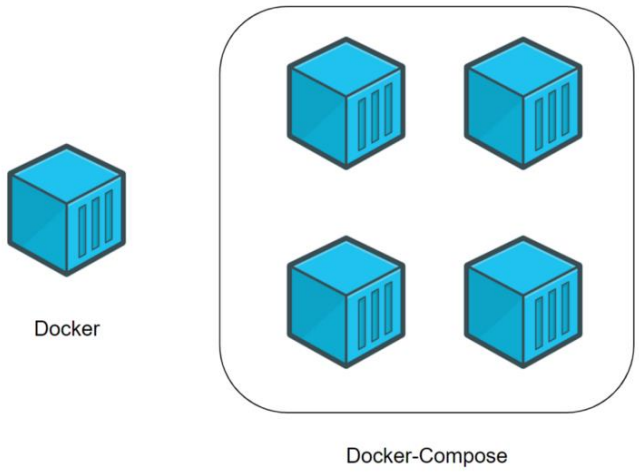


Рисунок 2.1 – Docker (окремий контейнер) та Docker Compose (кілька контейнерів)

Використання Docker Compose у ВЗ має багато переваг, тому його буде використовувано для вибраного набору контейнерів та взаємодії між ними, як це показано на рисунку 2.2.

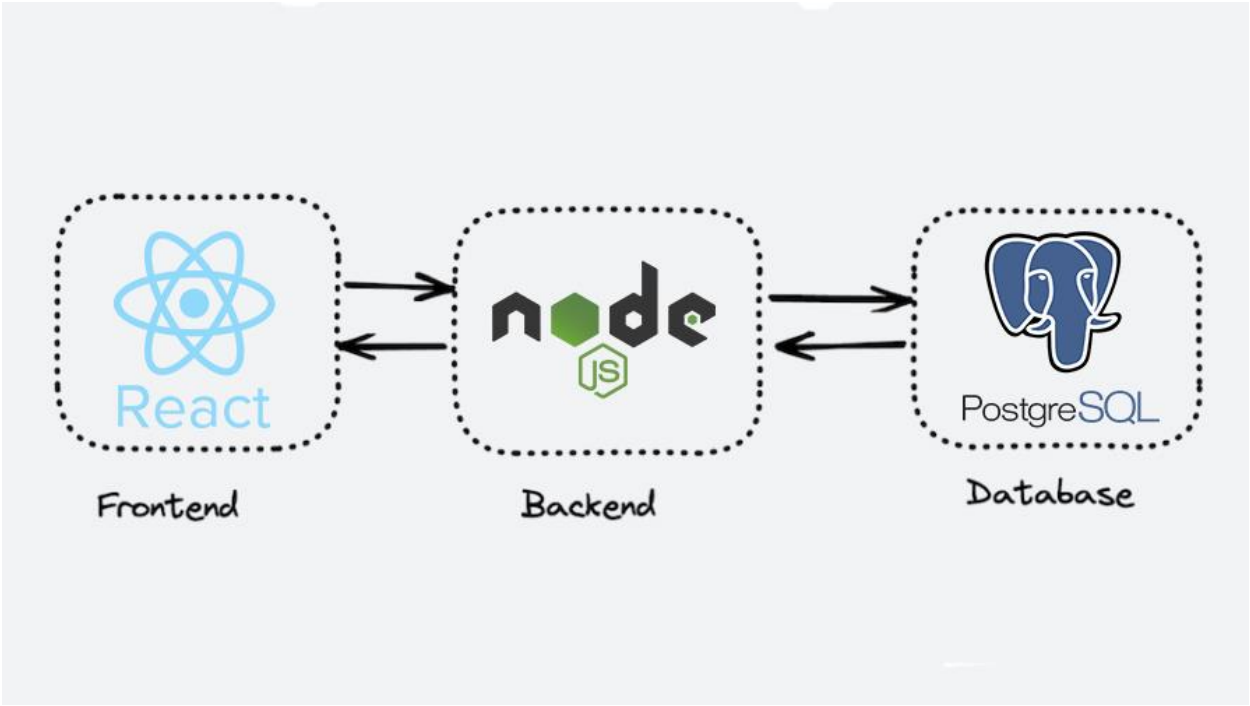


Рисунок 2.2 – Набір контейнерів та взаємодія між ними

## 2.2 Методики Front-end частини

Основою Front-end частини ВЗ є бібліотека React, саме вона може взаємодіяти з усіма Front-end технологіями такі як HTML, CSS і JavaScript , проте його основна функція полягає в рендерингу компонентів.

Коли розробник створює компоненти на React, він використовує мову JavaScript для опису, як компонент повинен виглядати та працювати. Використовуючи React, розробник може створювати різноманітні компоненти, такі як кнопки, форми, списки тощо. Кожен компонент можна розглядати як окремий блок, який можна перевикористовувати в різних частинах сторінки.

Крім того, React може взаємодіяти з HTML і CSS, що використовуються для створення розмітки та стилів на сторінках. HTML використовується для створення структури сторінки, а CSS – для оформлення і стилізації. React також дозволяє використовувати CSS-препроцесори, такі як SASS і LESS, для зручної роботи зі стилями. Розробник може використовувати React, щоб відображати компоненти на сторінці відповідно до розмітки та стилів, які він вже створив.

React також може взаємодіяти з JavaScript, який може бути використаний для додаткової функціональності, наприклад, для зміни стану компонентів або взаємодії з іншими елементами на сторінці.

Загалом, React може взаємодіяти з HTML, CSS і JavaScript, щоб створювати динамічні та взаємодіючі інтерфейси користувача. Він забезпечує зручний спосіб створення компонентів та оптимізує рендеринг, щоб забезпечити швидку продуктивність сторінки.

React не наказує певної методології, але в React-спільноті часто використовуються наступні методології як CBP, UDF, Flux, Redux, React Hooks, CBFS [13–19].

Методологія CBP – це підхід до розробки програмного забезпечення, який базується на створенні програмних компонентів. Компоненти є окремими незалежними частинами програми, які можуть бути повторно використані в інших частинах програми або навіть в інших програмах. Кожен компонент відповідає за свою функціональність, має свій стан та інтерфейс взаємодії з іншими компонентами.

За допомогою CBP можна значно зменшити складність коду та покращити його читабельність, оскільки програмісти можуть працювати над окремими компонентами незалежно один від одного. Крім того, використання компонентів дає змогу зосередитись на бізнес-логіці програми, а не на технічних деталях реалізації.

У розробці ВЗ з використанням CBP найбільш популярним фреймворком є React. У React кожен компонент відповідає за свою частину

UI та має свій власний стан. Разом з цим, React також забезпечує зручний механізм взаємодії між компонентами, що дозволяє швидко та ефективно розробляти ВЗ.

Одним з основних принципів СВР є принцип інверсії залежностей, який полягає в тому, що компоненти не повинні залежати один від одного безпосередньо. Замість цього, компоненти повинні спілкуватись між собою за допомогою визначеного інтерфейсу та подій. Це дозволяє робити компоненти більш масштабованими та забезпечує зручне тестування коду.

Методологія UDF є підходом до розробки програмного забезпечення, який полягає в організації потоку даних в одному напрямку. У цьому підході дані передаються від верхнього компонента до нижнього, а не в обидва напрямки, як у традиційному підході MVC. Це означає, що вся взаємодія між компонентами здійснюється через один централізований об’єкт, який називається Store.

У методології UDF використовується бібліотека Flux, яка дозволяє забезпечити централізоване керування потоком даних у застосунку. У цьому підході компоненти надсилають події (actions) в Store, який змінює свій стан і повідомляє про це всіх підписаних на нього компонентів (views). Компоненти views можуть зчитувати дані лише з Store і не можуть змінювати його стан безпосередньо. Якщо необхідно змінити стан застосунку, це може бути зроблено лише через відправку нової події в Store.

За допомогою методології UDF можна побудувати застосунки, які є більш простими в розумінні та супроводженні, оскільки весь потік даних визначений і централізований. Також цей підхід дозволяє забезпечити більшу масштабованість застосунку, оскільки при додаванні нових функцій не потрібно змінювати структуру даних всього застосунку, достатньо додати новий компонент та його взаємодію зі Store.

Flux – це архітектурна методологія для створення користувацьких інтерфейсів ВЗ. Flux було створено в компанії Facebook для поліпшення

масштабованості та забезпечення більш простого налагодження коду в складних ВЗ.

Основна ідея Flux полягає в тому, щоб спростити управління станом програми. Flux використовує суворий односпрямований потік даних (UDF), що означає, що дані рухаються тільки в одному напрямку: від вхідних даних, через дії користувача, до змін у стані та відображення на екрані. Це дає змогу створювати прості та передбачувані застосунки, адже вся логіка управління даними зосереджена в одному місці, а кожен елемент застосунку знає тільки про свій стан і ніяк не впливає на інші елементи.

У Flux стан програми представлено у вигляді одного об’єкта, який називається Store. Кожен компонент програми може отримувати доступ тільки до свого стану, який передається через властивості (props). Коли користувач взаємодіє з елементами застосунка, це викликає дію (Action), яка передає дані в Dispatcher. Dispatcher потім надсилає дію всім зареєстрованим Store, які оновлюють свій стан і сповіщають усі компоненти, які використовують їхній стан, про необхідність оновлення.

Таким чином, Flux забезпечує чітку структуру застосунка та спрощує його супровід і розширення. Однак, реалізація Flux вимагає більше коду, ніж простіші методології, і може бути складною для розуміння початківцями-розробниками. Наразі існує безліч бібліотек і фреймворків, які надають реалізацію Flux для JavaScript, такі як Redux, Alt.js і Reflux [20].

Redux – це бібліотека управління станом для JavaScript застосунків, заснована на ідеях Flux. Вона використовується для управління станом програми в єдиній, передбачуваній і незмінній формі.

Redux пропонує суворий контроль стану програми через єдиний об’єкт стану (store), який зберігається в пам’яті програми. Будь-які зміни стану можуть бути зроблені тільки через певні дії (actions), які є чистими функціями, що повертають новий об’єкт стану.

Стан у Redux є незмінним, тому кожна зміна створює новий об’єкт стану, який замінює попередній. Усі компоненти, які використовують стан, можуть підписатися на зміни та отримувати оновлені дані.

Redux також надає механізми для поділу стану на модулі та для управління асинхронними операціями. Його API проста і зрозуміла, що спрощує процес розробки. Redux не є обов’язковим для всіх застосунків, але може бути корисним у разі, коли необхідний суворий контроль стану та управління складними станами.

React Hooks – це нова методологія в React, додана у версії 16.8, яка дає змогу використовувати стан та інші можливості React без необхідності створювати класові компоненти. Вона пропонує простіший і елегантніший спосіб керування станом компонентів.

Основна ідея React Hooks полягає в тому, щоб винести логіку стану з компонентів і створювати її окремо, використовуючи спеціальні функції-хуки. Ці функції можна викликати всередині функціональних компонентів, щоб отримати доступ до стану, ефектів, контексту тощо.

За допомогою React Hooks можна використовувати стан у функціональних компонентах, а також використовувати життєвий цикл компонентів, створювати власні хуки для повторного використання логіки тощо. Існує кілька вбудованих хуків у React, таких як useState, useEffect, useContext, useReducer тощо, які надають доступ до різних функціональностей React. Вони можуть бути використані для опрацювання різних завдань, як-от опрацювання подій, отримання та надсилання даних на сервер, керування станом компонента тощо.

React Hooks дає змогу значно скоротити кількість коду, спростити його читання і зробити його більш зрозумілим. Вона також підвищує продуктивність програми, оскільки позбавляє від створення зайвих екземплярів класових компонентів і спрощує процес оновлення компонентів.

Методологія CBFS є популярним підходом до організації структури проєкту в React застосунках. Вона заснована на розбитті застосунка на безліч невеликих, незалежних компонентів.

Суть цієї методології полягає в тому, що кожен компонент розміщується у своїй окремій папці, яка містить усі необхідні файли для цього компонента (JSX, CSS, зображення тощо). Такий підхід допомагає спростити процес розробки, тестування і супроводу коду, оскільки кожен компонент легко ідентифікується і може бути використаний повторно в інших місцях програми. Структура папок може мати такий вигляд (рис. 2.3).

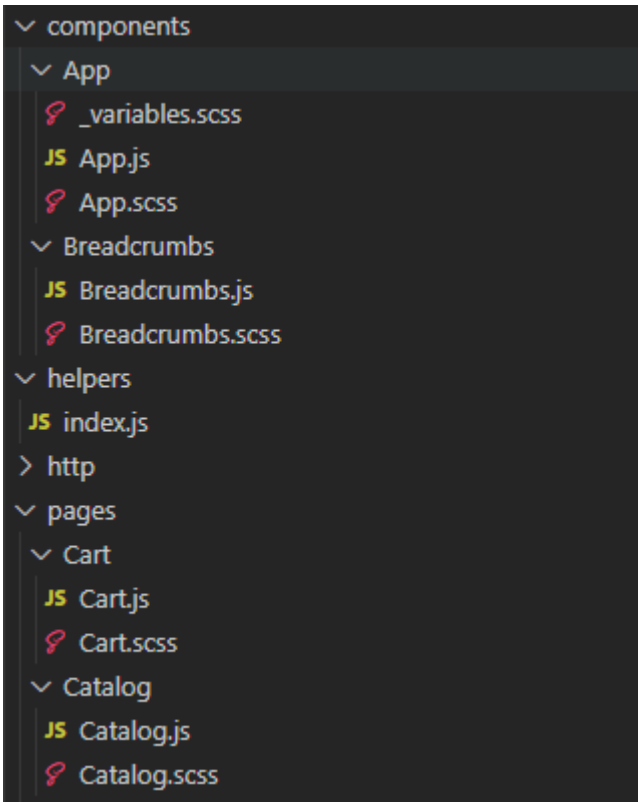


Рисунок 2.3 – Структура папок методології CBFS

У даній структурі папок components і pages відрізняються тим, що компоненти в папці components можуть бути елементи, що перевикористовуються, а компоненти в папці pages використовуються один раз для кожної сторінки.

Також може бути присутня папка helpers, яка містить допоміжні функції та утиліти. Ще може бути папка для API-викликів, яка може мати назву api, services або http. У цій папці можуть знаходитись файли з викликами API для різних моделей, такі як gameApi, userApi тощо. Кожен з цих файлів містить функції для виконання операцій з API, пов'язаних зі відповідною моделлю. Цей підхід дозволяє легко організувати код для викликів API та підвищує його читабельність та підтримку. Використання методології CBFS допомагає створювати більш структурований і легко підтримуваний код у React застосунках.

Кожна з цих методологій має свої переваги та недоліки і може використовуватися залежно від конкретних потреб і вимог проєкту [21, 22].

## 2.3 Методики Back-end частини

Основою Back-end частини ВЗ є платформа, яка дає змогу запускати JavaScript на стороні сервера Node.js, це означає, що розробники можуть використовувати JavaScript для створення повноцінних ВЗ на стороні сервера, не вдаючись до використання інших мов програмування. Node.js використовує рушій V8 JavaScript, який також використовується в браузері Google Chrome, що робить його швидким і ефективним.

На Back-end Node.js може використовуватися для опрацювання запитів клієнтів, створення API, опрацювання даних і виконання багатьох інших завдань. Також за допомогою Node.js можна працювати з базами даних, такими як MongoDB, MySQL, PostgreSQL та іншими. Node.js також забезпечує розширюваність і гнучкість, даючи змогу розробникам створювати і використовувати власні модулі для роботи з конкретними завданнями.

Через свою високу продуктивність, масштабованість і гнучкість Node.js є популярним вибором для створення серверної частини ВЗ. Він також має

велику спільноту розробників, яка створює і підтримує безліч пакетів і бібліотек, що спрощують розробку ВЗ на Node.js.

Ще Одним з основних інструментів для роботи з базами даних на Node.js є ORM (Object-Relational Mapping) – бібліотеки, які дають змогу розробникам використовувати об’єктно-орієнтований підхід до роботи з даними, замість написання SQL-запитів вручну. ORM надають інтерфейс для взаємодії з базою даних, дають змогу легко створювати і змінювати схему бази даних, а також забезпечують безпеку і захист від SQL-ін’єкцій.

Існує безліч ORM для Node.js, таких як Sequelize, TypeORM, Prisma та інші. Вони підтримують роботу з різними базами даних, надають можливість створення зв’язків між таблицями та забезпечують зручний і простий інтерфейс для роботи з даними. Використання ORM дає змогу значно скоротити час і зусилля, що витрачаються на роботу з базами даних, і спростити процес розроблення ВЗ на Node.js.

У Node.js є багато методології, але в Node-спільноті часто використовуються такі методології як MVC, REST, SOLID.

MVC – це популярна методологія, що використовується під час розроблення ВЗ, яка дає змогу розділити застосунок на три основні компоненти: модель (Model), подання (View) і контролер (Controller). У Node.js ця методологія також може бути використана для створення застосунків. Наприклад, при створенні ВЗ на Node.js, модель (Model) може являти собою схеми даних, які використовуються в базі даних, а контролер (Controller) може бути відповідальним за обробку запитів від клієнта та управління «бізнес-логікою» [23–27].

REST – це архітектурний стиль, який визначає набір обмежень і правил для створення вебсервісів, що працюють із ресурсами. RESTful API – це вебсервіс, який відповідає цим обмеженням і правилам. RESTful API використовує HTTP-методи та ресурси для надання доступу до інформації та взаємодії з нею. Крім того, RESTful API використовує формати даних, коди

стану HTTP та інші елементи для забезпечення надійності та ефективності взаємодії.

SOLID – це абревіатура, яка позначає п’ять основних принципів об’єктно-орієнтованого програмування (ООП). Кожна літера в SOLID відповідає одному з цих принципів:

– S – Single Responsibility Principle (принцип єдиної відповідальності).

Означає, що кожен клас або модуль повинен мати тільки одну відповідальність. Тобто клас повинен займатися тільки однією справою і не повинен мати змішаних або нерелевантних функцій. Це покращує підтримку, розширюваність і тестування коду;

– O – Open/Closed Principle (принцип відкритості/закритості).

Означає, що класи та модулі мають бути відкриті для розширення, але закриті для зміни. Це досягається шляхом використання абстракцій та інтерфейсів, які можна розширювати без зміни основного коду;

– L – Liskov Substitution Principle (принцип підстановки Лісків).

Означає, що об’єкти класів-спадкоємців мають бути здатні замінити своїх батьківських класів без порушення роботи програми. Тобто код, який використовує базовий клас, має працювати з будь-яким спадкоємцем цього класу без зміни своєї поведінки;

– I – Interface Segregation Principle (принцип розділення інтерфейсу).

Означає, що інтерфейси мають бути розділені на менші, більш специфічні інтерфейси, щоб клієнти могли використовувати тільки ті методи, які їм необхідні. Це зменшує зв’язність між класами та покращує модульність;

– D – Dependency Inversion Principle (принцип інверсії залежностей).

Означає, що модулі мають залежати від абстракцій, а не від конкретних реалізацій. Тобто залежності мають бути інвертовані. Це зменшує зв’язність між класами та покращує тестування і розширюваність коду.

Кожен із цих принципів є концептуальним правилом, яке допомагає створювати високоякісний код, який легко супроводжувати та модифікувати.

## 2.4 Обґрунтування структури бази даних

База даних – це колекція даних, організованих у такий спосіб, щоб забезпечити легкий та швидкий доступ до них, їх збереження та оновлення. Бази даних знаходять широке застосування у різних галузях, включаючи бізнес, науку, медицину, громадські послуги та багато інших.

Одна з важливих переваг баз даних полягає у тому, що вони дозволяють легко зберігати, організовувати та звертатися до великої кількості даних. Бази даних можуть містити інформацію про клієнтів, продукти, послуги, фінанси, наукові дослідження, медичні записи та інше.

За допомогою баз даних можна швидко та легко виконувати запити до даних, що дозволяє швидко знайти необхідну інформацію. Бази даних також дозволяють забезпечити безпеку даних, контролювати доступ до них та забезпечувати їх цілісність.

Оскільки бази даних є важливою складовою багатьох систем, важливо враховувати потреби користувачів та забезпечити правильну структуру баз даних для забезпечення їх надійності та ефективності. Крім того, необхідно забезпечувати резервне копіювання баз даних для захисту від втрати даних та забезпечувати захист від зломів та зловмисного впливу.

Важливість структури бази даних полягає в тому, що вона визначає, як дані будуть зберігатись та організовуватись у системі. Якщо структура бази даних не буде правильно спроєктована, це може призвести до непередбачуваних наслідків, таких як зниження продуктивності, недостовірність даних, складність в збереженні і оновленні даних, складність в пошуку та зверненні до даних, недоступність даних для роботи з ними.

Правильна структура бази даних дозволяє легко зберігати, організовувати та звертатися до даних, забезпечує їх цілісність і безпеку, дозволяє легко відслідковувати зв’язки між даними і здійснювати запити до бази даних. В результаті, правильно структурована база даних допомагає забезпечити ефективну та надійну роботу системи в цілому.

Під час вибору бази даних для проєкту необхідно враховувати безліч чинників, включно з типом застосунку, кількістю даних, вимогами до продуктивності та багатьма іншими. У цьому контексті PostgreSQL є одним із найпривабливіших варіантів для багатьох застосунків.

По-перше, PostgreSQL – це реляційна база даних, яка володіє високим ступенем стандартизації та широкими можливостями з маніпулювання даними, завдяки мові SQL.

По-друге, PostgreSQL вирізняється високою продуктивністю та стабільністю, що забезпечує швидке виконання запитів і мінімальний час простою програми.

Звісно, під час вибору бази даних необхідно враховувати специфічні вимоги та особливості кожного проєкту. Але загалом PostgreSQL є потужною і гнучкою базою даних, яка підходить для більшості застосунку і має низку переваг перед іншими СУБД.

У базі даних буде представлено декілька моделей користувач (User), гра (Game), покупок (Purchase), жанр (Genre) і категорія (Category), а також зв’язки між ними. На наведеному рисунку показано майбутню структуру бази даних (рис. 2.4).

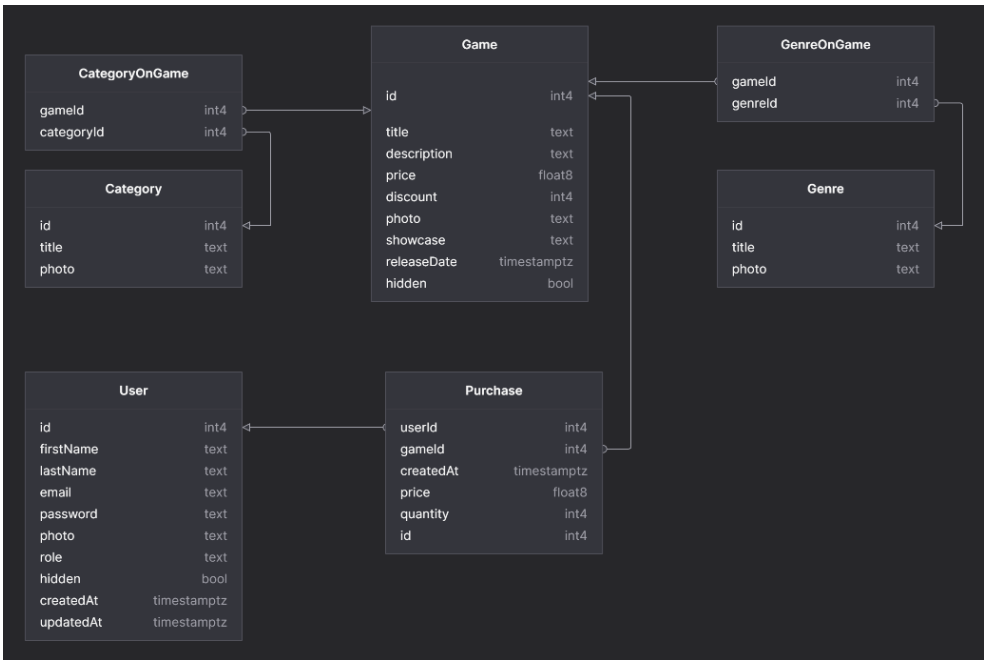


Рисунок 2.4 – Структура бази даних

Модель User містить поля, які необхідні для автентифікації користувача та відстеження його активності на платформі, такі як ім’я, прізвище, адреса електронної пошти, пароль, фотографія профілю та роль на платформі. Поле hidden дозволяє приховувати профіль користувача за необхідності. Модель також містить зв’язок з покупками.

Модель Game містить інформацію про гру, таку як назва, опис, ціна, знижка, фотографії та дата виходу гри. Поле hidden також дає змогу приховати гру за потреби. Модель також містить зв’язки з жанрами, категоріями і покупками.

Моделі Genre і Category містять інформацію про ігрові жанри та категорії відповідно. Вони мають поля назви та фотографії, а також зв’язки з іграми через моделі GenreOnGame і CategoryOnGame.

Модель Purchase відстежує інформацію про купівлю, як-от ідентифікатор користувача, ігри, кількість і ціна, а також зв’язки з користувачами та іграми.

PostgreSQL буде використовуватися як система управління базами даних, оскільки вона надає безліч можливостей для управління даними та підтримки зв’язків між таблицями.

Така структура дозволяє легко розширювати та модифікувати базу даних у майбутньому, наприклад, додавати нові поля до моделей чи відносини між ними, не порушуючи існуючої структури. Кожна модель містить набір полів, які відображають характеристики об’єктів, що зберігаються в цій таблиці. Завдяки такому підходу, база даних стає потужним інструментом для роботи з даними, що дозволяє зберігати, оновлювати та отримувати необхідну інформацію відповідно до потреб користувачів системи.

Загалом, структура цієї бази даних являє собою добре організовану модель, яка забезпечує ефективне зберігання і доступ до даних, необхідних для роботи ІМ [28–30].

# 3 ПРОГРАМНА РЕАЛІЗАЦІЯ

## 3.1 Реалізація контейнеризації

Контейнеризація є одним із найефективніших способом керування ресурсами, безпекою та розгортанням ВЗ. Контейнери Docker дають змогу створювати легковагі та портативні програми, які можна запускати практично в будь-якому оточенні.

Docker використовує концепцію контейнеризації, засновану на принципах ізоляції ресурсів і спільного використання ядра операційної системи сервера. Контейнери створюються на основі образів, які можуть бути створені або завантажені із загальнодоступних ресурсів.

Лістинг 3.1 Front-end Docker файл:

```
FROM node:14-alpine
WORKDIR /app
COPY . .
RUN npm install
RUN npm install -g react-scripts
RUN mv node_modules /node_modules
EXPOSE 3000
CMD ["npm", "start"]
```

Кроки для Front-end Docker-файлу будуть включають наступне:

Крок 1. FROM node:14-alpine – вказує на базовий образ, який буде використовуватися для створення контейнера. У цьому випадку це образ Node.js версії 14, заснований на легковажній версії Linux Alpine.

Крок 2. WORKDIR /app – задає робочу директорію всередині контейнера. У цьому випадку робоча директорія буде /app.

Крок 3. COPY . . – копіює всі файли з поточного каталогу (точка) всередину контейнера в поточну робочу директорію (/app).

Крок 4. RUN npm install – виконує команду встановлення всіх залежностей проєкту з package.json за допомогою менеджера пакетів npm.

Крок 5. RUN npm install -g react-scripts – встановлює пакет react-scripts глобально за допомогою менеджера пакетів npm. Цей пакет містить різні скрипти для роботи з React застосунками.

Крок 6. RUN mv node_modules /node_modules – переміщує папку node_modules в кореневу директорію контейнера. Це зроблено для оптимізації контейнера та зменшення його розміру.

Крок 7. EXPOSE 3000 – вказує на порт, який буде прослуховуватися всередині контейнера. У цьому випадку це порт 3000, на якому запускається React застосунок.

Крок 8. CMD ["npm", "start"] – задає команду, яка запускатиметься під час старту контейнера. У цьому випадку це команда npm start, яка запускає скрипт start з package.json. Це дає змогу запустити React застосунок усередині контейнера.

Лістинг 3.2 Back-end Docker файл:

```
FROM node:14.17.1-alpine
WORKDIR /app
COPY . .
RUN npm install
RUN npm install -g nodemon
RUN mv node_modules /node_modules
EXPOSE 4000
CMD ["npm", "start"]
```

Кроки для Back-end Docker-файлу не дуже відрізняються.

Контейнер для бази даних не потребує окремого Docker-файлу тому що усі налаштування будуть написані у файлі Docker-compose. Docker Compose дозволяє нам визначити всі контейнери, їх залежності та налаштування в одному конфігураційному файлі. Кожен контейнер визначається як окремий сервіс із зазначенням образу Docker, який використовуватиметься для створення контейнера.

Лістинг 3.3 Docker-compose файл:

```
version: '3.8'
services:
    backend:
        build: ./backend
        container_name: backend
        env_file: .env
        volumes:
            - ./backend:/app
        ports:
            - '4000:4000'
  frontend:
    build: ./frontend
    container_name: frontend
    environment:
        - WATCHPACK_POLLING=true
    volumes:
        - ./frontend:/app
    ports:
        - '3000:3000'
postgres:
    image: postgres:13
    container_name: postgres
```

*env_file: .env*

*volumes:*

*- ./postgres/db_data:/var/lib/postgresql/data*

*ports:*

*- '5432:5432'*

Цей файл – це файл Docker-compose, який використовується для створення і запуску декількох пов’язаних контейнерів Docker. Він описує три контейнери: backend, frontend і postgres, які будуть зібрані з відповідних Dockerfile і запущені у зв’язку один з одним.

Ось що відбувається в кожному рядку:

– version: '3.8' – вказує версію синтаксису файлу docker-compose;

– services: – це ключове слово, що означає початок визначення сервісів, які будуть створені;

– backend: – це ім’я сервісу. Воно використовується для ідентифікації контейнера та взаємодії з ним всередині мережі Docker;

– build: ./backend – вказує на шлях до директорії, що містить Dockerfile для складання образу для цього сервісу;

– container_name: backend – задає ім’я контейнера, який буде створено;

– env_file: .env – визначає файл, що містить змінні оточення, які будуть використовуватися в контейнері;

– volumes: – це ключове слово, що вказує список томів, які будуть пов’язані з контейнером;

– ./backend:/app – пов’язує локальну директорію backend з директорією /app усередині контейнера, щоб застосунок міг бути запущений у контейнері;

– ports: – вказує порти, які будуть проксіровані між контейнером і хостом;

– '4000:4000' – вказує, що порт 4000 у контейнері має бути доступний на порту 4000 на хості.

Аналогічно, для сервісів frontend і postgres визначаються відповідні параметри збірки, ім’я контейнера, файли оточення, томи і порти.

## 3.2 Реалізація Front-end частини

У реалізації Front-end частини важливою частиною є файл package.json це файл, який використовується в багатьох проєктах на Front-end розробці. Цей файл містить метадані про проєкт, такі як залежності, скрипти, версії та іншу інформацію.

Крім того, package.json має важливе значення для спільної роботи над проєктом у команді. Файл дає змогу переконатися в тому, що кожен член команди використовує однакові версії бібліотек і залежностей, що знижує ймовірність конфліктів і помилок.

Лістинг 3.4 package.json файл:

```
{
  "name": "frontend",
  "version": "1.0.0",
  "main": "src/index.js",
  "license": "ISC",
  "scripts": {
    "start": "react-scripts start"
  },
  "dependencies": {
    "react": "^18.2.0",
   ...
  },
```

```
  "browserslist": {
    "production": [...],
    "development": [...]
  },
  "devDependencies": {
    "sass": "^1.61.0"
  }
}
```

Основні поля файлу:

– «name» – ім’я проєкту;

– «version» – версія проєкту;

– «main» – головний файл проєкту;

– «license» – тип ліцензії, під якою поширюється проєкт;

– «scripts» – список скриптів, які можуть бути запущені в проєкті;

– «dependencies» – список залежностей проєкту, які потрібні для його роботи;

– «devDependencies» – список залежностей, які потрібні тільки для розробки проєкту;

– «browserslist» – список підтримуваних браузерів та їхніх версій.

У цьому прикладі використовуються бібліотеки та фреймворки, такі як React, React-Router, Redux Toolkit, Bootstrap, Tailwindcss та інші. Також тут присутні пакети для тестування і роботи з API – Axios, JWT-decode.Цей файл також містить список скриптів, які можна запускати в проєкті, наприклад, «start» для запуску проєкту в режимі розробки. Тепер перейдемо до головного файлу Front-end частини index.js

Лістинг 3.5 index.js файл:

```
import { createRoot } from "react-dom/client";
import { Provider } from "react-redux";
```

```
import store from "./store";
import App from "./components/App/App";
createRoot(document.getElementById("root")).render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Цей код відповідає за рендеринг React застосунка в браузері.

Перший рядок імпортує функцію createRoot з react-dom/client, яка використовується для створення кореневого елемента React-застосунку.

Другий рядок імпортує компонент Provider з react-redux, який забезпечує доступ до стану Redux для всіх компонентів у застосунку.

Третій рядок імпортує store з файлу ./store, який містить конфігурацію та налаштування для Redux.

Четвертий рядок визначає компонент App з файлу ./components/App/App, який являє собою основний компонент програми.

Останній рядок викликає функцію render на створеному кореневому елементі, передаючи як аргумент компонент Provider із пропсом store, усередині якого знаходиться компонент App. Таким чином, App стає дочірнім компонентом Provider і отримує доступ до стану Redux. Як наслідок, застосунок починає рендеритися на сторінці в елементі з «id="root"».

Лістинг 3.6 App компонент:

```
<BrowserRouter>
    <Header />
        <AppRouter />
    <Footer />
</BrowserRouter>
```

Перший рядок компонента імпортує компонент BrowserRouter із бібліотеки react-router-dom, який забезпечує клієнтську маршрутизацію на стороні клієнта. Усередині компонента BrowserRouter розташовується три дочірні компоненти:

– компонент Header, який являє собою верхню частину веб-сторінки, наприклад, навігаційне меню, логотип, кнопки тощо;

– компонент AppRouter, який відповідає за маршрутизацію між різними сторінками ВЗ. Він використовує компоненти Route з react-router-dom, щоб відображати відповідний компонент залежно від шляху URL;

– компонент Footer, який являє собою нижню частину сторінки, наприклад, копірайт, посилання на соціальні мережі тощо;

– компонент BrowserRouter обертає ці три компоненти, щоб забезпечити маршрутизацію між сторінками ВЗ. Таким чином, застосунок може динамічно змінювати вміст сторінки залежно від шляху URL, без необхідності перезавантаження всієї сторінки.

Лістинг 3.7 AppRoute компонент:

```
<Routes>
        {publicRoutes.map(({ path, element }) => (
          <Route key={path} path={path} element={element} exact />
        ))}
        {isAuth &&
          authRoutes.map(({ path, element }) => (
            <Route key={path} path={path} element={element} exact />
          ))}
        <Route path="*" element={<Navigate to="/" />} />
</Routes>
```

Цей компонент являє собою маршрутизатор у React застосунку і використовує бібліотеку react-router-dom. Він дає змогу визначати різні маршрути ВЗ і пов’язувати їх із відповідними компонентами.

Компонент Routes приймає на вхід список маршрутів, які задаються у вигляді масиву об’єктів. У цьому масиві маршрутів можуть міститися як публічні маршрути, які доступні всім користувачам, так і маршрути, доступні тільки авторизованим користувачам.

Для кожного маршруту з масиву використовується компонент Route, який має такі властивості:

– path – шлях маршруту у вигляді рядка;

– element – компонент, який буде відображатися при збігу маршруту;

– exact – прапор, що вказує на точний збіг шляху. Якщо він дорівнює true, то маршрут працюватиме тільки для точного збігу шляху, інакше – для всіх шляхів, які починаються із заданого шляху.

Для кожного маршруту зі списку publicRoutes створюється Route, а для кожного маршруту зі списку authRoutes створюється Route, якщо користувач авторизований. Якщо користувач не авторизований, то маршрути зі списку authRoutes не відображатимуться.

Компонент Navigate використовується для переправлення на іншу сторінку в разі, якщо користувач вводить неіснуючий шлях. У цьому випадку, якщо користувач вводить неправильний шлях, то його буде переправлено на головну сторінку програми.

Лістинг 3.8 Routes файл:

```
export const ROUTE = {
  HOME: "/",
  LOGIN: "/login",
  SIGN_IN: "/sign-in",
  PROFILE: "/profile",
  CART: "/cart",
```

```
  PURCHASES: "/purchases",
 ...
};

export const publicRoutes = [
 {
   path: ROUTE.HOME,
   element: <Home />,
 },
 {
   path: ROUTE.LOGIN,
   element: <Login />,
 },
 {
   path: ROUTE.SIGN_IN,
   element: <SignIn />,
 },
 ...
];


export const authRoutes = [
 {
   path: ROUTE.PROFILE,
   element: <Profile />,
 },
 {
   path: ROUTE.CART,
   element: <Cart />,
 },
```

```
 {
  path: ROUTE.PURCHASES,
  element: <Purchases />,
 },
 ...
];
```

Цей компонент містить визначення маршрутів для програми. У цьому компоненті експортуються об’єкти ROUTE, publicRoutes і authRoutes.

Об’єкт ROUTE являє собою константи для кожного маршруту програми:

– publicRoutes – це масив, що містить визначення маршрутів, доступних для неавторизованих користувачів. Кожен об’єкт у масиві представляє маршрут і відповідний йому компонент;

– authRoutes – це масив, що містить визначення маршрутів, доступних тільки для авторизованих користувачів.

Разом ці масиви визначають доступні маршрути в застосунку залежно від того, чи авторизований користувач.

Лістинг 3.9 Home компонент:

```
const Home = () => {
 const [games, setGames] = useState([]);
 const [gamesShowcase, setGamesShowcase] = useState([]);
 const [isLoaded, setIsLoaded] = useState(false);
 useEffect(() => {
  fetchGames()
   .then((data) => {
    setGames(data);
    setGamesShowcase(data.filter((game) => game.showcase));
   })
```

```
 .finally(() => setIsLoaded(true));
  }, []);

  return (
    <Layout isLoaded={isLoaded}>
      <section className="section--showcase">
        <Showcase games={gamesShowcase} />
      </section>
      <section className="section--game-slider">
        <GameSlider games={games} />
      </section>
    </Layout>
  );
};
```

Змінні games, gamesShowcase оголошуються за допомогою хука useState, який використовується для створення станів у React-компонентах.

Games і gamesShowcase являють собою масиви, в які будуть поміщені дані, отримані з функції fetchGames().

Хук useEffect використовується для запуску функції, коли компонент вперше монтується на сторінці. Він містить функцію fetchGames(), яка отримує дані та оновлює стан games. Потім використовується функція setGamesShowcase(), щоб відфільтрувати масив games і створити новий масив gamesShowcase, що містить тільки ігри для вітрини.

Компонент також містить два дочірні компоненти: Showcase і GameSlider, які відображають дані на сторінці. Showcase відображає ігри з масиву gamesShowcase, тоді як GameSlider відображає всі ігри з масиву games.

Таким чином відображається та заповнюється головна сторінка ІМ за допомогою засобів React та Axios.

Головна сторінка буде мати такий вигляд (рис. 3.1).

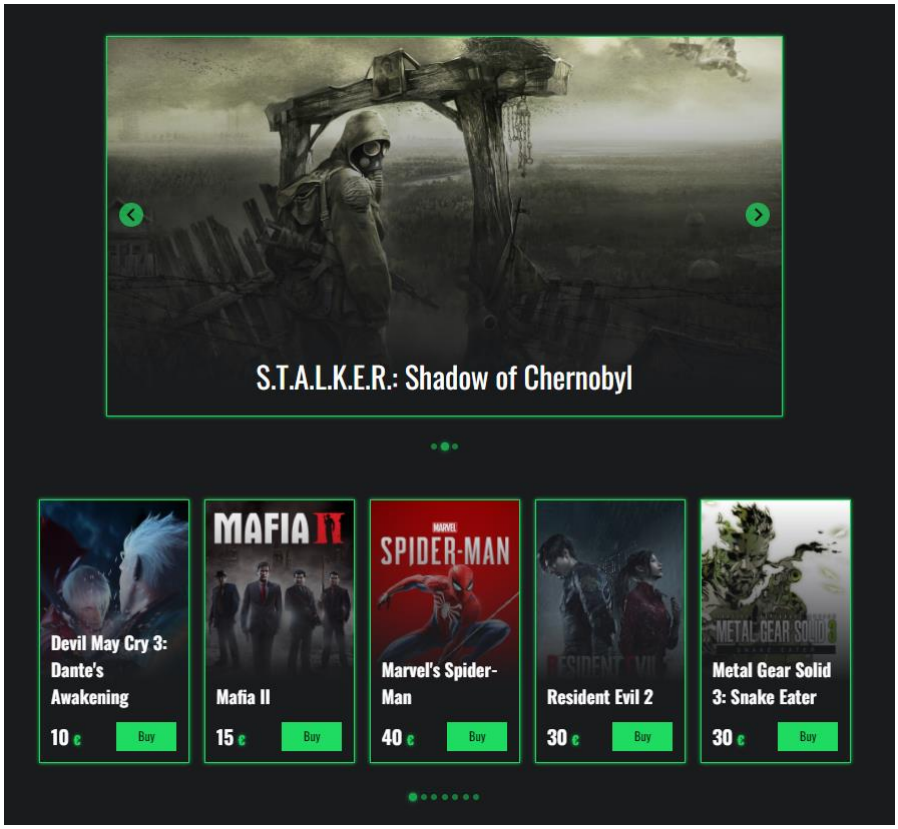


Рисунок 3.1 – Головна сторінка

Лістинг 3.10 Компонент каталогу:

```
const Catalog = () => {
  const [games, setGames] = useState([]);
  const [totalCount, setTotalCount] = useState(0);
  const [activePage, setActivePage] = useState(1);
  const [isLoaded, setIsLoaded] = useState(false);

  useEffect(() => {
    fetchCountOfGames().then((data) => setTotalCount(data));
  }, []);

  useEffect(() => {
```

```
  fetchGamesByPage(activePage)
    .then((data) => {
      setGames(data.games);
      setTotalCount(data.totalCount);
    })
    .finally(() => setIsLoaded(true));
 }, []);

 useEffect(() => {
  fetchGamesByPage(activePage).then((data) => setGames(data.games));
 }, [activePage]);

 return (
   <Layout isLoaded={isLoaded}>
     <section className="section--catalog">
       <h1>Catalog</h1>
       <GameList
         games={games}
         activePage={activePage}
         totalCount={totalCount}
         setPage={setActivePage}
       />
     </section>
   </Layout>
 );
};
```

Змінні games, totalCount і activePage оголошуються за допомогою хука useState, який використовується для створення станів у React-компонентах. Змінна games – це масив ігор, який міститиме дані, отримані з сервера.

Змінна totalCount – це загальна кількість ігор на сервері. Змінна activePage – це поточна сторінка каталогу.

Тут, перший useEffect викликається для отримання загальної кількості ігор на сервері під час монтування компонента на сторінку.

Другий useEffect викликається при зміні activePage, який змінюється користувачем для перемикання сторінок у каталозі. Функція fetchGamesByPage(activePage) викликається для отримання списку ігор, які потрібні для відображення на поточній сторінці.

Третій useEffect також викликається при зміні activePage. Цей хук викликає функцію fetchGamesByPage(activePage), яка отримує список ігор для поточної сторінки та оновлює стан games.

Компонент також містить один дочірній компонент GameList, який відображає список ігор на сторінці відповідно до поточної сторінки activePage і загальної кількості ігор totalCount. Компонент GameList також дозволяє користувачеві перемикатися між сторінками каталогу ігор.

Сторінка каталогу буде мати такий вигляд (рис. 3.2).

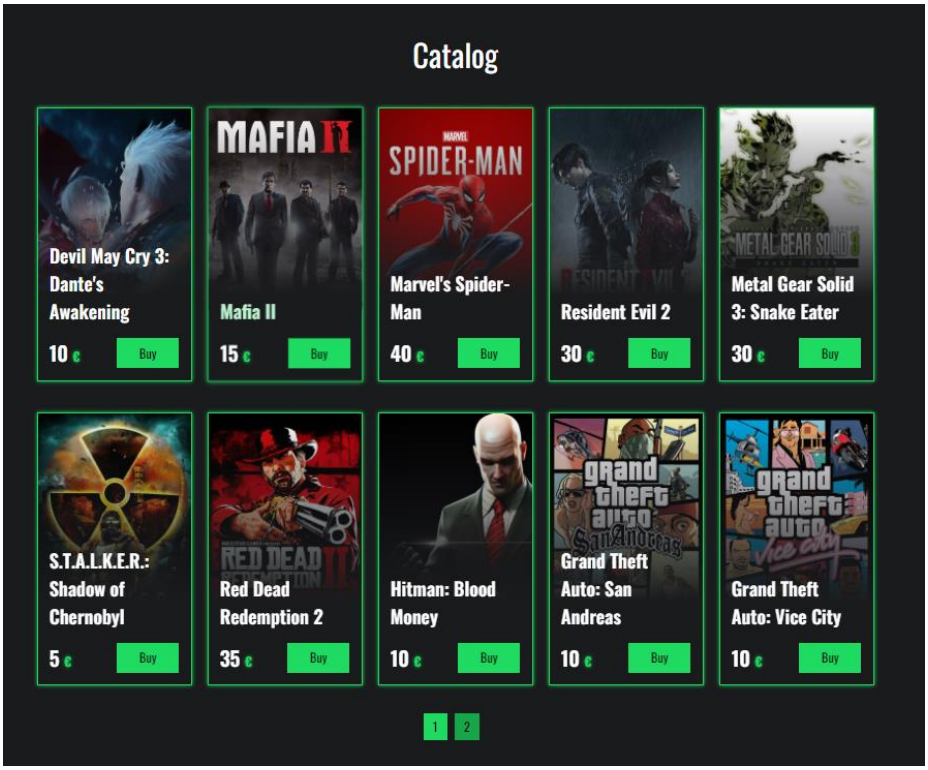


Рисунок 3.2 – Сторінка каталогу

Для сторінки жанри структура компонента аналогічна та використовувати такі ж методи. Сторінка жанрів буде мати такий вигляд (рис. 3.3).

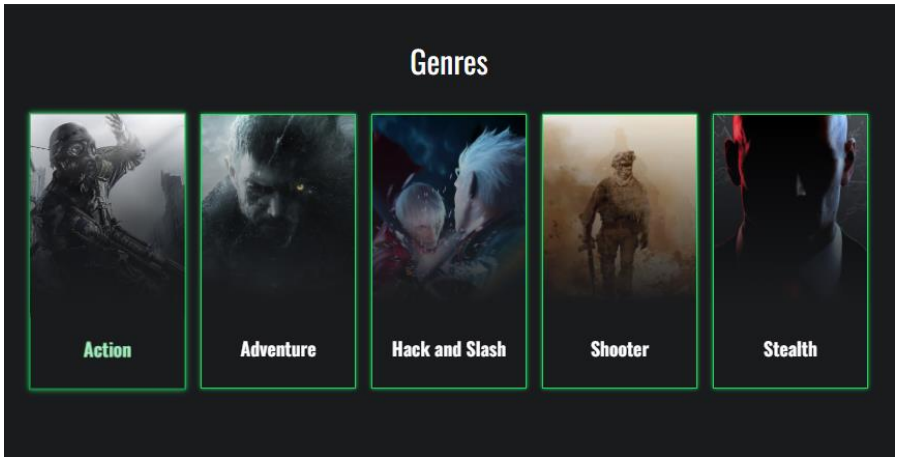


Рисунок 3.3 – Сторінка жанрів

Лістинг 3.11 Логіка компоненту логін:

```
const Catalog = () => {
    const navigate = useNavigate();
    const dispatch = useDispatch();
    const [form, setForm] = useState({});
    const [validationMessages, setValidationMessages] = useState([]);
    const [isLoaded, setIsLoaded] = useState(false);

    useEffect(() => {
      setIsLoaded(true);
    }, []);

    const loginHandle = async () => {
      validate();

      if (!validationMessages.length) {
```

```
    try {
      const { email, password } = form;
      const { id, firstName, lastName, role, photo } = await login(
        email,
        password
      );

      dispatch(setUser({ id, firstName, lastName, email, role }));
      dispatch(setPhoto({ photo }));
      dispatch(setIsAuth(true));
      navigate(ROUTE.PROFILE);
    } catch (e) {
      console.log(e);
    }
  }
};

const handleChange = ({ target }) => {
  setForm({ ...form, [target.name]: target.value });
};

const validate = () => {
  const { email, password } = form;
  let messages = [];

  setValidationMessages(messages);

  if (!email) {
    messages.push("Email is required");
  } else {
```

```
    if (!checkEmail(email)) {
      messages.push("Email is wrong");
    }
  }

  if (!password) {
    messages.push("Password is required");
  } else {

    if (password.length < 4) {
      messages.push("Password is less than 4 characters");
    }

    if (password.length > 20) {
      messages.push("Password is more than 20 characters");
    }
  }

  setValidationMessages(messages);
};
```

Цей компонент представляє форму для входу в систему. Він містить кілька станів, таких як form для зберігання значень полів форми, validationMessages для зберігання повідомлень про помилки валідації.

Коли користувач заповнює поля форми і натискає кнопку входу, викликається функція loginHandle, яка перевіряє валідність введених даних і відправляє запит на сервер для аутентифікації користувача. Якщо аутентифікація пройшла успішно, функція loginHandle оновлює стан застосунку і перенаправляє користувача на сторінку профілю.

Функція handleChange викликається при зміні полів форми і оновлює стан form. Функція validate перевіряє валідність даних, введених користувачем, і встановлює стан validationMessages з повідомленнями про помилки валідації. Сторінка логіну буде мати такий вигляд (рис. 3.4).

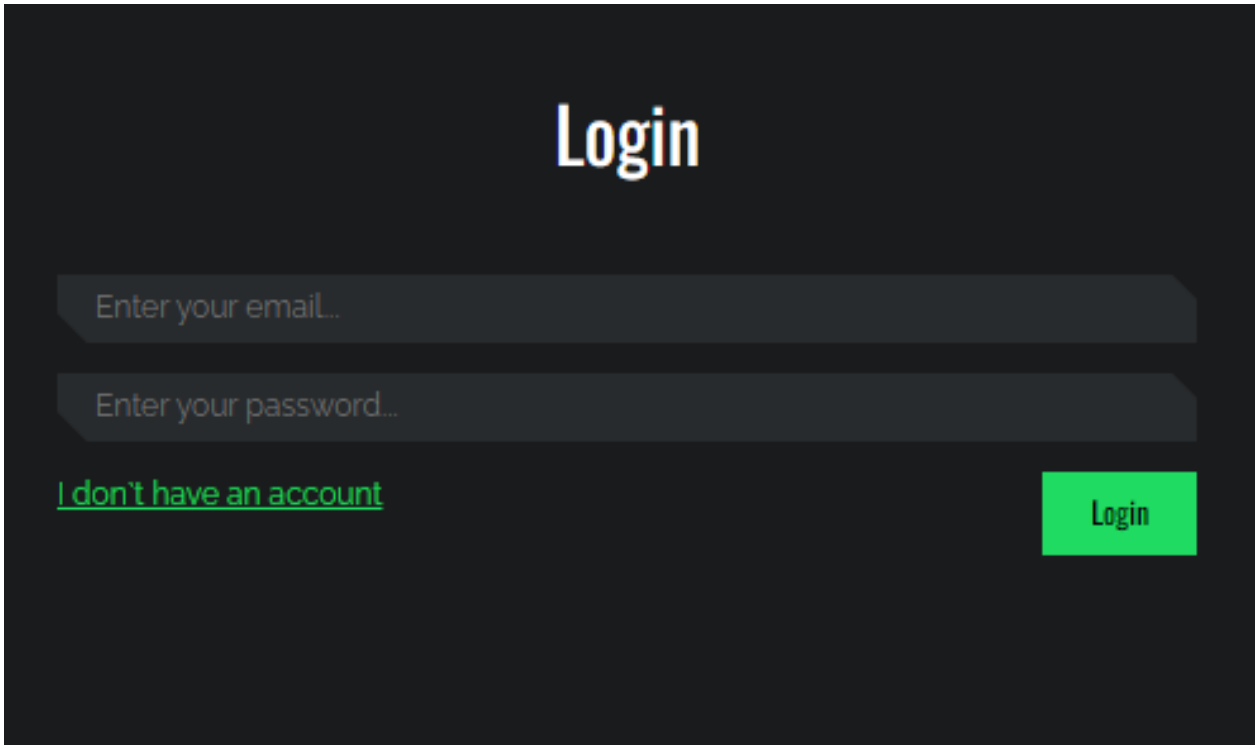


Рисунок 3.4 – Сторінка логіну

## 3.3 Реалізація Back-end частини

У реалізації Back-end частини теж важливою частиною є файл package.json це файл, який використовується в багатьох проєктах на node.js.

У Back-end версії package.json використовуються бібліотеки та фреймворки, такі як Prisma, Express, Bcrypt для хешування паролів, Cors для опрацювання CORS-запитів, Jsonwebtoken для створення і перевірки токенів автентифікації та інше. Цей файл також містить список скриптів, які можна запускати в проєкті, наприклад, «start» для запуску проєкту в режимі розробки. Тепер перейдемо до головного файлу Front-end частини index.js.

У секції scripts визначено команди, які можна використовувати для запуску та розробки програми. Команда start використовує пакет nodemon для автоматичного перезавантаження застосунку в разі змін у коді. Команди migrate і seed використовують пакет prisma для міграції бази даних і заповнення даних тестовими даними відповідно. Команда build комбінує команди migrate і seed для одночасної міграції та заповнення даних.

У розділі prisma вказується команда для запуску скрипта seed.js, який запускається під час виконання команди npm run seed. Це дає змогу завантажити початкові дані в базу даних, які можуть бути необхідними для розроблення та тестування програми.

Загалом це вся різниця між версію package.json для Front-end. Тепер перейдемо до головного файлу Back-end частини app.js.

Лістинг 3.12 app.js файл:

```
require('dotenv').config()
const express = require('express');
const fileUpload = require('express-fileupload')
const router = require('./routes/index')
const cors = require('cors')
const path = require('path')
const errorHandler = require('./middlewares/errorHandlingMiddleware')

const PORT = process.env.PORT || 4000;

const app = express()
app.use(cors())
app.use(express.json())
app.use(fileUpload({}))
app.use('/api', router)
app.use('/img', express.static(path.resolve(__dirname, 'static/img')))
```

```
app.use(errorHandler)


const start = async () => {
    try {
        app.listen(PORT, () => console.log(`Server running on port ${PORT},
        http://localhost:${PORT}`))
    } catch (e) {
        console.log(e)
    }
}
start()
```

Цей код запускає сервер застосунка на зазначеному порту (за замовчуванням 4000), який обробляє запити клієнтів.

На початку файлу підключаються всі необхідні модулі та бібліотеки, як-от Express, Express-fileupload, CORS, path і middleware для обробки помилок.

Потім налаштовується екземпляр Express, де використовуються middleware для парсингу JSON-запитів і обробки файлів, а також CORS для обробки запитів з інших доменів.

Далі, підключається основний роутер, який визначено у файлі ./routes/index.js.

Також додається middleware для обслуговування статичних файлів (зображень), які розташовані в папці static/img.

І, нарешті, застосунок запускається на зазначеному порту, а якщо щось піде не так, помилки будуть оброблені відповідним middleware.

Розберемо фрагмент файлу міграції який буде створювати таблиці та зв’язки за допомогою ORM Prisma.

Лістинг 3.13 Фрагмент файлу міграції:

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Game {
  id           Int           @id @default(autoincrement())
  title        String
  description  String
  price        Float
  discount     Int?
  photo        String
  showcase     String?
  releaseDate  DateTime      @default(now()) @db.Timestamptz(2)
  hidden       Boolean       @default(false)
  genre        GenreOnGame[]
  category     CategoryOnGame[]
  purchase     Purchase[]
}

model Genre {
  id    Int          @id @default(autoincrement())
  title String
  photo String
  game  GenreOnGame[]
}
```

Перший блок визначає джерело даних (data source), у цьому випадку базу даних PostgreSQL, вказуючи параметри підключення через змінну оточення DATABASE_URL. Далі визначено дві моделі: Game і Genre.

Модель Game містить такі поля:

– id: унікальний ідентифікатор гри;
– title: назва гри;
– description: опис гри;
– price: ціна гри;
– discount: знижка на гру (необов’язкове поле);
– photo: ім’я файлу фотографії гри;
– showcase: ім’я файлу фотографії вітрини (необов’язкове поле);
– releaseDate: дата випуску гри;
– hidden: прапор приховування гри (необов’язкове поле);
– genre: масив відношень багато-до-багатьох із моделлю Genre;
– category: масив відносин багато-до-багатьох із моделлю Category;
– purchase: масив відносин багато-до-багатьох з моделлю Purchase.

Модель Genre містить такі поля:

– id: унікальний ідентифікатор жанру;
– title: назва жанру;
– photo: ім’я файлу фотографії жанру;
– game: масив відношень багато-до-багатьох із моделлю Game.

Розберемо фрагмент файлу заповнення бази даних за допомогою ORM Prisma.

Лістинг 3.14 Фрагмент файлу заповнення бази даних:

```
async function createGames() {
  try {
    const games = await prisma.game.createMany({
      data: data.games,
      skipDuplicates: true,
    });
```

```
    let count = games?.count || 0;
    console.log("Creted games:", count);
  } catch (error) {
    let errorMessage = error?.message || "Undefined error";
    console.error("failed to create games:", errorMessage);
  }
}
```

Це функція createGames з використанням async/await синтаксису, яка створює нові записи в таблиці Game в базі даних з використанням Prisma ORM.

Функція використовує метод prisma.game.createMany для створення записів у базі даних. Вона приймає об’єкт data як параметр, що містить дані про нові ігри, які потрібно створити, і параметр skipDuplicates, який вказує, чи потрібно пропустити створення запису, якщо він вже існує в базі даних.

Якщо записи були успішно створені, функція виводить кількість створених записів у консоль. В іншому випадку, якщо сталася помилка, функція виводить повідомлення про помилку в консоль.

Тепер переглянемо файл index.js у папці routes імпортуємо його у головному файлі для того щоб усі кінцеві точки були доступні за адресою /api

Лістинг 3.15 Фрагмент index.js файлу:

```
router.use("/user", userRouter);
router.use("/game", gameRouter);
router.use("/category", categoryRouter);
router.use("/genre", genreRouter);
router.use("/purchase", authMiddleware, purchaseRouter);
```

Цей код оголошує об’єкт роутера і підключає до нього кілька інших роутерів (userRouter, gameRouter, categoryRouter, genreRouter і purchaseRouter) для обробки запитів на різні кінцеві точки ці назви конкатенуються з перефіксом /api.

Зокрема, роутер /purchase використовує middleware authMiddleware, який перевіряє наявність авторизації користувача перед обробкою запиту. Таким чином, цей роутер доступний тільки для авторизованих користувачів.

Лістинг 3.16 Фрагмент gameRouter.js файлу:

```
const Router = require("express");
const router = new Router();
const gameController = require("../controllers/gameController");

router.post("/", gameController.create);
router.get("/", gameController.getAll);
router.get("/search/:value", gameController.search);
router.get("/:id", gameController.getOne);

module.exports = router;
```

Цей код створює екземпляр маршрутизатора Express, який відповідає за обробку запитів, пов’язаних з іграми.

Потім він створює маршрути за допомогою методів HTTP: POST, GET – для створення, отримання всіх, пошуку та отримання однієї гри відповідно. Кожен маршрут обробляється функцією-контролером з модуля gameController.

Нарешті, експортується об’єкт маршрутизатора, який може бути під’єднаний до основного застосунка Express.

Лістинг 3.17 Фрагмент gameController.js файлу:

```
async create(req, res, next) {
  try {
    const { title, description } = req.body;
    const price = parseFloat(req.body?.price) || 0;
    const releaseDate = new Date(req.body?.releaseDate) || new Date();
    const { photo } = req.files;
    const photoName = uuid.v4() + ".jpg";
    photo.mv(path.resolve(__dirname, "../static/img", photoName));

    const game = await Game.create(
      title,
      description,
      price,
      photoName,
      releaseDate
    );

    return res.json(game);
  } catch (e) {
    next(ApiError.badRequest(e.message));
  }
}

async getOne(req, res, next) {
  try {
    const id = parseFloat(req.params.id);
    const game = await Game.getOne(id);

    return res.json(game);
  } catch (e) {
```

```
    next(ApiError.badRequest(e.message));
  }
 }
```

Функція create відповідає за створення нової гри. Із запиту витягуються дані про назву, опис, ціну, дату випуску та зображення гри. Зображення зберігається на диск з унікальним ім’ям і розширенням «.jpg». Потім використовується метод create моделі Game для створення нової гри в базі даних. Якщо операція завершується успішно, функція повертає створену гру у вигляді JSON-об’єкта. Якщо виникає помилка, функція передає управління наступній функції-обробнику помилок, який повертає HTTP-відповідь із кодом 400 Bad Request.

Функція getOne відповідає за отримання інформації про одну гру за її ідентифікатором. З параметрів запиту витягується ідентифікатор гри, і потім використовується метод getOne моделі Game для пошуку гри в базі даних. Якщо гру знайдено, функція повертає її у вигляді JSON-об’єкта. Якщо виникає помилка, функція передає управління наступній функції-обробнику помилок, який повертає HTTP-відповідь із кодом 400 Bad Request.

Лістинг 3.18 Фрагмент моделі game.js файлу:

```
 async create(title, description, price, photo, releaseDate) {
  try {
   const game = await prisma.game.create({
    data: {
     title,
     description,
     price,
     photo,
     releaseDate,
    },
```

```
    });
    return game;
  } catch (error) {
    let errorMessage = error?.message || "Undefined error";
    return errorMessage;
  }
}

async getOne(id) {
  const game = await prisma.game.findUnique({ where: { id } });
  return game;
}
```

Тут визначено два асинхронні методи для роботи з моделлю Game.

Метод create приймає параметри title, description, price, photo і releaseDate, створює новий запис у базі даних із зазначеними значеннями і повертає створену гру в разі успіху. Якщо відбувається помилка, метод повертає відповідне повідомлення про помилку.

Метод getOne отримує запис в таблиці game бази даних і повертає їх у вигляді об’єкту гри. Тим самим будо розібрано весь цикл кінцевої точки /api/game/:id.

Лістинг 3.19 Відповідь кінцевої точки /api/game/:id:


```
{
    "id": 1,
    "title": "Devil May Cry 3: Dante's Awakening",
    "description": "Devil May Cry 3: Dante's Awakening is a 2005 action-adventure game developed and published by Capcom. \n The game is a prequel to the original Devil May Cry, featuring a younger Dante. Set a decade before the events \n of the first Devil May Cry in an enchanted tower,
```

Temen-ni-gru, the story centers on the dysfunctional relationship \n between Dante and his brother Vergil. The game introduces combat mechanics with an emphasis on combos and fast-paced \n action. The story is told primarily in cutscenes using the game's engine, with several pre-rendered full motion videos.",
"price": 10,
"discount": null,
"photo": "dmc_3.jpg",
"showcase": "dmc_3.jpg",
"releaseDate": "2005-02-17T00:00:00.000Z",
"hidden": false
}

# ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений і реалізований інтернет-магазин для продажу ігор.

Застосунок успішно здійснює усі базові функції інтернет-магазину, такі як перегляд каталогу товарів, додавання товарів до кошика, перегляд покупок, пошук товарів, перегляд товарів, реалізація категорій, авторизація та реєстрація. Для зручності користувачів було розроблено зручний та інтуїтивно зрозумілий інтерфейс, який дозволяє швидко та зручно здійснювати пошук та покупки товарів.

У ході роботи були використані React і Node.js це дозволило реалізувати зручний та масштабований інтерфейс користувача та ефективну обробку даних на серверній стороні. Також, у розробці застосунку були використані сучасні технології та фреймворки, що дозволило забезпечити високу швидкість роботи та підвищити безпеку даних користувачів.

Застосунок має можливість масштабування та розширення, що дозволяє додавати новий функціонал та збільшувати обсяг продажів без втрати продуктивності та швидкості роботи.

Отже, розроблений інтернет-магазин є функціональним та надійним інструментом для онлайн продажу ігор, який задовольняє всі необхідні потреби користувачів (рис. А.1–А.9, рис. Б.1–Б.6).

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Popularity: Angular vs React vs Vue. URL: https://www.reddit.com/r/angular/comments/ik02qv/popularity_angular_vs_react_vs_vue (дата звернення 12.05.2023).

2. Яку мову програмування обрати початківцю. Поради досвідчених розробників. URL: https://dou.ua/lenta/articles/choosing-programming-language-for-junior/ (дата звернення 12.05.2023).

3. Ситніков, Д. Е., & Тітова, О. В. (2013). Аналіз веб-сайтів органів місцевої влади як механізму забезпечення права доступу до публічної інформації. Вісник Харківської державної академії культури, (41), 134-142.

4. Творошенко, І.С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. Харків: ХНУРЕ.

5. Тітов, С. В., & Тітова, О. В. (2015). Оцінка юзабіліті освітніх сайтів: методи і технології. Вісник Харківської державної академії культури. Серія: Соціальні комунікації, (47), 127-134.

6. Iryna, T., & Heorhii, M. (2021). Research of regression and modular testing of web applications. Editorial Board, 406.

7. Tvoroshenko, I., & Andrieieva, A. (2021). Development of web applications for remote learning of English.

8. Tvoroshenko, I. S., & Maksimenko, H. (2021). To the question of analysis of existing mechanisms of web application testing.

9. Kuzomin, O., Tolmachova, T., & Astappiev, O. (2017). Analysis of Web user activity data. International Journal of Information Models and Analyses, 6(2), 108-118.

10. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, *Сучасні інформаційні системи*, 6(3), С. 5-12.

11. Gorokhovatskyi V., Tvoroshenko I., Kobylin O., and Vlasenko N. (2023) Search for visual objects by request in the form of a cluster representation for the structural image description, *Advances in Electrical and Electronic Engineering*, 21(1), pp. 19-27.

12. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Tools for fast metric data search in structural methods for image classification, *IEEE Access*, 10, pp. 124738-124746.

13. Osmani, A. (2023). Learning JavaScript design patterns. " O'Reilly Media, Inc.".

14. Rawat, P., & Mahajan, A. N. (2020). ReactJS: A modern web development framework. International Journal of Innovative Science and Research Technology, 5(11), 698-702.

15. Saks, E. (2019). JavaScript Frameworks: Angular vs React vs Vue.

16. Miell, I., & Sayers, A. (2019). Docker in practice. Simon and Schuster.

17. Potdar, A. M., Narayan, D. G., Kengond, S., & Mulla, M. M. (2020). Performance evaluation of docker container and virtual machine. Procedia Computer Science, 171, 1419-1428.

18. Ranjan, A., Sinha, A., & Battewad, R. (2020). JavaScript for Modern Web Development: Building a Web Application Using HTML, CSS, and JavaScript. BPB Publications.

19. Makris, A., Tserpes, K., Spiliopoulos, G., Zissis, D., & Anagnostopoulos, D. (2021). MongoDB Vs PostgreSQL: A comparative study on performance aspects. GeoInformatica, 25, 243-268.

20. Жулкевський, В. (2020). Використання архітектурного підходу Flux для побудови веб-застосунків на прикладі бібліотеки Redux.

21. Hoque, S. (2020). Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node. js. Packt Publishing Ltd.

22. Mai, N. (2020). E-commerce Application using MERN stack.

23. Гороховатський, В. О., & Творошенко, І. С. (2022). Аналіз багатовимірних даних за описом у формі множини компонент.

24. Гороховатський В., Передрій О., Творошенко І., Марков Т. (2023) Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень, *Сучасні інформаційні системи*, 7(1), С. 5-13.

25. Kinoshenko, D., Mashtalir, V., Yegorova, E., & Vinarsky, V. (2005). Hierarchical partitions for content image retrieval from large-scale database. In Machine Learning and Data Mining in Pattern Recognition: 4th International Conference, MLDM 2005, Leipzig, Germany, July 9-11, 2005. Proceedings 4 (pp. 445-455). Springer Berlin Heidelberg.

26. Єльчанінов, Д. Б., Косіло, М. С., & Бєлова, Н. В. (2014). Технології автоматизації проектування програмних систем. Системи обробки інформації, (6), 135-140.

27. Iryna, T., & Maksym, K. (2021). Research results of functional, white box and smoke testing methods for mobile applications. Trends in science and practice of today, 5, 418.

28. Mohammadi, R. Performance, and Efficiency based Comparison of Angular and React in a case study of Single page application (SPA) (Doctoral dissertation, Dissertation 2020, Herat University, https://scholar. google. com/scholar).

29. Nguyen, H. (2021). Front end architecture for a single page web application.

30. Kornienko, D. V., Mishina, S. V., & Melnikov, M. O. (2021, November). The Single Page Application architecture when developing secure Web services. In Journal of Physics: Conference Series (Vol. 2091, No. 1, p. 012065). IOP Publishing.