МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії програмного забезпечення, канд. техн. наук, доцент

_____ Є. О. Давиденко

«___»_____2022р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ МОНІТОРИНГУ НАЯВНОСТІ ТОВАРУ НА МАРКЕТПЛЕЙСАХ

Спеціальність «Інженерія програмного забезпечення»

121 – КРБ.1 – 408.21920802

Студент _____ А. О. Соловйов

«__»_____ 2022р.

Керівник викладач _____ І. О. Кандиба

«__»_____ 2022р.

Консультант канд. техн. наук, доцент _____ А. О. Алексєєва

«__»_____ 2022р.

Миколаїв – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ІС – Інформаційна система

ПЗ – Програмне забезпечення

СКБД – Система керування базою даних

API – Application Programming Interface

UML – Unified Modeling Language

GUI – Graphical User Interface

SASS – Syntactically Awesome Style Sheets

LESS – Leaner Style Sheets

ORM – Object-relational mapping

SPA – Single Page Application

DOCR – Digital Ocean Container Registry

CSRF – Cross-Site Request Forgery

HTTP – HyperText Transfer Protocol

REST – Representational State Transfer

SEO – Search Engine Optimization

ВСТУП

Актуальність теми даної кваліфікаційної роботи пов’язана зі складністю замовлення деяких товарів на ринку Північної Америки та Європи для звичайних покупців. Придбання таких товарів, як PlayStation 5, Xbox Series X та S, Nintendo Switch, нові відеокарти Nvidia та багато інших виявилося випробуванням для споживачів. Така ситуація склалася з низки причин. Розглянемо проблему на прикладі відомої у всьому світі консолі PlayStation 5.

Понад 6 мільйонів споживачів придбали PlayStation 5 з моменту запуску ігрової консолі Sony наступного покоління в листопаді. Ще незліченні мільйони спробували і зазнали невдачі. Це тому, що з ряду причин, які не залежать від звичайного споживача, покупка PS5 є більш складною, ніж будь-якої з ігор, в які вони можуть грати на ній.

Саме існування консолі вартістю 500 доларів (400 доларів без дисковода) разом із Xbox наступного покоління X і S від Microsoft є лише чуткою для більшості споживачів — шепіт вночі. Вони часто залишаються днями чи тижнями, не будучи в наявності в інтернеті чи на полицях магазинів. Потім раптом вони з’являться на Amazon, Target, BestBuy або Walmart. Але якщо покупець перебуває на дзвінку або виконує доручення, коли акція починає працювати, у нього практично немає шансів отримати товар. Вони розпродаються за лічені хвилини.

На веб-сайті Sony, у тих рідкісних випадках, коли він продає PS5, використовується система віртуальної черги, яка стала сумно відомою тим, що виставляє споживачів у годинні віртуальні черги лише для того, щоб їм повідомили, вибачте, що консолі давно розпродані. Існує небагато доказів того, що трюки, поширені в Інтернеті, можуть збільшити шанси придбати консоль: відкривати сайти в кількох браузерах на різних пристроях, зберігати всю інформацію про замовлення, постійно оновлювати сторінку, мати блискавичні пальці. З’явилися деякі облікові записи Twitter, які повністю

присвячені сповіщенню підписників, коли консолі є в наявності десь в інтернеті. Навіть використовуючи всі ці методи, придбати PS5 надзвичайно важко. Це неможливо зробити для більшості випадкових споживачів.

Причин для цього багато, і вони охоплюють все: від пандемії Covid-19 до галузі судноплавства. Але перший і найбільш значущий елемент – це глобальний дефіцит напівпровідників, який триває з початку пандемії 2019 року.

Напівпровідники, які часто називають просто чіпами, є «мозком електронних пристроїв» і присутні у всьому, від смартфонів, автомобілів і мікрохвильових печей до підключених іграшок, ігрових консолей і пральних машин.

Попит на чіпи різко зріс протягом 2020 року, оскільки люди та підприємства поспішили купувати нові комп’ютери, ноутбуки, планшети, телефони та інші пристрої, щоб працювати, навчатися та дистанційно адмініструвати охорону здоров’я під час карантину під час пандемії.

Це посилюється дефіцитом кремнію, міді та цинку, що призвело до зростання цін.

У той же час виробничі фабрики, якими користуються компанії, включаючи Apple, були змушені закрити свої двері та призупинити виробництво пристроїв через порушення ланцюга поставок.

Об'єктом роботи є процес відстеження та замовлення товарів у інтернет магазинах.

Предметом роботи є технології скрапінгу та програмних інтерфейсів при розробці програмного забезпечення виявлення наявності актуальних товарів у онлайн магазинах.

Метою кваліфікаційної роботи є автоматизація процесу відстеження та замовлення актуальних товарів на ринку шляхом розробки програмного забезпечення моніторингу товарів на маркетплейсах.

Досягнення мети можливо за допомогою вирішення наступних завдань:

1. провести аналіз аналогів, виявити їх недоліки та переваги;
2. визначити цільові товари на ринку та інтернет магазини у яких їх можливо придбати;
3. дослідити методики стеження за товарами у магазинах;
4. визначити технології розробки, які задовольняють вимогам майбутнього сервісу;
5. розробити сучасний та зручний дизайн інтерфейсу сервісу;
6. спроектувати архітектуру застосунку і бази даних;
7. виконати кодування клієнтської та серверної частини застосунку;
8. провести тестування розробленого ПЗ;
9. розгорнути застосунок на сервері.

1 АНАЛІЗ ОБЛАСТІ СПОСТЕРЕЖЕННЯ ЗА ТОВАРАМИ

1.1 Опис предметної області

Відповідно до об’єкту роботи розглянемо процес відстеження та замовлення товарів у інтернет магазинах.

Результатом даного процесу є успішна покупка товару користувачем у онлайн магазині. Проблема полягає у тому, що придбання деяких товарів стає досить складним завданням. Зазвичай, їх просто не має в наявності, а після поповнення складу вони зникають протягом декількох хвилин. У таких умовах звичайному покупцю досить складно отримати бажаний товар.

Така ситуація спостерігається у більшості відомих магазинах, таких як Amazon, Walmart, Target, BestBuy та інших. Для подолання цієї проблеми було вирішено розробити програмне забезпечення, яке може відслідковувати наявність товарів на популярних маркетплейсах і одразу сповіщати покупців про те, що товар з’явився на полицях магазину. Для досягнення поставленої мети необхідно реалізувати наступні аспекти проекту:

– **Стеження за даними.** Необхідно дослідити методи, за допомогою яких можна отримувати актуальні дані про наявність та ціну в магазинах.

– **Відправка повідомлень.** Визначити шляхи, за допомогою яких можна швидко і ефективно доставляти інформацію про наявність товару користувачам.

– **Аналіз аналогів.** Визначити переваги та недоліки існуючих систем.

– **Сформулювати вимоги.** Відповідно до аналізу аналогічних систем визначитись з вимогами, які будуть забезпечувати розробку системи, яка буде задовольняти потреби бізнесу та користувачів.

– **Розробка веб-застосунку.** Виявити оптимальні технології та архітектуру ПЗ.

1.2 Методи та засоби стеження і збирання інформації

Одним із способів збору даних на веб-ресурсах є використання офіційного API, яке створено розробниками, у тому числі, для подібних цілей. Так як відомі компанії дбають про свою репутацію і якість програмного забезпечення такий підхід має низку переваг:

– **Актуальна інформація**. Один з найважливіших параметрів, адже успішність розроблювальної системи напряму залежить від того, наскільки вчасно буде надходити користувачу повідомлення про надходження товару на склад магазину.

– **Простота реалізації.** Використовувати API дуже просто. Для цього необхідно лише зробити запит за вказаним у документації маршрутом, що забезпечує швидкість розробки продукту. Крім того великі магазини, такі як Amazon, BestBuy, Target та інші мають детальну та добре структуровану документацію, що також полегшує життя розробнику.

– **Швидкість роботи.** Отже використання запитів до програмного інтерфейсу надає можливість достатньо швидко отримувати необхідні дані. Окрім того, інформація вже підготовлена і структурована, що вберігає від написання складного парсингу, який може вповільнити роботу системи.

– **Офіційне джерело.** Останній, але не менш важливий критерій, завдяки якому, ми маємо можливість отримати партнерську програму, а також звертатися до служби підтримки, у разі виникнення питань по роботі з API.

Не зважаючи на те, що такий підхід є дуже вдалим, є перелік проблем, через які не вдасться використати його у всіх випадках при розробці даного сервісу:

– **Відсутність API.** На жаль, не усі з ресурсів, які за вимогами замовника необхідно відстежувати забезпечують даною технологією.

– **Обмеження на частоту запитів.** Зазвичай, розробники подібних систем виставляють обмеження по кількості запитів на одиницю часу задля

контролю навантаження на сервер. Іноді, ця цифра може бути занадто висока для задоволення потребам бізнесу, для якого розроблюється програмне забезпечення у даній роботі.

У випадку, коли не вдається використати API, на допомогу приходить скрапінг.

Веб-скрапінг – це автоматичний метод отримання великої кількості даних із веб-сайтів. Більшість цих даних є неструктурованою інформацією у форматі HTML, які потім перетворюються на впорядковані дані в електронних таблицях або базі даних, щоб їх можна було використовувати в різних застосунках.

Веб-скрапінг вимагає двох частин, а саме краулера та скрапера.

Краулер – це алгоритм штучного інтелекту, який переглядає веб-сторінки, щоб шукати потрібні дані, перейшовши за посиланнями в Інтернеті.

Скрапер, з іншого боку, є специфічним інструментом, створеним для вилучення даних з веб-сайту. Конструкція скрапера може сильно відрізнятися залежно від складності та обсягу проекту, щоб він міг швидко та точно витягувати дані.

Використання скрапінгу забезпечує можливість отримувати необхідні дані без використання API. Крім того, якісний скрапер буде отримувати інформацію ідентичну до тої, яку користувач буде бачити на сторінці інтернет магазину.

Але такий підхід має і недоліки. Наприклад, сайти можуть мати систему безпеки, яка буде блокувати процес збирання інформації, що призводить, до необхідності використовувати сторонні proxy сервіси, які зазвичай є платними. Також, у деяких випадках доведеться використовувати автоматизовані системи управління браузером, що значно понизить швидкість роботи системи.

1.3 Аналіз аналогічних застосунків

Виходячи з мети та завдань роботи було проаналізовано аналогічні застосунки, а також існуючі рішення та веб-технології.

Основним аналогом подібної системи є HotStock. Сервіс надає можливість користувачам легко оформити підписку на товари. Після чого вони будуть отримувати браузерні та імейл повідомлення про надходження товару на склад магазину. Крім того, за бажанням, користувач може обрати лише той тип повідомлення який влаштовує саме його у персональному кабінеті.

На сайті існує система пошуку товарів, каталог розподілений на дві секції – популярні та останні додані товари. Також, на головній сторінці представлений опис принципу роботи сервісу та список відстежуваних магазинів (рис. 1.1).

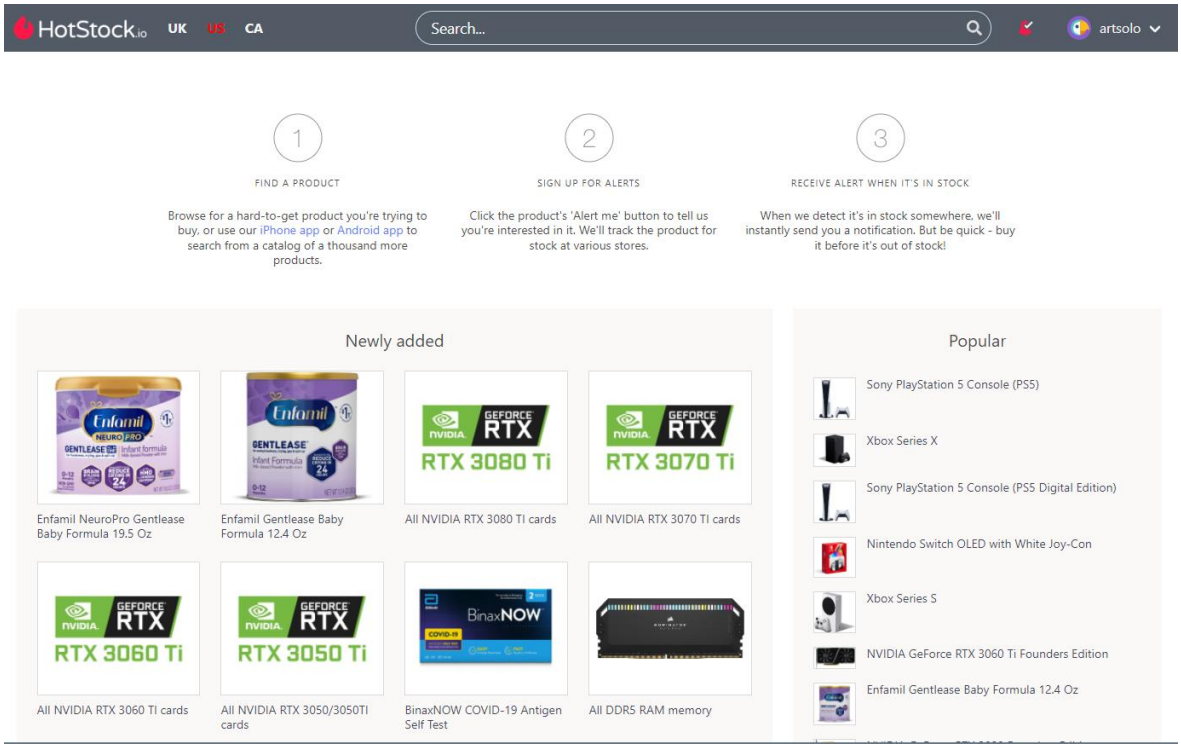

Рисунок 1.1 – Інтерфейс HotStock

На сайті реалізована система авторизації та аутентифікації користувачів. Наявний персональний кабінет у якому можна редагувати свої

дані, вибрати тип повідомлень та відмовитись від усіх підписок, які були зроблені до цього (рис. 1.2).

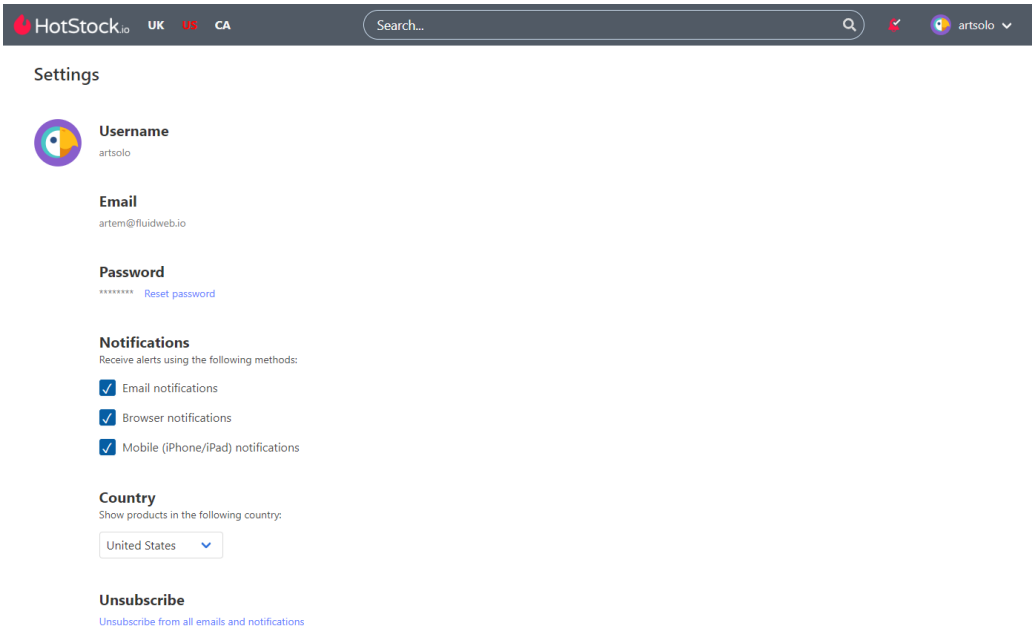

Рисунок 1.2 – Персональний кабінет користувача

На сторінці товару відображається інформація про його наявність на маркетплейсах, ціну та назву, яка представлена у магазинах (рис. 1.3). З цієї сторінки можна оформити підписку на конкретний товар, а також, залишити коментар або переглянути коментарі інших користувачів стосовно даного товару. Уся інформація оновлюється у реальному часі.

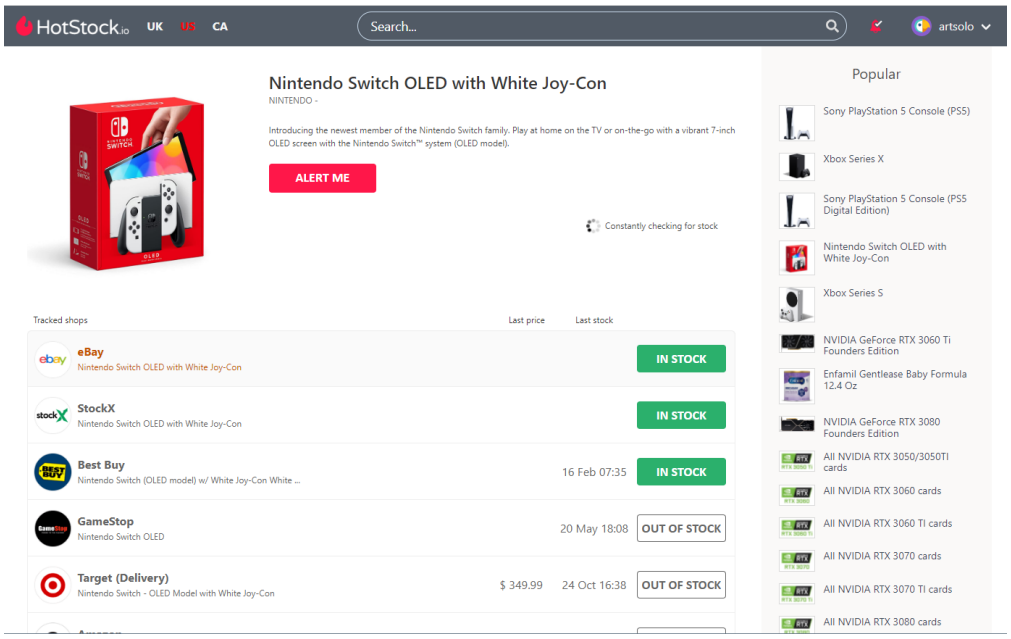

Рисунок 1.3 – Сторінка товару

Моніторинг товарів здійснюється цілодобово, що збільшує шанси користувача на замовлення бажаного продукту. Як тільки система фіксує, що товар з’явився у продажі вона відправляє повідомлення користувачу.

Хоча на перший погляд сервіс побудований досить не погано, але при користуванні, можна виявити ряд недоліків. Наприклад, визначено, що надходження товару на склад не завжди супроводжується повідомленням. Також, коректність роботи самої системи відстеження знаходиться під питанням. Є випадки, коли на сайті зазначено, що товар є у продажі в магазині, але перейшовши за посиланням можна помітити, що це не так.

Крім того, на сьогоднішній день інтерфейс системи не можна назвати сучасним і достатньо зручним для користувача.

У наступній таблиці представлено опис, та основні переваги та недоліки проаналізованого сервісу.

Таблиця 1.1 – Аналогічне ПЗ HotStock

Назва	HotStock
Архітектура	3tier web application
Виробник	Приватна компанія
Мова реалізації	PHP, JavaScript
Функції	1. пошук товару; 2. створення власного профілю; 3. перегляд каталогу товарів; 4. перегляд цін та наявності товару у магазинах; 5. можливість додати коментар до товару; 6. можливість підписатися на повідомлення про наявність товару.
Переваги	1. велика кількість відстежуваних магазинів; 2. велика кількість актуальних на ринку товарів; 3. відстежування товарів у реальному часі; 4. наявність браузерних та імейл повідомлень.

Кінець таблиці 1.1

Недоліки	1. недостатньо коректна робота системи стеження; 2. не зручна система повідомлень; 3. застарілий інтерфейс;
Веб-сайт	https://www.hotstock.io/

Проаналізувавши аналогічну систему було виявлено, що вона не є досконалою і буде доцільно розробити сервіс, у якому відсутні перераховані вище недоліки, щоб задовольнити потреби користувачів.

Для цього було складено специфікацію вимог до майбутнього програмного забезпечення.

1.4 Специфікація вимог до ПЗ

Веб-застосунок, що розробляється представляє собою систему відстеження і збереження даних про наявність товару на маркетплейсах. Сервіс призначається для користувачів, які бажають замовити товар, але не можуть цього зробити через те, що він більшість часу відсутній на складі, а коли з’являється, то продається за лічені хвилини.

Основні ролі користувачів:

– Гість – звичайний користувач який може переглядати каталог та дані про конкретні товари на сайті, зареєструватися в системі, оформити підписку на повідомлення про надходження товару.

– Адміністратор – користувач з повним набором прав доступу, який має ті самі можливості, що і гість, але крім того, може редагувати контент сайту, додавати та редагувати товари, управляти списком користувачів, модерувати коментарі та частково змінювати конфігурацію процесу стеження за товарами.

Функціональні вимоги до програмного забезпечення:

– Система повинна відстежувати та зберігати дані про назву, ціну та наявність товарів у інтернет магазинах.

– Частоту оновлення даних про товар можна буде змінювати у адміністративній панелі сайту окремо для кожного магазину.

– Можливість додавання та редагування товарів на сайті.

– Функція управлінням списком користувачів, редагування їх даних та можливість відмінити підписку.

– Керування контентом та SEO інформацією для усіх сторінок сайту.

– Можливість переглянути дані про відстежуванні магазини та надання інформації про процес скрапінгу для кожного з них.

– Система повинна відокремлювати товари і магазини для наступних країн: Канада, США, Велика Британія. А також, надавати можливість зручного переключення між ними.

– Функціонал, який буде дозволяти приховати товар для користувача та відключення товару від процесу стеження.

– Функція мануальної зміни статусу товару з «є у наявності» і навпаки.

– Функціонал по управлінню коментарями на сайті та створення чорного списку, у який можна додавати слова для валідації коментарів.

– Можливість користувачам переглядати каталог товарів на сайті, а також, інформації по кожному наявному товару.

– Наявність системи пошуку товарів у каталозі.

– Можливість залишити коментар на сторінці товару.

– Можливість оформити підписку на товар для зареєстрованих користувачів.

– Наявність персонального кабінету користувача.

– Система повідомлень про наявність товару повинна бути представлена у вигляді імейл розсилки, браузерних повідомлень та повідомлень на самому сайті.

– Можливість переглядати та керувати списком власних підписок користувачем.

– Функціонал авторизації користувачів за допомогою імейлу та паролю або через Google API.

Вимоги до дизайну інтерфейсу сайту:

– Дизайн повинен бути сучасним і лаконічним.

– Інтерфейс повинен бути інтуїтивно зрозумілим, елементи повинні розташовуватися так, щоб користувачу було зручно взаємодіяти з системою.

– Інтерфейс має бути адаптивним до всіх сучасних пристроїв, браузерів та розмірів екранів.

Вимоги до надійності та безпеки системи:

– Передбачити обробку усіх помилок та виключень, що можуть призвести до аварійного завершення роботи системи.

– Забезпечити валідацію вхідних даних.

– Доступ до API має бути захищеним і розподіленим за ролями у системі.

– Адміністративна панель має бути захищена від доступу для звичайних користувачів.

– Відправка повідомлень повинна відпрацьовувати без помилок не залежно від їх кількості.

– Система повинна витримувати великий трафік під час дропу товару.

Висновки до розділу 1

Виконано аналіз предметної області моніторингу та збереження інформації на маркетплейсах. При цьому були визначені основні особливості та аспекти, які необхідно дослідити для досягнення мети проекту.

Проведено дослідження існуючих методів стеження за інформацією на веб-ресурсах, таких як використання API та скрапінгу. Визначено слабкі та сильні сторони обох засобів та умови за яких їх буде доцільно використовувати.

Виконавши аналіз аналогічної системи, було виявлено низку переваг та недоліків, які вона має. Вони будуть враховані при розробці програмного продукту.

Також, були сформовані функціональні вимоги до майбутнього програмного забезпечення, вимоги до дизайну інтерфейсу веб-сайту та вимоги до безпеки і надійності системи.

2 МОДЕЛЮВАННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Методи та засоби моделювання

Виконавши аналіз предметної області, було досліджено специфіку та особливості бізнес процесів даної області, а також, сформовані функціональні та не функціональні вимоги до програмного забезпечення.

На основі цих даних необхідно провести моделювання предметної області, виконати її декомпозицію, виявивши основні сутності системи та взаємозв’язки між ними.

Моделювання – це проектування програмних додатків перед кодуванням. Моделювання є важливою частиною великих програмних проектів, а також корисним для середніх і навіть малих проектів. Модель відіграє аналогічну роль у розробці програмного забезпечення, яку відіграють креслення та інші плани (карти сайту, висоти, фізичні моделі) у будівництві хмарочоса. Використовуючи модель, ті, хто відповідає за успіх проекту розробки програмного забезпечення, можуть переконатися, що бізнес-функціональність є повною та правильною, що потреби кінцевого користувача задовольняються, а дизайн програми підтримує вимоги щодо масштабованості, надійності, безпеки, розширюваності та інших характеристик перед початком написання коду [1].

Опитування показують, що великі програмні проекти мають величезну ймовірність провалу – насправді велика ймовірність того, що у великій програмі не вдасться задовольнити всі вимоги вчасно та в рамках бюджету, більша, ніж успіх. При розробці проекту, потрібно зробити все можливе, щоб збільшити шанси на успіх, а моделювання – єдиний спосіб візуалізувати свій дизайн і перевірити його на відповідність вимогам, перш ніж почати кодувати.

Такий підхід допоможе ефективно та структуровано проводити розробку програми та кодування використовуючи об’єктно-орієнтовану парадигму.

Для цього побудуємо наступні діаграми:

– Діаграма прецедентів
– ER діаграма
– Діаграма класів

Побудову діаграм реалізуємо за допомогою UML.

Уніфікована мова моделювання (UML) – це сімейство графічних нотацій, основу якого є єдина мета-модель. Він допомагає в описі та проектуванні програмних систем, особливо систем, побудованих з використанням об'єктно-орієнтованих (ОО) технологій [2].

Моделі допомагають нам, дозволяючи працювати на вищому рівні абстракції. Модель може зробити це, приховуючи або маскуючи деталі, показуючи загальну картину або фокусуючи увагу на різних аспектах прототипу. В UML 2.0 ви можете зменшити масштаб від детального перегляду програми до середовища, де вона виконується, візуалізуючи з’єднання з іншими програмами або, збільшуючи ще більше, до інших сайтів. Крім того, ви можете зосередитися на різних аспектах програми, наприклад, бізнес-процесі, який він автоматизує, або перегляді бізнес-правил. Нова можливість вкладення елементів моделі, додана в UML 2.0, безпосередньо підтримує цю концепцію.

Уніфікована мова моделювання (UML) допомагає вам вказувати, візуалізувати та документувати моделі програмних систем, включаючи їх структуру та дизайн, у спосіб, який відповідає всім цим вимогам. (Ви також можете використовувати UML для бізнес-моделювання та моделювання інших непрограмних систем.) Використовуючи будь-який із великої кількості інструментів на основі UML на ринку, ви можете проаналізувати вимоги майбутнього додатку та розробити рішення, яке їм відповідає,

представляючи результати з використанням тринадцяти стандартних типів діаграм UML 2.0 [1].

2.2 Проектування діаграми прецедентів

В UML діаграми варіантів використання моделюють поведінку системи та допомагають охопити вимоги системи.

Діаграми варіантів використання описують функції високого рівня та область застосування системи. Ці діаграми також визначають взаємодії між системою та її акторами. Варіанти використання та дійові особи на діаграмах прецедентів описують, що робить система і як учасники її використовують, але не те, як система працює всередині.

Діаграми варіантів використання ілюструють і визначають контекст і вимоги або всієї системи, або важливих частин системи. Можна змоделювати складну систему за допомогою однієї діаграми варіантів використання або створити багато діаграм варіантів використання для моделювання компонентів системи. Зазвичай розробка діаграми варіантів використання відбувається на ранніх етапах проекту і звертаєтеся до них протягом усього процесу розробки [3].

Діаграми варіантів використання корисні в таких ситуаціях:

– Перед початком проекту можна створити діаграму варіантів використання для моделювання бізнесу, щоб усі учасники проекту мали розуміння працівників, клієнтів та діяльності підприємства.

– Збираючи вимоги, корисно буде створити діаграму варіантів використання, щоб охопити системні вимоги та представити іншим, що має робити система.

– На етапах аналізу та проектування можна використовувати варіанти використання та акторів із діаграм прецедентів, щоб визначити класи, які потрібні системі.

– На етапі тестування можна використовувати діаграми варіантів використання для визначення тестів для системи.

Далі наведено моделі на діаграмах варіантів використання:

– **Випадки використання.** Описує функцію, яку виконує система для досягнення мети користувача. Випадок використання повинен давати спостережуваний результат, який є цінним для користувача системи.

– **Актори.** Актор представляє роль користувача, який взаємодіє з системою, яку ви моделюєте. Користувач може бути користувачем-людиною, організацією, машиною або іншою зовнішньою системою.

– **Підсистеми.** У моделях UML підсистеми є типом стереотипних компонентів, які представляють незалежні, поведінкові одиниці в системі. Підсистеми використовуються в діаграмах класів, компонентів і варіантів використання для представлення великомасштабних компонентів у системі.

– **Зв'язки.** В UML зв’язок – це зв’язок між елементами моделі. Відношення UML – це тип елемента моделі, який додає семантику до моделі, визначаючи структуру та поведінку між елементами моделі.

У розроблюваній системі визначено 2 основних актора:

– Користувач

– Адміністратор

На рис. 2.1 зображено діаграму прецедентів сервісу стеження за товарами на маркетплейсах.

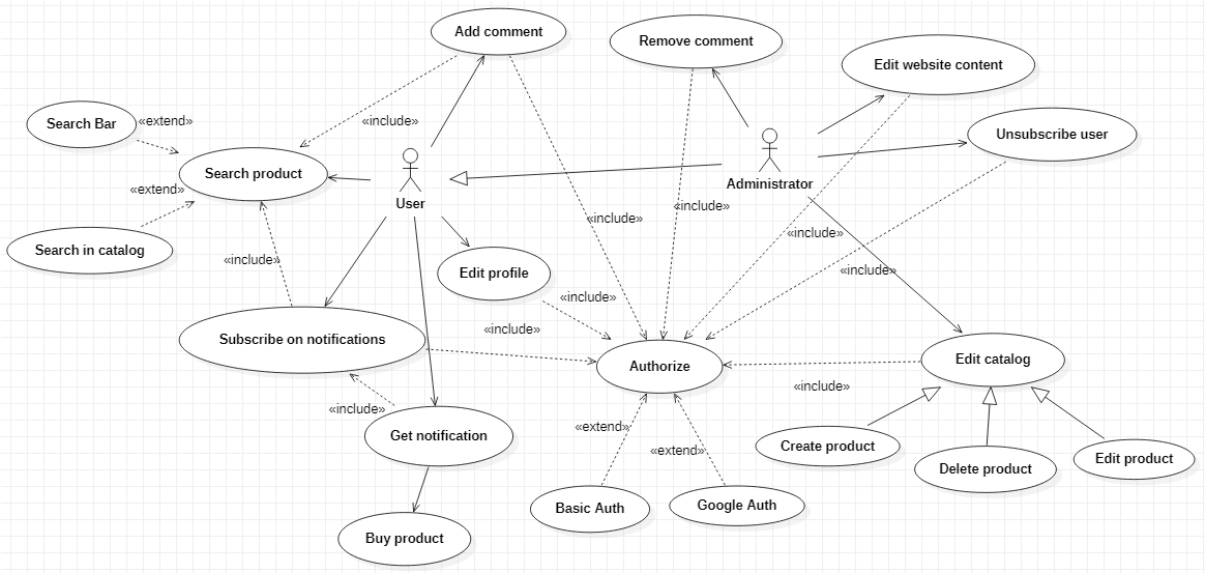

Рисунок 2.1 – Діаграма прецедентів

На діаграмі вказані актори системи та можливі випадки використання даної системи акторами.

2.3 Проектування ER діаграм

Діаграма зв’язків між об’єктами (ER) – це тип блок-схеми, яка ілюструє, як «суб’єкти», такі як люди, об’єкти або поняття, пов’язані один з одним у системі. Діаграми ER найчастіше використовуються для проектування або налагодження реляційних баз даних у сферах програмної інженерії, бізнес-інформаційних систем, освіти та досліджень. Також відомі як ERD або моделі ER, вони використовують певний набір символів, таких як прямокутники, ромби, овали та сполучні лінії, щоб відобразити взаємозв’язок сутностей, відносин та їх атрибутів. Вони відображають граматичну структуру з сутностями як іменники, а відносини як дієслова.

Діаграми ER пов’язані з діаграмами структури даних (DSD), які зосереджуються на зв’язках елементів всередині сутностей, а не на відносинах між самими сутностями. Діаграми ER також часто використовуються в поєднанні з діаграмами потоків даних (DFD), які відображають потік інформації для процесів або систем [6].

2.3.1 Концептуальна модель даних

Етап проектування концептуальної моделі починається зі створення концептуальної моделі даних предметної області, яка повністю не залежить від деталей реалізації, таких як цільова СУБД, прикладні програми, мови програмування, апаратна платформа, проблеми з продуктивністю або будь-які інші фізичні міркування [4].

Для побудови концептуальної моделі необхідно виконати наступні етапи:

– Першим кроком у побудові концептуальної моделі даних є визначення сутностей системи.

– Після визначення сутностей наступним кроком є визначення всіх зв’язків, які існують між цими сутностями.

– Наступним кроком у методології є визначення типів відносин між сутностями.

– Останній етапом є визначення атрибутів для кожної сутності.

Виконавши всі етапи проектування маємо діаграму прецедентів наведену на рис. 2.2.

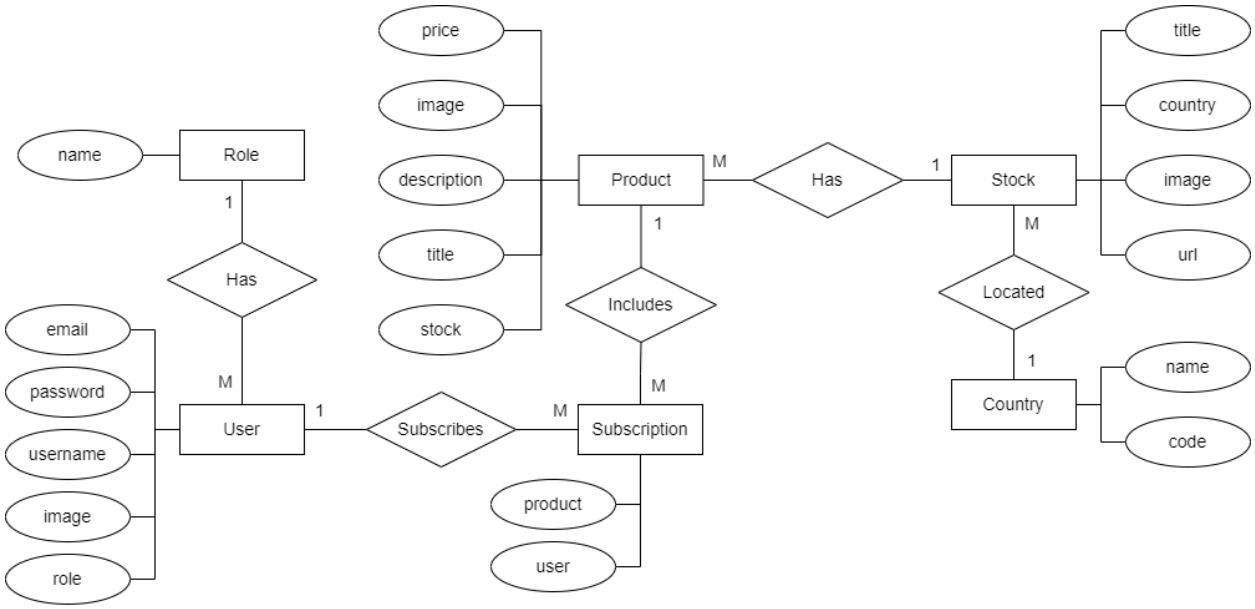

Рисунок 2.2 – Концептуальна модель даних

Відповідно до побудованої діаграми основними сутностями предметної області є:

– Продукт

– Магазин

– Країна

– Підписка

– Користувач

– Роль

Для кожної сутності визначені атрибути та типи зв’язків з іншими сутностями.

2.3.2 Логічна модель даних

Фаза проектування логічної моделі даних відображає концептуальну модель даних на логічну модель, на яку впливає модель даних цільової бази даних (наприклад, реляційна модель). Логічна модель даних є джерелом інформації для фази фізичного проектування, надаючи розробнику фізичної моделі засіб для досягнення компромісів, які дуже важливі для розробки ефективної бази даних [4].

Проаналізувавши предметну область, можна виділити наступні типи сутностей:

– **Продукт.** Містить загальну інформацію про товари в системі, дані про які необхідно відстежувати у інтернет магазинах, а також, мета дані для пошукових ботів та деякі налаштування.

– **Магазин.** Зберігає інформацію про маркетплейси на яких будуть відстежуватися товари та дані, які потрібні для роботи системи стеження за товарами.

– **Трекер**. Важлива сутність, що поєднує магазин і товар. Вона зберігає результати роботи системи стеження, а також дані, які необхідні для самого процесу скрапінгу.

– **Група.** Сутність, яка потрібна для групування трекерів на групи.

– **Джерело.** Містить довідкову інформацію про варіанти стеження для магазинів.

– **Ідентифікатор.** Містить варіанти ідентифікаторів товару для стеження.

– **Країна.** Зберігає список країн, магазини яких використовуються для відстеження товарів.

– **Користувач.** Містить загальну інформацію про користувачів систему, а також дані необхідні для авторизації в системі.

– **Підписка.** Сутність, у якій знаходиться інформація про список користувачів і їх підписок на повідомлення про надходження товарів на склад магазинів.

– **Коментар.** Містить текст коментарів, які користувачі можуть залишати на сторінках продукту.

– **Роль.** Сутність, що зберігає список ролей, які можуть мати користувачі. Відповідає за розподілення прав між користувачами.

– **Токен.** Зберігає токени пристроїв коритувачів для відправки браузерних повідомлень.

– **Девайс.** Об’єднує токени та користувачів.

– **Сторінка.** Містить загальні та мета дані про сторінки веб-застосунку.

– **Контент**. Містить контент для веб-сторінок сайту.

Використавши UML нотацію та сформовані сутності на рис. 2.3 відображено результат проектування логічної моделі.

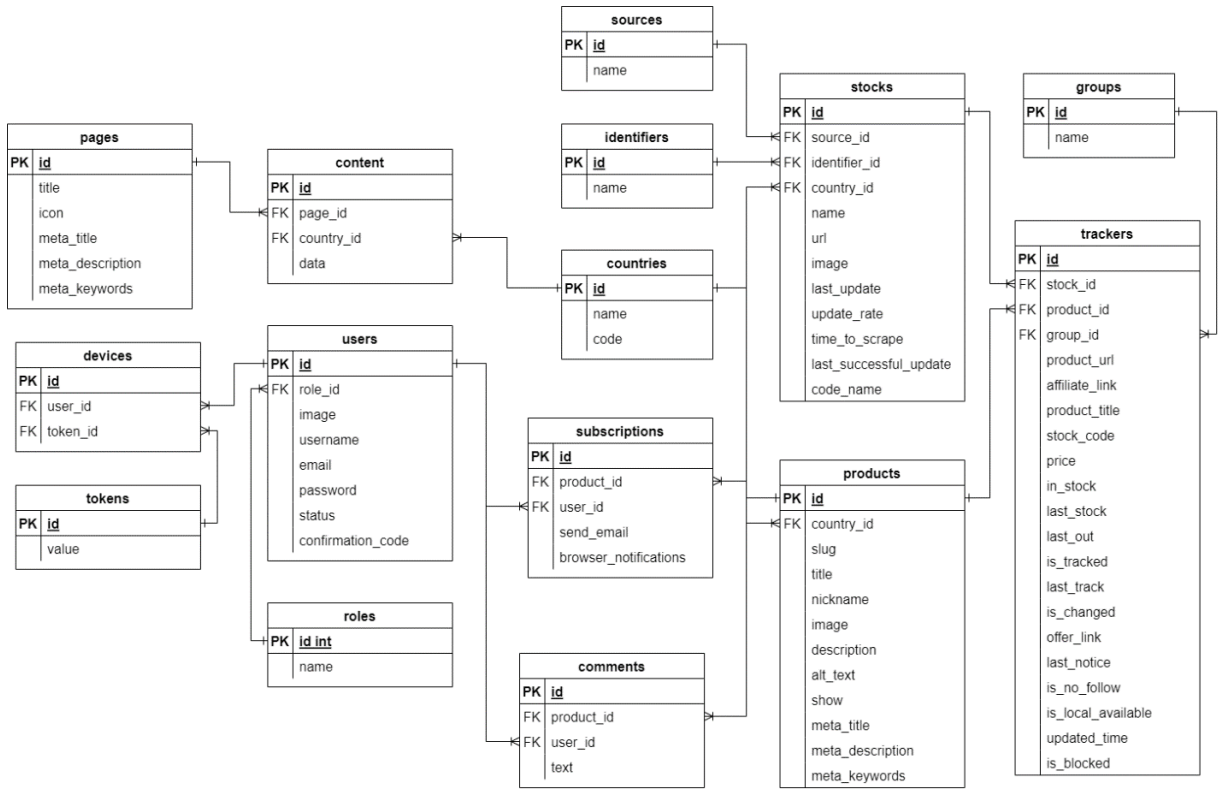

Рисунок 2.3 – Логічна модель даних

2.3.3 Фізична модель даних

Фаза проектування фізичної моделі даних дозволяє дизайнеру приймати рішення про те, як база даних має бути реалізована. Тому фізичне проектування розробляється під конкретну СКБД. Між фізичним і логічним проектуванням існує зворотний зв’язок, оскільки рішення, прийняті під час фізичного проектування для підвищення продуктивності, можуть вплинути на логічну модель даних [4].

Фізична модель даних визначає дані засобами конкретної СКБД. Обмеження, що є в логічній моделі даних, реалізуються різними засобами СКБД, наприклад, за допомогою індексів, декларативних обмежень цілісності, тригерів, процедур, що зберігаються. При цьому знову ж таки рішення, прийняті на рівні логічного моделювання, визначають деякі межі, в межах яких можна розвивати фізичну модель даних. Так само, в межах цих кордонів можна приймати різні рішення. Наприклад, відносини, що містяться в логічній моделі даних, повинні бути перетворені на таблиці, але для кожної таблиці можна додатково оголосити різні індекси, що підвищують швидкість звернення до даних. Багато чого залежить від конкретної СКБД.

Якщо фізична модель даних реалізована засобами реляційної СКБД, то відносини, розроблені на стадії формування логічної моделі даних, перетворюються на таблиці, атрибути стають стовпцями таблиць, для ключових атрибутів створюються унікальні індекси, домени перетворюються на типи даних, прийняті конкретної СКБД.

Власне база даних та інформаційна система. І, нарешті, як наслідок попередніх етапів з'являється власне сама база даних. База даних реалізована на конкретній програмно-апаратній основі, і вибір цієї основи дозволяє суттєво підвищити швидкість роботи з базою даних. Наприклад, можна вибирати різні типи комп'ютерів, змінювати кількість процесорів, обсяг оперативної пам'яті, дискові підсистеми тощо. Дуже велике значення має також налаштування СКБД у межах обраної програмно-апаратної

платформи. На рис. 2.4 відображено фізичну модель даних, яка згенерована засобами програмного забезпечення phpMyAdmin.

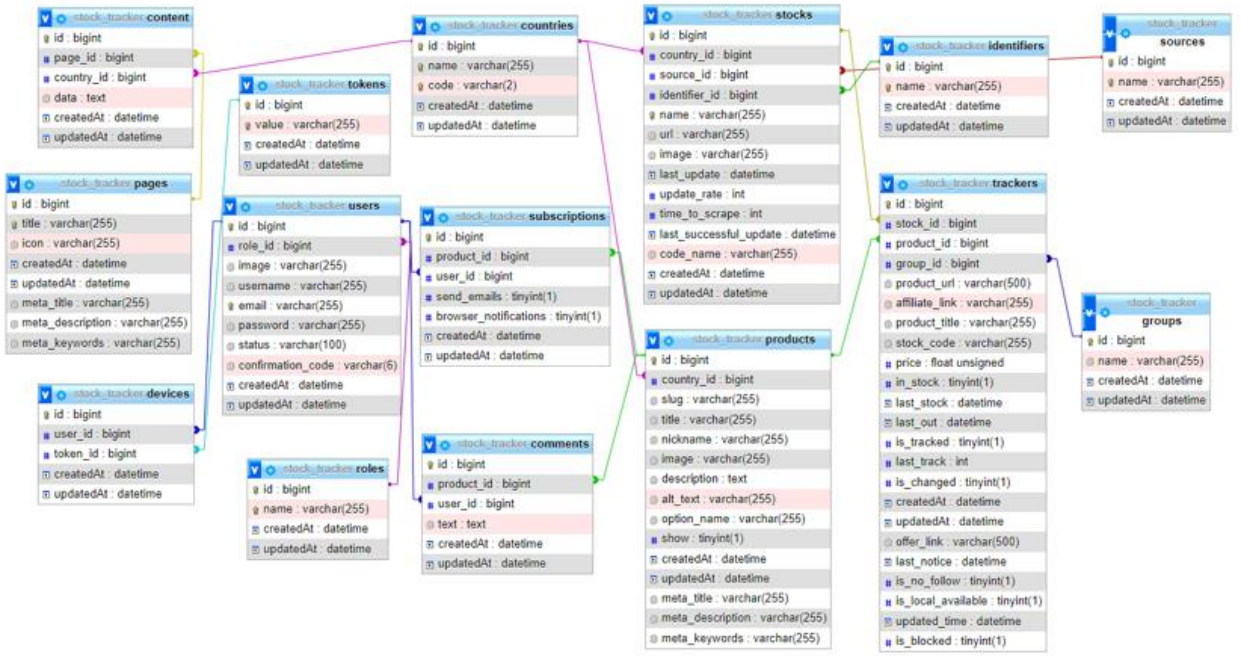

Рисунок 2.4 – Фізична модель даних

Далі, у таблиці 2.1 наведено список усіх таблиць майбутньої бази даних та їх атрибутів з описом.

Таблиця 2.1 – Атрибути таблиць у базі даних

Таблиця	**Атрибут**	**Опис атрибуту**
products	id	Ключ
	country_id	Зовнішній ключ, id (countries)
	slug	Назва товару у нижньому реєстрі без проміжків
	titile	Назва товару
	nickname	Скорочена назва товару
	image	Ім’я файлу з зображенням товару
	description	Опис товару
	alt_text	Напис при наведенні на зображення
	show	Відображення товару на сайті

Продовження таблиці 2.1

	meta_title	Мета назва товару для SEO
	meta_description	Мета опис товару для SEO
	meta_keywords	Ключові слова для пошуку
stocks	id	Ключ
	source_id	Зовнішній ключ, id (sources)
	identifier_id	Зовнішній ключ, id (identifiers)
	country_id	Зовнішній ключ, id (countries)
	name	Назва магазину
	url	Доменне ім’я магазину
	image	Ім’я файлу з зображенням логотипу
	last_update	Час останнього оновлення даних
	update_rate	Частота скрапінгу в хвилинах
	time_to_scrape	Час до старту скрапінгу
	last_successful_update	Час останнього успішного скрапінгу
	code_name	Назва ідентифікатора товару
trackers	id	Ключ
	stock_id	Зовнішній ключ, id (stocks)
	product_id	Зовнішній ключ, id (products)
	group_id	Зовнішній ключ, id (groups)
	product_url	Адреса сторінки товару в магазині
	affiliate_link	Партнерське посилання
	product_title	Назва товару в магазині
	stock_code	Код товару у магазині
	price	Ціна товару
	in_stock	Наявність товару
	last_stock	Час останнього перебування в наявності

Продовження таблиці 2.1

	last_out	Час останньої відсутності товару в наявності
	is_tracked	Необхідність відстеження товару
	last_track	Час останнього стеження
	is_changed	Чи були отриманні дані по товару
	offer_link	Посилання для додавання у кошик
	last_notice	Час останньої відправки повідомлень
	is_no_follow	Відображати посилання на товар на сайті з атрибутом no-follow
	is_local_available	Чи є в наявності локально
	updated_time	Час останнього оновлення
	is_blocked	Чи заблоковано товар
groups	id	Ключ
	name	Назва групи
sources	id	Ключ
	name	Назва джерела
countries	id	Ключ
	name	Назва країни
	code	Код країни
subscriptions	id	Ключ
	product_id	Зовнішній ключ, id (products)
	user_id	Зовнішній ключ, id (users)
	send_emails	Відсилати емейл повідомлення
	browser_notifications	Відсилати браузерні повідомлення
comments	id	Ключ
	product_id	Зовнішній ключ, id (products)

Кінець таблиці 2.1

	user_id	Зовнішній ключ, id (users)
	text	Текст коментаря
users	id	Ключ
	role_id	Зовнішній ключ, id (roles)
	image	Назва файлу з аватаром користувача
	username	Ім’я користувача
	email	Емейл
	password	Пароль
	status	Статус
	confirmation_code	Код підтвердження реєстрації
roles	id	Ключ
	name	Назва ролі
pages	id	Ключ
	title	Назва сторінки
	icon	Назва файлу з іконкою сторінки
	meta_title	Мета назва сторінки для SEO
	meta_description	Мета опис сторінки для SEO
	meta_keywords	Ключові слова для пошуку
tokens	id	Ключ
	value	Значення токену
devices	user_id	Зовнішній ключ, id (users)
	token_id	Зовнішній ключ, id (tokens)
content	id	Ключ
	page_id	Зовнішній ключ, id (pages)
	country_id	Зовнішній ключ, id (countries)
	data	Контент сторінки у JSON форматі

2.4 Проектування діаграми класів

Діаграма класів описує типи об'єктів у системі та різні види статичних зв'язків, які існують між ними. Діаграми класів також показують властивості та операції класу та обмеження, які застосовуються до способу з’єднання об’єктів. UML використовує термін функція як загальний термін, який охоплює властивості та операції класу [5].

Відповідно до результатів аналізу предметної області на рис. 2.5 представлено діаграму класів розроблюваної системи.

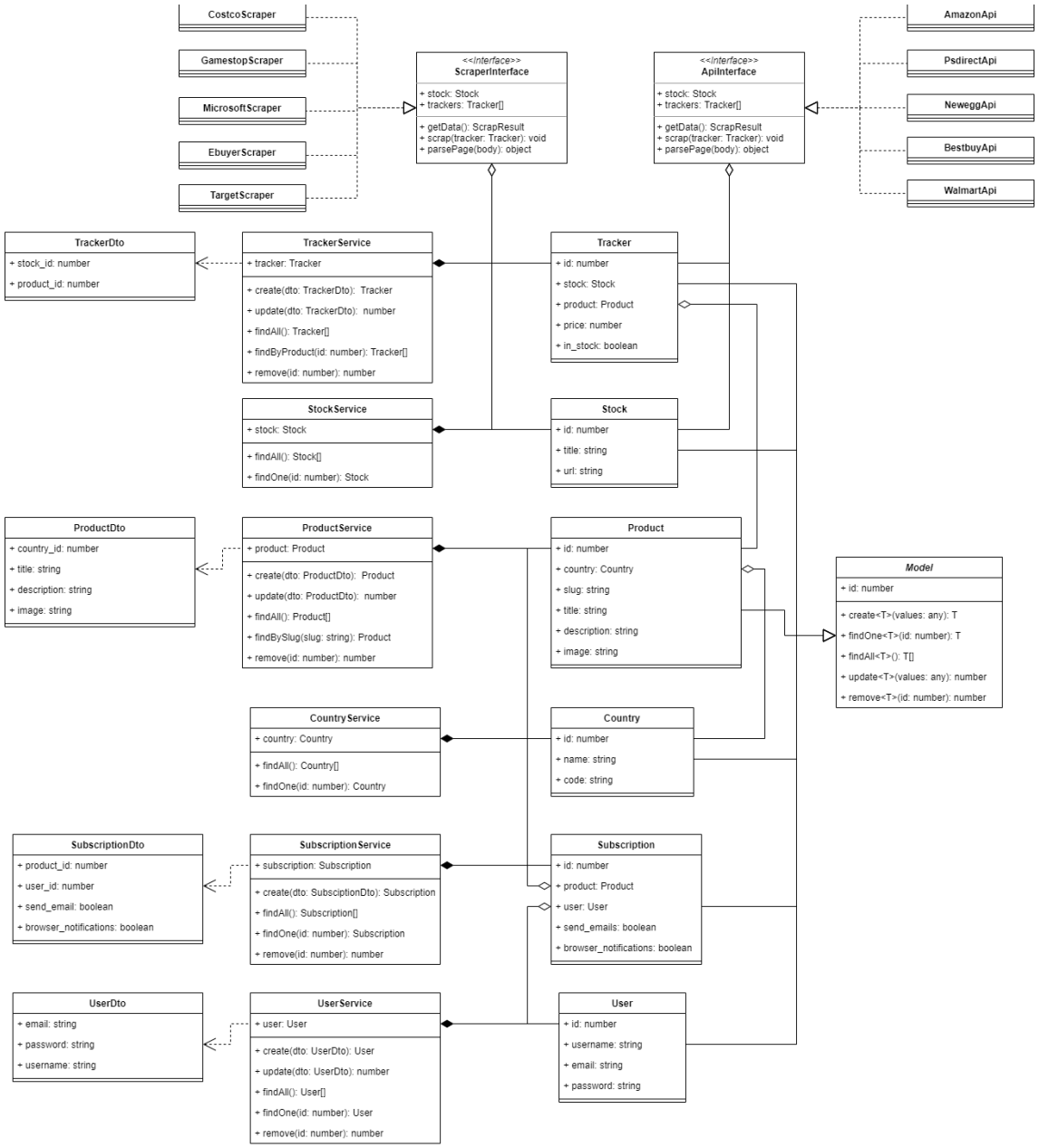

Рисунок 2.5 – Діаграма класів

Висновки до розділу 2

В межах розділу були досліджені методи моделювання та декомпозиції предметної області.

Виявлено основні сутності предметної області та типи зв’зків між ними. Також, були розписані атрибути кожної сутності.

За допомогою графічної нотації UML побудовано діаграму прецедентів, концептуальну, логічну і фізичну модель даних, а також, діаграму класів.

3 АНАЛІЗ ЗАСОБІВ РОЗРОБКИ

3.1 Архітектура веб-застосунку

При проектуванні програмного забезпечення необхідно визначитися з архітектурою яка буде використовуватися для розробки. Якісна архітектура це насамперед вигідна архітектура, що робить процес розробки та супроводу програми більш простим та ефективним. Програму з гарною архітектурою легше розширювати та змінювати, а також тестувати, налагоджувати та розуміти. Можна сформулювати список критеріїв, які притаманні гарній архітектурі:

- Ефективність системи
- Гнучкість системи
- Розширюваність системи
- Масштабованість процесу розробки
- Зручність тестування
- Можливість повторного використання
- Добре структурований, читабельний та зрозумілий код.

Існує велика кількість варіантів архітектури, які застосовуются для розробки веб-застосунків, найпоширенішими серед них є:

- Багатошарова архітектура
- Багаторівнева архітектура
- Сервіс-орієнтована архітектура
- Мікросервісна архітектура

Для розробки застосунку моніторингу за товарами на маркетплехсах було вирішено використовувати трирівневу архітектуру, яка є підвидом багаторівневої.

Трирівнева архітектура програми – це модульна клієнт-серверна архітектура, яка складається з рівня представлення, рівня застосунку та рівня даних. Рівень даних зберігає інформацію, рівень застосунку обробляє логіку, а рівень представлення – це графічний інтерфейс користувача (GUI), який

взаємодіє з двома іншими рівнями. Ці три рівні є логічними, а не фізичними, і можуть виконуватися на одному фізичному сервері, а можуть і на окремих [7].

Така архітектура має високу масштабованість як по горизонталі, так і по вертикалі. Реалізація n-рівневої системи, як правило, обходиться дорожче, але забезпечує високу продуктивність. Тому вона зазвичай застосовується у великих та комплексних програмних рішеннях.

Цей підхід можна поєднувати із сучасною сервіс-орієнтованою архітектурою, щоб створювати найскладніші моделі. Оскільки реалізація може виявитися дорогою з точки зору часу та ресурсів, рекомендується використовувати його для складних програм, що потребують продуктивності та масштабованості.

На рис. 3.1 відображено схему класичного трирівневого застосунку.

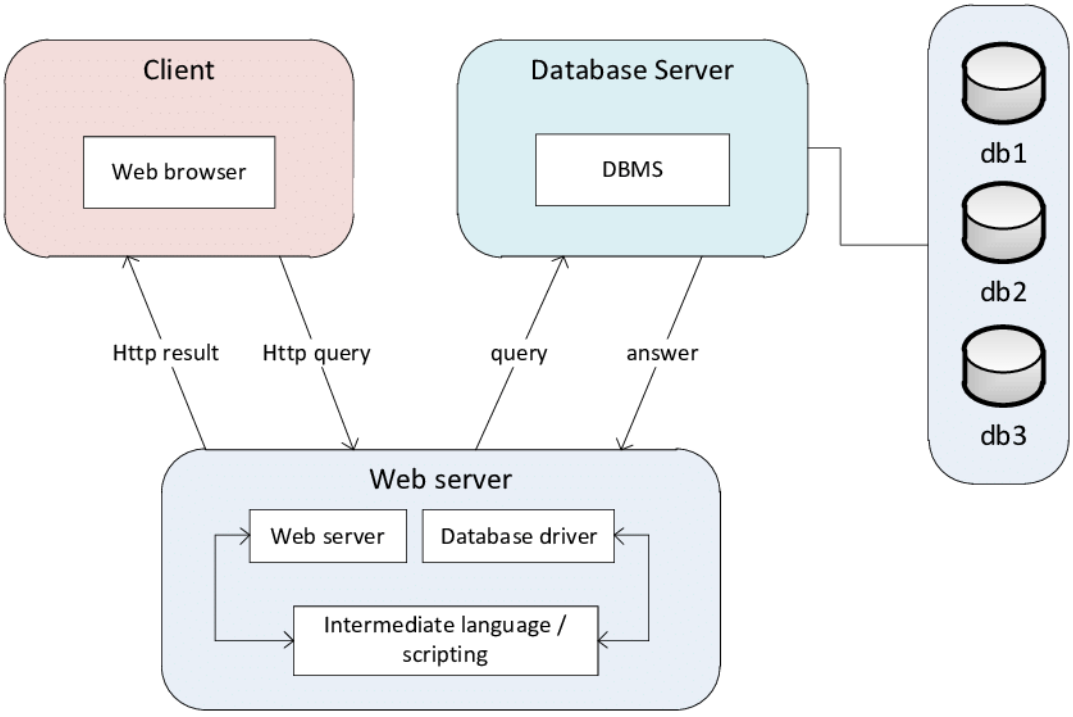

Рисунок 3.1 – Схема трирівневої архітектури

Головна перевага трирівневої архітектури полягає в тому, що, оскільки кожен рівень працює на власній інфраструктурі, він може одночасно

розроблятися окремою командою розробників і за потреби оновлюватися або масштабуватися, не впливаючи на інші рівні.

Протягом десятиліть трирівнева архітектура була переважною архітектурою для клієнт-серверних програм. Сьогодні більшість трирівневих додатків є цілями для модернізації з використанням хмарних технологій, таких як контейнери та мікросервіси, а також для міграції в хмару [8].

Наведемо детальний опис кожного рівня архітектури:

– **Рівень представлення.** Цей рівень відповідає за інтерфейс користувача програми. Його головне призначення – відображати інформацію та отримувати її від користувача. Для взаємодії з користувачем рівень представлення використовує веб-браузер. Розробка даного рівня зазвичай відбувається за допомогою HTML, CSS та JavaScript.

– **Рівень застосунку**, також відомий як логічний рівень або середній рівень, є серцем програми. На цьому рівні інформація, зібрана на рівні представлення, обробляється з використанням бізнес-логіки, певного набору бізнес-правил. Рівень застосунку також може додавати, видаляти або змінювати інформацію на рівні даних. Рівень застосунку зазвичай розробляється за допомогою Python, Java, Perl, Node, PHP або Ruby і взаємодіє з рівнем представлення за допомогою викликів API.

– **Рівень даних**, який іноді називають рівнем бази даних, рівнем доступу до даних або серверною частиною, є місцем, де інформація, оброблена програмою, зберігається та керується. Це може бути система управління реляційною базою даних, така як PostgreSQL, MySQL, MariaDB, Oracle, DB2, Informix або Microsoft SQL Server, або сервер баз даних NoSQL, такий як Cassandra, CouchDB або MongoDB.

У трирівневій програмі вся комунікація проходить через рівень застосунку. Рівень представлення та рівень даних не можуть безпосередньо взаємодіяти один з одним [8].

Обравши архітектуру застосунку, наступним кроком є визначення з технологіями розробки, для реалізації усіх частин застосунку: представлення(frontend), серверна частина(backend) та робота з базою даних.

3.2 Засоби та технології розробки представлення

3.2.1 Базові технології розробки GUI

Реалізація рівня представлення належить до розробки frontend частини застосунку. Frontend відповідає за візуальне представлення графічного інтерфейсу користувача, можливість комунікації користувача з системою за допомогою інтерактивності та отримання і передачу даних до рівня бізнес-логіки за допомогою запитів до API.

Основний набір інструментів для інтерфейсу чітко визначений: HTML, CSS і JavaScript. Однак технології розробки інтерфейсу можуть бути розширені за допомогою менеджерів пакетів, препроцесорів CSS, фреймворків та інших технологій [9].

HTML (або мова гіпертекстової розмітки) – це мова, призначена для створення веб-сайтів, які пізніше може переглядати будь-хто, хто має доступ до Інтернету. HTML зазвичай використовується для структурування веб-документа. Він визначає такі елементи, як заголовки або абзаци, і дозволяє вставляти зображення, відео та інші медіа.

CSS (або каскадні таблиці стилів) – це мова таблиць стилів. Вона використовується для визначення того, як елементи HTML мають бути представлені на веб-сторінці з точки зору дизайну, макета та варіацій для різних пристроїв з різними розмірами екрана.

JavaScript (JS) є однією з найпопулярніших мов сценаріїв. Він здебільшого відомий тим, що надає повний набір технологій для розробки інтерфейсу та сервера. Оскільки ми говоримо про перший, він використовується, щоб зробити веб-сторінки динамічними.

На рис. 3.2 відображено список основних засобів розробки будь-якого сучасного веб-застосунку.

Рисунок 3.2 – Базові засоби розробки

Даний набір інструментів є базовим для практично будь-якого веб-застонунку або сайту. Отже, всі ці технології було використано при розробці сервісу. Для підвищення швидкості та якості використання каскадних таблиць стилів використовують CSS препроцесори та фреймворки.

3.2.2 Bootstrap та CSS препроцесори

Фреймворк CSS – це набір файлів CSS і HTML за замовчуванням. Він розширює можливості фронт-енд розробника щодо дизайну веб-сайтів. На додаток до допомоги під час створення адаптивного дизайну, фреймворки CSS також мають чіткі та симетричні макети, позбавляючи розробників від написання коду з нуля в будь-якому випадку. Зазвичай вони вважаються хорошим вибором для різних платформ і розмірів екранів. Завдяки поширеним компонентам інтерфейсу користувача, системам сіток, макетам та багатьом іншим функціям, фреймворки CSS значно прискорюють робочий процес розробки. У всесвіті CSS існує багато фреймворків:

– Повнофункціональний (Bootstrap, Foundation, Semantic UI та багато іншого).

– Спрямований на Material Design: (Materialize and Material Design Lite), і Lightweight: (Pure).

Для розробки даного проекту було вирішено використати framework Bootstrap. Він має добре продуману сітку(рис. 3.3), яка значно полегшує процес розробки, особливо, адаптацію інтерфейсу для різних пристроїв.

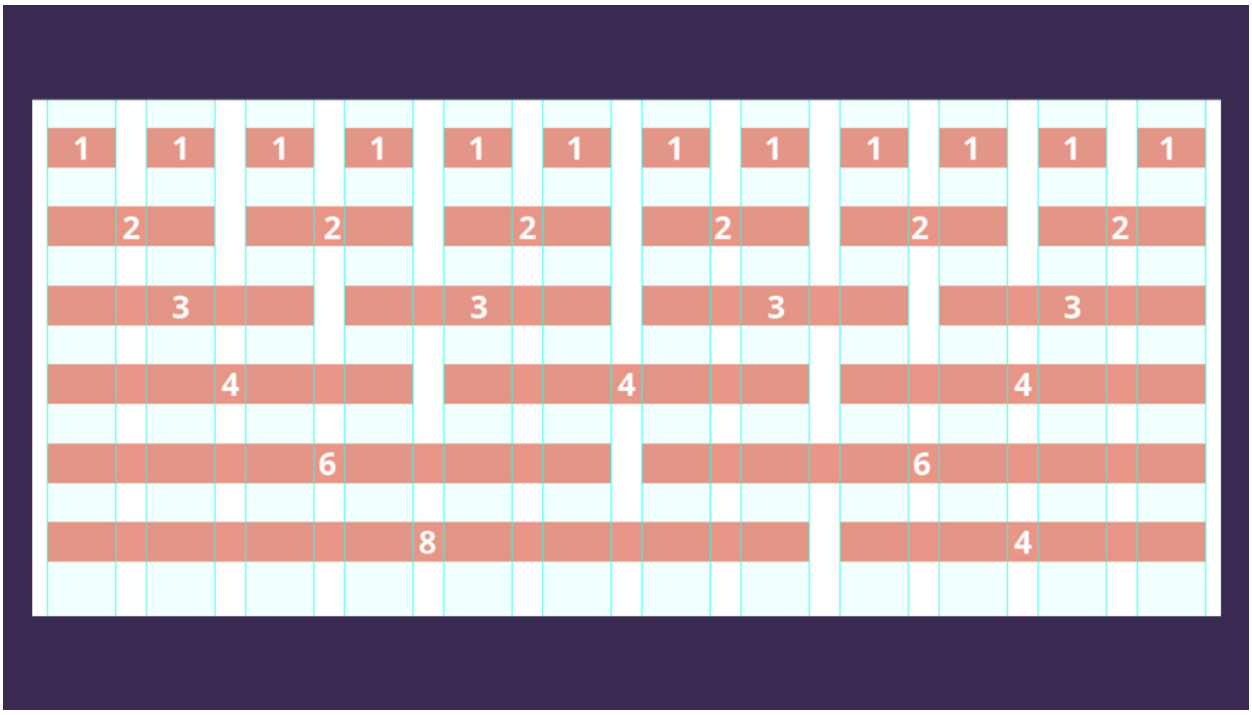

Рисунок 3.3 – Структура сітки Bootstrap

Складання CSS є рутиною, тоді як короткі завдання, такі як пошук значень кольору, закриваючі теги або будь-які інші повторювані операції, займають багато часу. Ось тут і стане в нагоді препроцесор. Препроцесор CSS – це мова сценаріїв, яка розширює CSS і збирає його в загальний CSS.

SASS і LESS є найпоширенішими препроцесорами. Вони поділяють деякі спільні основи, наприклад:

– Елементи синтаксису

– Зворотна сумісність із звичайними файлами CSS

Однак між ними також є чимало відмінностей:

– SASS означає Syntactically Awesome Style Sheets. Sass працює на Ruby і відображається на стороні сервера. Через походження мови Ruby інсталяція здійснюється за допомогою так званих gems (кілька бібліотек Ruby/Rails).

– LESS розшифровується як Leaner Style Sheets. Це бібліотека JavaScript, яка відображається на стороні клієнта в браузері. Розробники вибирають LESS набагато частіше, коли використовують JavaScript з

таблицями стилів. Це нагадує складання звичайного CSS. Технологія дозволяє повторно використовувати фрагменти CSS у файлах LESS.

При розробці системи було вирішено використання препроцесору SASS, а точніше його діалекту – SCSS. Головну роль зіграла різниця у синтаксисі препроцесорів.

3.2.3 Порівняння JavaScript фреймворків

На сьогоднішній день існує велика кількість потужних JavaScript фреймворків. Але найбільш популярними серед них за правом вважаються React.js, Angular та Vue.js.

Щоб обрати один з них необхідно виявити переваги та недоліки кожного та порівняти їх між собою враховуючи особливості та вимоги розроблюваної системи.

Vue.js. Vue – це фреймворк JavaScript, який використовується для створення інтерфейсів користувача для веб-додатків. Він був створений Еваном Ю у 2014 році і найкраще підходить для одно сторінкових додатків, асинхронних компонентів, прототипів і рендерингу на стороні сервера. Якщо ви вже знайомі з JavaScript, ви можете легко вибрати Vue.js, оскільки вам не потрібно знати багато про JSX або ES2016, щоб почати. Остання версія Vue містить API композиції, глобальну зміну mounting API та фрагменти. Деякі відомі компанії, які використовують Vue.js – це Facebook для маркетингу в каналі новин, продукту Adobe Portfolio та інтерфейсу Grammarly.

Переваги:

– Простий для освоєння
– Легкість та швидкість Vue дає високу продуктивність
– Хороша документація та велика спільнота
– Гнучкість в розробці шаблонів

Недоліки:

– Обмежена кількість плагінів
– Основна частина матеріалів написана китайською мовою

– Не найнадійніший вибір для великих систем

Angular. Інший популярний інтерфейсний фреймворк – Angular. Створений Міско Хевері та Адамом Аброно в 2009 році як побічний проект, зараз він підтримується Google. Його поточна версія, випущена в травні 2019 року, називається Angular 8 і повністю побудована за допомогою Typescript. Фреймворк містить диференціальне завантаження, синтаксис відкладеного завантаження, а також API конструктора та робочої області.

За допомогою Angular ви можете використовувати HTML для своїх шаблонів і зв'язувати HTML, щоб чітко відображати ваші компоненти. Це також допомагає зменшити кількість коду, який потрібно написати, за допомогою прив'язування даних і ін'єкції залежностей, і все це в браузері. Деякі відомі компанії з Angular у своєму технологічному стеку — це Google, Udemy та Amazon.

Переваги:

– Зручний механізм прив'язки даних
– Підвищена продуктивність сервера
– Використання асинхронного програмування
– Чистий код завдяки використанню TypeScript
– Перевикоритовуванні компоненти

Недоліки:

– Складна міграція зі старих версій
– Складніть освоєння
– Іноді надлишково складна структура компонентів

React.js. React – це широко популярна інтерфейсна бібліотека, створена інженером-програмістом Facebook Джорданом Уоком у 2011 році. Вона використовується для створення динамічних інтерфейсів користувача і користується популярністю серед розробників JavaScript. React має процвітаючу спільноту в Інтернеті, чудову документацію та навіть курси на

своєму веб-сайті. Через деякий час після його випуску команда React створила React Native, фреймворк для гібридної мобільної розробки.

React відтворює веб-сторінки таким чином, щоб вони були динамічними та реагували на введення користувача. Це бібліотека JavaScript з відкритим вихідним кодом, яка дозволяє створювати вражаючі інтерфейси на стороні користувача, які є швидкими та зручними для SEO. Деякі функції React.js включають компоненти для багаторазового використання, одностороннє прив’язування даних і віртуальний DOM. Деякі відомі програми, створені за допомогою React, — це Facebook (домашня сторінка), Instagram (геолокація та тегування) та New York Times (інтерфейс проекту «Червона доріжка Оскара»).

Переваги:

– Пере використання HTML коду
– Легкий в освоєнні
– Використання віртуального DOM надає високу продуктивність
– Легкий перехід зі старих версій
– Наявність сховища Redux

Недоліки:

– Через велику швидкість розвитку розробникам складно встигати за останніми тенденціями бібліотеки.
– Погана документація, через те, що вона не встигає за темпами розвитку.
– Немає чіткого стандарту написання коду, що може стати проблемою для роботи в команді.

На основі дослідження було вирішено використовувати фреймворк Vue,js. А точніше фреймворк на основі Vue – Nuxt.js.

3.2.4 Фреймворк Nuxt.js

Nuxt.js – це безкоштовна бібліотека JavaScript з відкритим вихідним кодом на основі Vue.js, Node.js, Webpack і Babel.js. Nuxt натхненний Next.js, який є фреймворком подібного призначення, заснованим на React.js.

Фреймворк рекламується як «Мета-фреймворк для універсальних додатків». Термін універсальний тут використовується у значенні, що мета фреймворка полягає в тому, щоб дозволити користувачам створювати веб-подання в JavaScript, використовуючи систему одно файлових компонентів Vue.js, і яка може функціонувати як одно сторінкова програма у браузері (SPA), а також веб-погляди, відтворені на сервері, які потім (після відтворення сервера) «регідратуються» до повної функціональності SPA. Крім того, фреймворк дає користувачам можливість повністю попередньо відтворювати вміст або його частини на сервері та обслуговувати у вигляді статичних генераторів сайтів.

Переваги цього підходу, серед іншого, полягають у зменшенні часу на інтерактивність і покращенні SEO в порівнянні з SPA, завдяки тому, що повний вміст кожної сторінки обслуговується веб-сервером до того, як буде виконано будь-який клієнтський JavaScript. Інакше кажучи, можна підтримувати як переваги традиційних відтворених HTML-сторінок на стороні сервера, так і покращену інтерактивність і розширений інтерфейс користувача SPA. Основна перевага самого фреймворка Nuxt.js полягає в тому, що він спрощує конфігурацію та налаштування таких програм для розробника програми, який може просто розробляти частини інтерфейсу програми, як якщо б це був більш поширений один файл Vue.js. додаток [11].

Переваги використання Nuxt.js:

– Полегшує створення застосунку з нуля надаючи попередньо налаштовані модулі сховища, роутеру та інше

– Надає стандартну структуру папок, що покращує організацію коду

– Покращена генерація маршрутизації
– Єдиний файл конфігурації проекту
– Оптимізація для SEO
– Підвищена швидкодія при завантаженні

3.3 Аналіз технологій розробки серверної частини застосунку

3.3.1 Серверний JavaScript та середовище Node.js

Будучи найпопулярнішою мовою програмування, JavaScript також є однією з найбільш універсальних технологій розробки програмного забезпечення. Традиційно використовуваний як інструмент розробки веб-інтерфейсу, він також став основним між платформним інструментом мобільної розробки як базова технологія для великої кількості платформ, таких як Apache Cordova/PhoneGap, React Native, NativeScript, Appcelerator Titanium.

Але на цьому сфери застосування JavaScript не закінчуються. Останнім часом було багато галасу навколо використання JavaScript для програмування на стороні сервера. Одним із інструментів, які вказали на зрушення у веб-розробці, був Node.js [12].

Node.js – це середовище виконання JavaScript з відкритим вихідним кодом і є між платформним. Це популярний інструмент практично для будь-яких проектів!

Node.js запускає движок JavaScript V8, ядро Google Chrome, поза браузером. Це дозволяє Node.js бути дуже продуктивним.

Програма Node.js працює в одному процесі, не створюючи новий потік для кожного запиту. Node.js містить у своїй стандартній бібліотеці набір асинхронних примітивів введення-виводу, які запобігають блокування коду JavaScript, і загалом бібліотеки в Node.js написані з використанням неблокуючих парадигм, що робить поведінку блокування скоріше винятком, ніж нормою.

Коли Node.js виконує операцію введення-виведення, наприклад, читання з мережі, доступ до бази даних або файлової системи, замість того, щоб блокувати потік і витрачати цикли ЦП на очікування, Node.js відновить операції, коли відповідь повернеться.

Це дозволяє Node.js обробляти тисячі одночасних підключень з одним сервером, не вводячи тягар керування паралельністю потоків, що може бути значним джерелом помилок.

Node.js має унікальну перевагу, оскільки мільйони розробників інтерфейсу, які пишуть JavaScript для браузера, тепер можуть писати код на стороні сервера на додаток до коду на стороні клієнта без необхідності вивчати зовсім іншу мову [13].

У Node.js нові стандарти ECMAScript можна використовувати без проблем, оскільки вам не потрібно чекати, поки всі ваші користувачі оновлять свої браузери – ви самі вирішуєте, яку версію ECMAScript використовувати, змінюючи версію Node.js, і ви також можете ввімкнути певні експериментальні функції, запустивши Node.js з прапорцями.

3.3.2 Фреймворк NestJS

Nest (NestJS) – це платформа для створення ефективних, масштабованих додатків Node.js на стороні сервера. Він використовує прогресивний JavaScript, побудований на основі TypeScript і повністю підтримує його (проте все ще дозволяє розробникам кодувати на чистому JavaScript) і поєднує елементи OOP (об’єктно-орієнтоване програмування), FP (функціональне програмування) і FRP (функціональне реактивне програмування).

Під капотом Nest використовує надійні фреймворки HTTP-сервера, такі як Express (за замовчуванням), і за бажанням його можна також налаштувати на використання Fastify.

Nest забезпечує рівень абстракції вище цих поширених фреймворків Node.js (Express/Fastify), але також надає їхні API безпосередньо розробнику. Це дає розробникам свободу використовувати безліч модулів сторонніх розробників, які доступні для базової платформи [14].

Переваги використання NestJS:

– **Універсальність і розширюваність.** Nest.js дає розробникам максимум свободи у використанні додаткових модулів. Він забезпечує високий рівень абстракції, який дозволяє використовувати API інших фреймворків, бібліотек та іншого, збираючи з модулів унікальне серверний додаток будь-якого типу. У Nest відкритий вихідний код і практично безмежні можливості масштабування. Зокрема вже є модулі для підключення баз даних PostgreSQL, MongoDB, MySQL та інтеграції технологій Caching, Mongoose, GraphQL, WebSockets тощо.

– **Міцна основа та найкраще з нового.** Nest.js побудований на принципах Express та будь-який додаток для цього фреймворку можна використовувати і в Nest, або взагалі забути про цю можливість, якщо ці модулі вам не потрібні. Nest.js – готовий каркас MVC-додатку з коробки, що написано на TypeScript та підтримує JavaScript, а також масу рішень для них. При цьому він не обмежується стандартними функціями та дозволяє підключати всі найактуальніші JavaScript-рішення. Крім того, додатки на Nest.js дуже просто тестувати, адже при всьому різноманітті можливостей система змушує використовувати сувору архітектуру, як в Angular. Вона ж відповідає за те, що ви не зіткнетеся з величезними витратами ресурсів на масштабування додатку, коли це знадобиться – кожен мікросервіс можна допрацьовувати окремо, не зупиняючи всю систему.

– **Перспективи.** Зараз NestJS – фреймворк з найшвидшим зростанням популярності, серед розроблених для NodeJS на TypeScript. Він подобається розробникам за можливість створювати додатки з незвичайними функціями та втілювати оригінальні ідеї. Для нього вже написано багато

модулів та прикладів розв'язання задач, які є у відкритому доступі та можуть стати в пригоді в вашому проєкті. Така адаптивна екосистема і масштабованість – причини, чому ви можете вибрати Nest для свого проекту, особливо якщо це стартап або додаток з нестандартною бізнес-логікою [15].

3.4 Вибір СКБД

3.4.1 Порівняння реляційних та не реляційних баз даних

Під час вибору сучасної бази даних одним із найважливіших рішень є вибір реляційної (SQL) або нереляційної (NoSQL) структури даних. Хоча обидва варіанти є життєздатними, між ними є ключові відмінності, про які потрібно пам’ятати, приймаючи рішення [18].

П'ять критичних відмінностей між SQL і NoSQL:

– Бази даних SQL є реляційними, бази даних NoSQL нереляційними.

– Бази даних SQL використовують структуровану мову запитів і мають попередньо визначену схему. Бази даних NoSQL мають динамічні схеми для неструктурованих даних.

– Базам даних SQL притаманне вертикальне масштабування, а NoSQL – горизонтальне.

– Бази даних SQL засновані на таблицях, тоді як бази даних NoSQL є сховищами документів, ключів і значень, графіків або широких стовпців.

– Бази даних SQL краще для багаторядкових транзакцій, тоді як NoSQL краще для неструктурованих даних, таких як документи або JSON.

Таблиця 3.2 – Відмінності SQL та NoSQL БД

Структура даних	
SQL	Структура даних SQL заснована на реляційній моделі, яка нормалізує дані в строго визначених таблицях і стандартизує відносини між цими таблицями, завдяки чому бази даних SQL добре підходять для високоструктурованих даних.

Продовження таблиці 3.2

NoSQL	Структура даних NoSQL не вимагає нормалізованої конфігурації або дотримується реляційної моделі, але натомість є достатньо гнучкою, щоб вмістити різні моделі, включаючи ключ-значення, документ, орієнтований на стовпець і графік.
Мова	
SQL	Бази даних SQL пов’язані з мовою SQL. Деякі продукти реляційних баз даних підтримують чистий SQL, але багато з них включають розширені версії мови — наприклад, Transact-SQL (T-SQL) SQL Server — для розміщення функцій, характерних для продукту. Проте всі бази даних SQL підтримують основні елементи мови ANSI/ISO.
NoSQL	Бази даних NoSQL не заблоковані однією мовою. Використовувана мова залежить від типу бази даних NoSQL, окремої реалізації та конкретної операції. Наприклад, MongoDB зберігає всі документи у форматі JSON із запитами на основі мови програмування JavaScript.
Схеми	
SQL	База даних SQL вимагає попередньо визначеної схеми, яка визначає, як налаштовуються таблиці та зберігаються дані, в результаті чого утворюється жорстка структура, яка допомагає оптимізувати зберігання та забезпечити цілісність даних, але обмежує гнучкість.
NoSQL	База даних NoSQL використовує динамічну схему, яка не вимагає попередньо визначеної структури даних, що забезпечує високий ступінь гнучкості, наприклад можливість додавати документи з різними полями до однієї бази даних.

Продовження таблиці 3.2

Цілісність даних	
SQL	Бази даних SQL забезпечують високий ступінь цілісності даних, дотримуючись принципів атомарності, узгодженості, ізоляції та довговічності (ACID), які є важливими для підтримки робочих навантажень, таких як фінансові транзакції.
NoSQL	Базам даних NoSQL може бути важко забезпечити такий самий рівень цілісності даних, як і бази даних SQL, при цьому більшість дотримується принципів BASE (базова доступність, м’який стан і остаточна узгодженість), що означає, що дані в розподіленому середовищі можуть бути тимчасово неузгодженими.
Масштабованість	
SQL	Бази даних SQL в основному масштабуються вертикально, що означає, що їх можна легко збільшити, додавши ресурси, такі як процесори або пам’ять, але бази даних SQL не дуже ефективні при горизонтальному масштабуванні, що робить їх погано придатними для великих розподілених наборів даних.
NoSQL	Бази даних NoSQL можуть дуже ефективно горизонтально масштабуватися в системах і місцях, що дає змогу розміщувати великі сховища розподілених даних, підтримуючи збільшений рівень трафіку.
Запити	
SQL	Бази даних SQL ефективно обробляють запити та об’єднують дані між таблицями, що спрощує виконання складних запитів щодо структурованих даних, включаючи спеціальні запити.
NoSQL	Базам даних NoSQL бракує узгодженості між продуктами і, як правило, вимагають більше роботи для запиту даних, зокрема, у міру збільшення складності запиту.

Кінець таблиці 3.2

Завершеність	
SQL	Бази даних SQL створені на основі зрілих технологій, які добре відомі та підтримуються великими спільнотами розробників.
NoSQL	Продукти NoSQL не настільки зрілі, а технології не так добре підтримуються, як продукти SQL, але технології NoSQL швидко проникають у галузь, а спільноти розробників постійно зростають.

На основі аналізу відмінностей та особливостей SQL та NoSQL баз даних було вирішено використовувати саме реляційну модель. Такий вибір було зроблено враховуючи великий рівень забезпечення цілісності даних при використанні реляційної БД, наявності чіткої схеми, що не передбачає серйозного масштабування, та наявність переваг щодо використання складних запитів під час роботи з базою.

3.4.2 Аналіз реляційних СКБД

Система управління базами даних або СКБД – це тип програмного забезпечення, яке взаємодіє з самою базою даних, програмами та інтерфейсами користувача для отримання даних та їх аналізу. СКБД також містить ключові інструменти для керування базою даних [17].

До найвідоміших реляційних СКБД можна віднести наступні: MySQL, MariaDB, Oracle, PostgreSQL, MSSQL. На рис. 3.4 зображено графік популярності СКБД.

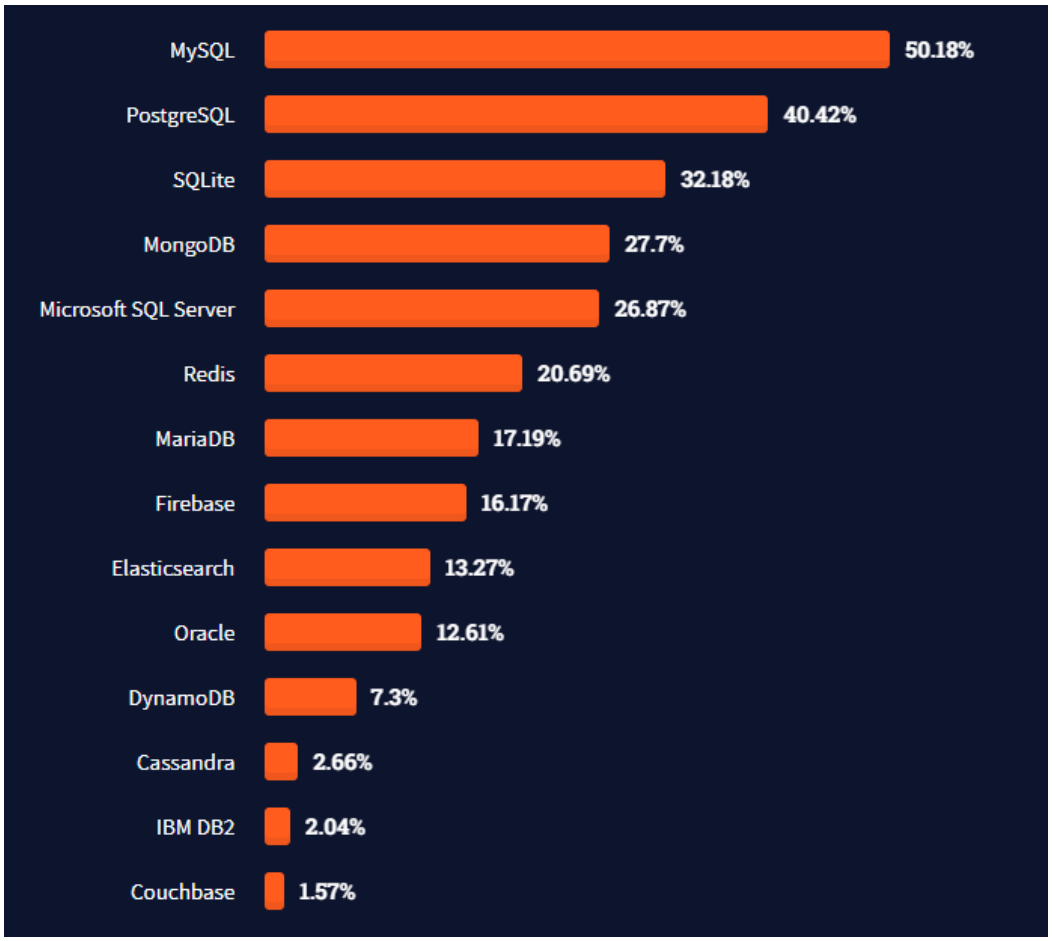

Рисунок 3.4 – Графік популярності СКБД

MySQL. Це одна з найпопулярніших систем реляційних баз даних. Спочатку рішення з відкритим кодом MySQL тепер належить Oracle Corporation. Сьогодні MySQL є опорою прикладного програмного забезпечення LAMP. Це означає, що він є частиною стеку Linux, Apache, MySQL та Perl/PHP/Python. Маючи під капотом C і C++, MySQL добре працює з такими системними платформами, як Windows, Linux, MacOS, IRIX та іншими.

MariaDB. MariaDB, форк з відкритим кодом від MySQL, має комерційну підтримку. Він працює під загальною загальнодоступною ліцензією GNU і має команди, API та бібліотеки, подібні до MySQL.

Oracle. Oracle – це система керування реляційними базами даних, створена та керована корпорацією Oracle. Наразі він підтримує декілька моделей даних, таких як документ, графік, реляційний та ключ-значення в

одній базі даних. У своїх останніх випусках вона переорієнтована на хмарні обчислення. Ліцензування СКБД Oracle є повністю запатентованою, і доступні як безкоштовні, так і платні варіанти.

PostgreSQL. Ця система управління базами даних поділяє свою популярність з MySQL. Це об’єктно-реляційна СКБД, де визначені користувачем об’єкти та підходи до таблиць поєднуються для побудови більш складних структур даних. Крім того, PostgreSQL має багато подібності з MySQL. Він спрямований на посилення стандартів відповідності та розширюваності. Отже, він може обробляти будь-яке робоче навантаження, як для одно машинних продуктів, так і для складних додатків. Власник і розробник PostgreSQL Global Development Group. Вона все ще залишається повністю відкритою. Ця СКБД доступна для використання з такими платформними системами, як Microsoft, iOS, Android та багатьма іншими.

MSSQL. Як повністю комерційний інструмент, Microsoft SQL Server є однією з найпопулярніших реляційних СКБД, на додаток до MySQL, PostgreSQL та Oracle. Він добре справляється з ефективним зберіганням, зміною та керуванням реляційними даними. Для взаємодії з базами даних SQL Server інженери БД зазвичай використовують мову Transact-SQL (T-SQL), яка є розширенням стандарту SQL.

Розглянемо основні переваги та недоліки перерахованих вище реляційних СУБД, які наведені у таблиці 3.3.

Таблиця 3.3 – Переваги та недоліки СКБД

СКБД	**Переваги**	**Недоліки**
MySQL	Безкоштовне встановлення	Проблеми масштабованості
	Простий синтаксис і помірна складність	Часткова підтримка з відкритим кодом
	Хмарна сумісність	Обмежена відповідність стандартам SQL
MariaDB	Шифрування	Спільнота все ще зростає

Кінець таблиці 3.3

	Широка функціональність	Розриви між версіями оновлення MySQL та MariaDB
	Висока продуктивність	
Oracle	Інновації для щоденного робочого процесу	Висока вартість
	Потужна технічна підтримка та документація	Ресурсомістка технологія
	Велика місткість	Важка крива навчання
PostgreSQL	Велика масштабованість	Непослідовна документація
	Підтримка спеціальних типів даних	Відсутність інструментів звітності та аудиту
	Легко інтегровані сторонні інструменти	
	Підтримка з відкритим кодом і громадою	
MSSQL	Різноманітність версій	Висока вартість
	Наскрізне рішення для бізнес-даних	Нечіткі та плаваючі умови ліцензії
	Багата документація та допомога громади	Складний процес налаштування
	Підтримка хмарних баз даних	

На основі аналізу переваг та недоліків СКБД було вирішено використовувати MySQL. Адже вона є безкоштовною, простою і при цьому відповідає усім вимогам розроблюваної системи.

Висновки до розділу 3

В даному розділі були проаналізовані основні засоби розробки майбутньої системи та визначено основний стек технологій.

Для розробки представлення застосунку було вирішено використовувати сучасний і потужний фреймворк Nuxt.js, який базується на Vue.js. Головними факторами використання даної технології була підтримка SSR та швидкість і зручність розробки.

Backend програмного забезпечення було вирішено писати з використанням фреймворку NestJS. Основними перевагами якого є надання чіткої структури застосунку та підтримка TypeScript, що забезпечує високу надійність роботи застосунку.

Для роботи з базою даних було обрано популярне рішення – СКБД MySQL. Вона є простою і відповідає усім вимогам до розробки програмного продукту.

4 РОЗРОБКА ЗАСТОСУНКУ

4.1 Робота з базою даних

4.1.1 Огляд ORM Sequelize

При розробці програмного забезпечення використовуючи ООП, кодування представляє собою процес маніпуляції даними у вигляді об’єктів. Проблема полягає в тому, що база даних представляє дані у реляційній моделі. Для того, щоб синхронізувати об’єктну та реляційну модель даних, та надати розробнику зручний спосіб роботи з БД використовують ORM.

Об’єктно-реляційне відображення (ORM) – це механізм, який дає змогу звертатися до об’єктів, отримувати доступ та маніпулювати ними, не враховуючи, як ці об’єкти пов’язані з їхніми джерелами даних. ORM дозволяє програмістам підтримувати узгоджене уявлення про об’єкти з часом, навіть якщо джерела, які їх доставляють, приймачі, які їх отримують, і програми, які звертаються до них, змінюються [19].

Для розробки під Node.js існує багато ORM, таких як, TypeORM, MikroORM, Sequelize, Prisma та інші. У даному проекті було вирішено використовувати дуже популярну ORM Sequelize.

Sequelize – це простий у використанні і заснований на промісах інструмент ORM Node.js для Postgres, MySQL, MariaDB, SQLite, DB2, Microsoft SQL Server і Snowflake. Він має надійну підтримку транзакцій, відносини, швидке й ліниве завантаження, реплікацію читання тощо [20].

Щоб мати змогу використовувати Sequelize у NestJS проекті необхідно завантажити npm пакети sequelize, sequelize-typescript, mysql2 та @types/sequelize за допомогою команди npm install.

Для роботи з базою даних було створено окремий модуль у проекті. В ньому знаходяться класи міграцій, сидів, конфігурація підключення до БД та провайдер, за допомогою якого створюється інстанс sequelize. Код провайдера зображено на рис. 4.1.

```
export const databaseProviders = [
  {
    provide: 'SEQUELIZE',
    useFactory: async () => {
      const sequelize = new Sequelize( options: {
        dialect: 'mysql',
        host: process.env.DB_HOST,
        port: parseInt(process.env.DB_PORT),
        username: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_NAME,
        logging: false
      });
      sequelize.addModels( models: [
        Country,
        Role,
        Source,
        Identifier,
        User,
        Stock,
        Product,
        Comment,
        Tracker,
        Subscription,
        Device,
        Page,
        Content,
        Group,
        Token
      ]);
      await sequelize.sync();
      return sequelize;
    }
  }
]
```

Рисунок 4.1 – Код провайдеру бази даних

При створенні екземпляру sequelize вказується конфігурація підключення до БД та усі моделі які будуть використовуватися в проекті.

4.1.2 Моделі та запити у Sequelize

Моделі – це сутності Sequelize. Модель – це абстракція, яка представляє таблицю у базі даних. У Sequelize це клас, який розширює Model.

Модель повідомляє Sequelize кілька речей про сутність, яку вона представляє, наприклад, ім’я таблиці в базі даних і стовпці, які вона має (і їхні типи даних). Модель у Sequelize має назву. Це ім’я не обов’язково збігається з назвою таблиці, яку вона представляє в базі даних. Зазвичай моделі мають однині імена (наприклад, User), а таблиці мають множину (наприклад, users), хоча це можна повністю налаштувати.

На рис. 4.2 зображено клас моделі Role, яка представляє таблицю roles у базі даних.

```
@Table( options: {
  tableName: 'roles'
})
export class Role extends Model {
  @Column( options: {
    type: DataType.BIGINT,
    allowNull: false,
    autoIncrement: true,
    unique: true,
    primaryKey: true
  })
  public id: number

  @Column( options: {
    type: DataType.STRING,
    allowNull: false,
    unique: true
  })
  name: string

  @CreatedAt public createdAt: Date

  @UpdatedAt public updatedAt: Date

  @HasMany( associatedClassGetter: () => User)
  users: User[]
}
```

Рисунок 4.2 – Модель Role

Як можна побачити на рисунку у класі моделі вказуються назва таблиці, поля, що відповідають таблиці у базі даних та залежності з іншими сутностями.

Надалі класи моделі можуть використовуватися для виконання запитів до бази даних. Це можуть бути як прості так і досить складні запити з

великою кількістю приєднаних залежностей, умов, впорядкуванням та іншими можливостями SQL. Для цього ORM Sequelize впроваджує інтерфейс для роботи з CRUD операціями.

На рис. 4.3 відображається приклад запиту який відповідає за вибірку інформації про товари з врахуванням країни, обмеженням на кількість вибраних рядків та впорядкуванням за датою створення.

```
return await this.productRepository.findAll<Product>( options: {
  include: { model: Country, as: 'country', attributes: ['id', 'code'] },
  limit: limit,
  where: { country_id: countryId, show: true },
  order: [ [ 'createdAt', 'DESC' ] ]
})
```

Рисунок 4.3 – Приклад запиту до БД

Sequelize надає широкі можливості для створення складних запитів до бази даних, але якщо цього видається замало, то завжди можна скористатися написанням запиту за допомогою чистого SQL.

4.1.3 Міграції та сіди

Так само, як використання системи контролю версій, такої як Git, для керування змінами у вихідному коді, можна використовувати міграції для відстеження змін у базі даних. За допомогою міграцій можна перевести наявну базу даних в інший стан і навпаки: ці переходи стану зберігаються у файлах міграції, які описують, як перейти до нового стану та як повернути зміни, щоб повернутися до старого стану.

Для цього знадобиться інтерфейс командного рядка Sequelize (CLI). CLI забезпечує підтримку міграції та завантаження проекту. Міграція в Sequelize – це файл javascript, який експортує дві функції, up та down, які визначають, як виконати міграцію та скасувати її. Функції визначаються вручну, але не викликаються вручну, вони будуть автоматично викликані CLI. У цих функціях необхідно просто виконувати будь-які запити, які

потрібні, за допомогою sequelize.query та будь-яких інших методів, які надає Sequelize. Немає додаткової магії, крім цього.

На рис. 4.4 зображено код міграція для таблиці countries.

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    return queryInterface.createTable( tableName: 'countries', attributes: {
      id: {
        type: Sequelize.BIGINT,
        allowNull: false,
        autoIncrement: true,
        unique: true,
        primaryKey: true
      },
      name: {
        type: Sequelize.STRING,
        allowNull: false,
        unique: true
      },
      code: {
        type: Sequelize.STRING(2),
        allowNull: false,
        unique: true
      },
      createdAt: {
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal( val: 'CURRENT_TIMESTAMP')
      },
      updatedAt: {
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal( val: 'CURRENT_TIMESTAMP')
      }
    })
  },

  down: async (queryInterface, Sequelize) => {
    return queryInterface.dropTable( tableName: 'countries');
  }
};
```

Рис. 4.4 – Приклад міграції

Зазвичай буває необхідно вставити до таблиць бази даних якісь дані за замовчуванням або тестові дані. Для цього можна використати механізм сидів. За допомогою цього інструменту можна легко заповнювати таблицю даними та відмінити вставку за необхідністю.

На рис. 4.5 зображено код сіда, який заповнює даними таблицю identifiers.

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert( tableName: 'identifiers', records: [{
        id: 1,
        name: 'code',
        createdAt: new Date(),
        updatedAt: new Date()
      }, {
        id: 2,
        name: 'url',
        createdAt: new Date(),
        updatedAt: new Date()
      }
    ])
  },

  down: async (queryInterface, Sequelize) => {
    return queryInterface.bulkDelete( tableName: 'identifiers', identifier: null, options: {})
  }
};
```

Рисунок 4.5 – Приклад сіда

За допомогою цих інструментів можна швидко та ефективно працювати з базою даних.

4.2 Розробка візуальної частини застосунку

Візуальна частина застосунку представляє собою SPA розроблений за допомогою технології Nuxt.js. Інтерфейс є повністю адаптивним для всіх сучасних пристроїв та браузерів.

Далі наведено опис основних графічних елементів які були створені для користувача сайту та адміністратора.

– **Головна сторінка.** На цій сторінці(рис. 4.6) знаходяться усі товари які представлені на сайті. Оскільки їх кількість невелика, то вони чудово розміщуються на ній без шкоди для дизайну. Також, на сторінці наявний список відстежуваних магазинів і рядок для пошуку товарів по назві.

– **Хедер.** У хедері сайту відображається логотип компанії, перемикач між країнами та кнопки переходу до реєстрації та авторизації.

– **Футер.** У футері також знаходиться логотип, навігація по сторінкам та кнопка виклику контактної форми.

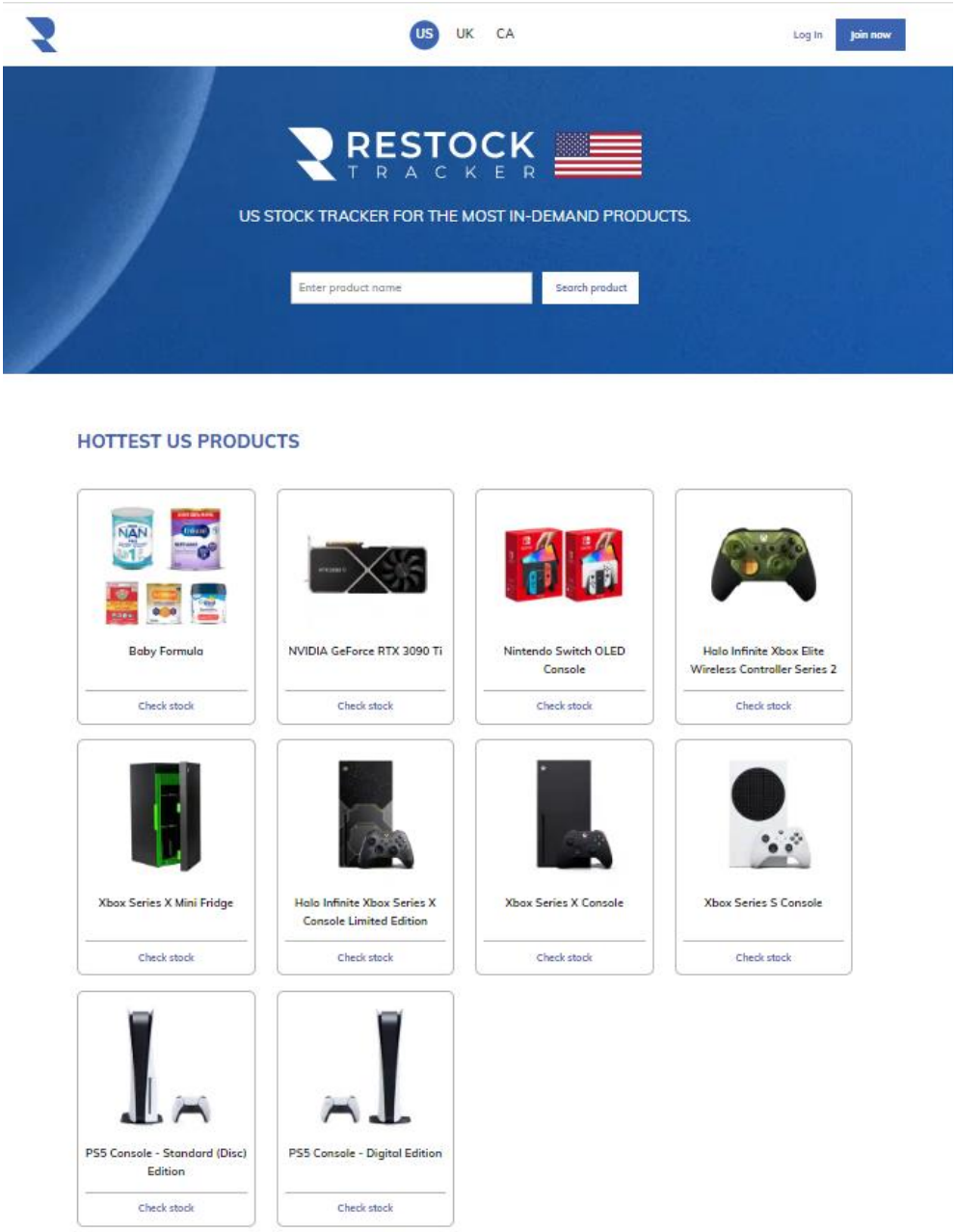

Рисунок 4.6 – Головна сторінка сайту

– **Сторінка товару.** На цій сторінці(рис. 4.7) можна побачити детальну інформацію про товар з зображенням товару. Важливим елементом на сторінці є таблиця яка відображає наявність, назву та ціну товару у відстежуваних магазинах. Також, наявна кнопка Subscribe to Instant Stock

Alerts, функція якої підписати користувача на повідомлення про надходження товару на склад магазину.

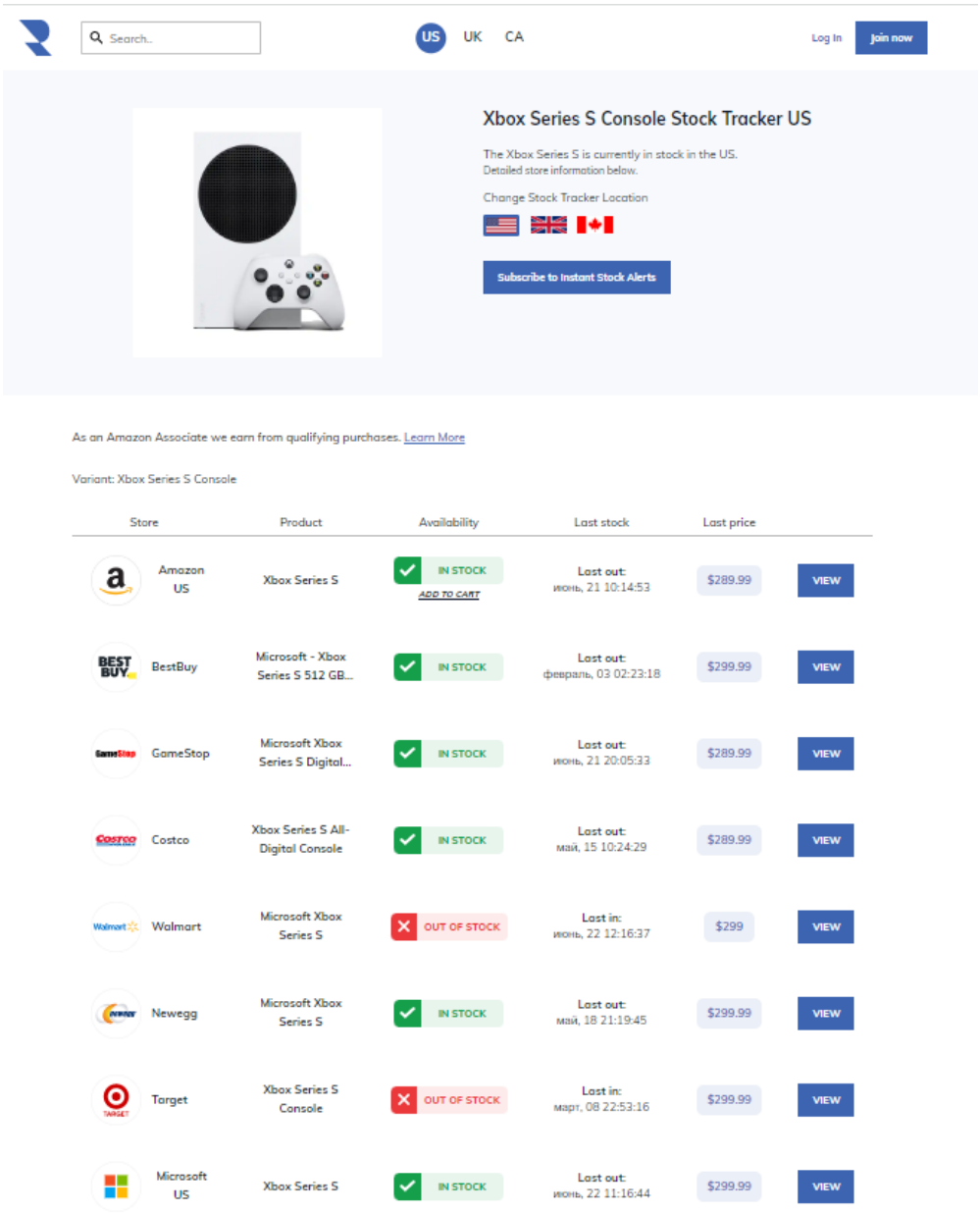

Рисунок 4.7 – Сторінка товару

– **Форма авторизації.** Необхідна для авторизації користувача у застосунку. На вибір існує 2 методи авторизації – через пароль та емейл, або за допомогою Google OAuth. Форма зображена на рис. 4.8.

– **Форма реєстрації.** Схожа на форму авторизації. Для реєстрації користувачу необхідно заповнити 4 поля: Ім’я користувача, емейл, пароль та

поле підтвердження паролю, натиснути чекбокс з підтвердженням можливості відправки повідомлень при підписці на товар.

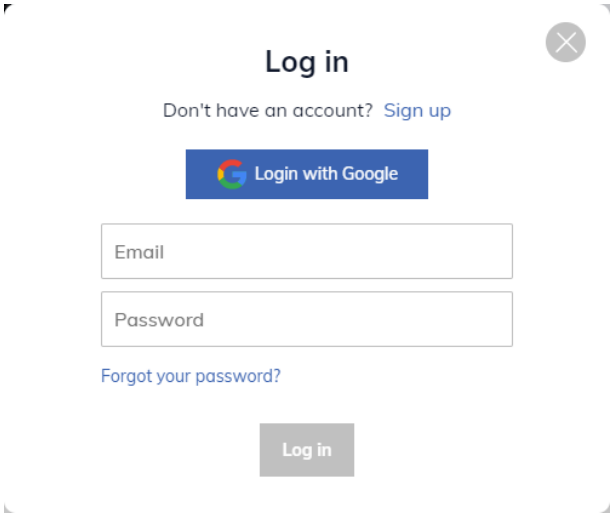

Рисунок 4.8 – Форма авторизації

– **Персональний кабінет.** У персональному кабінеті користувача (рис. 4.9) можна змінити аватар, ім’я користувача, емейл та пароль, а також, видалити аккаунт з сервісу.

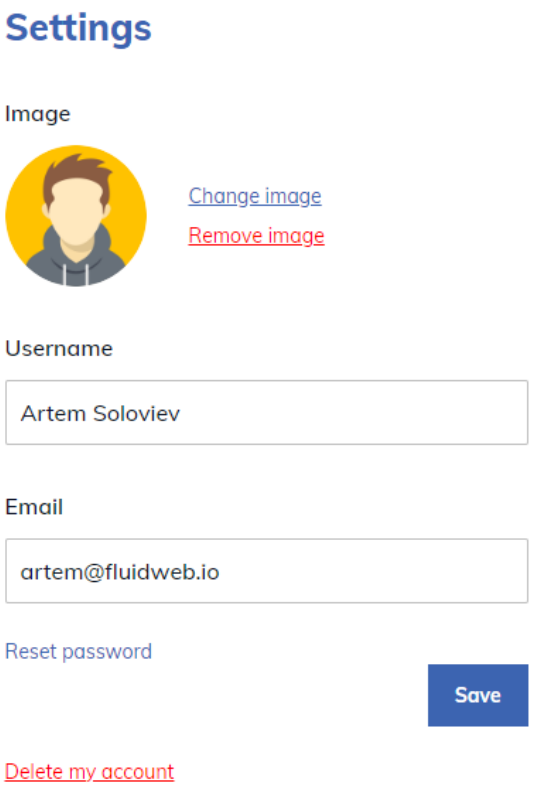

Рисунок 4.9 – Персональний кабінет

– **Сторінка товарів (адміністратор).** На цій сторінці(рис. 4.10) відображається перелік усіх продуктів які є на сервісі. Їх можна сортувати та фільтрувати.

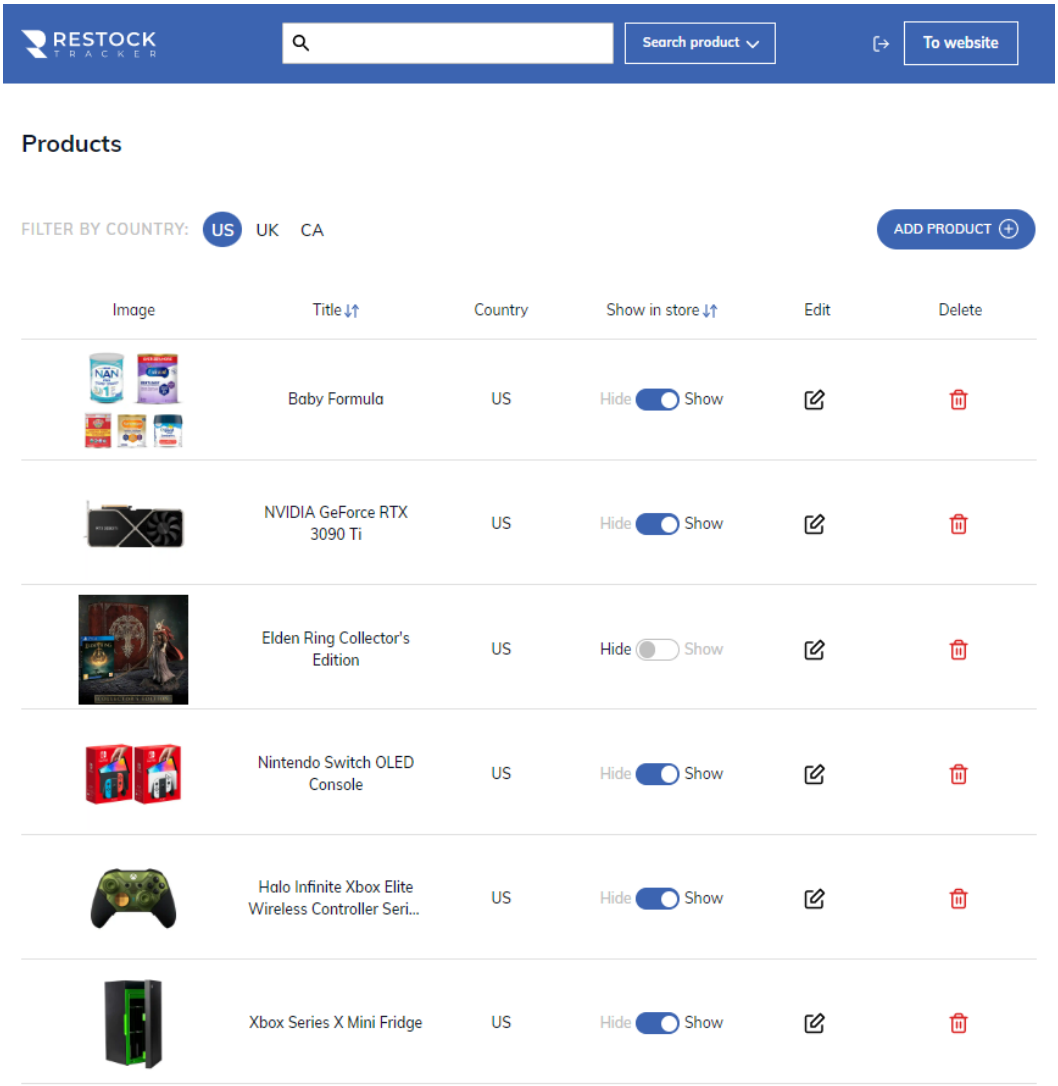

Рисунок 4.10 – Сторінка товарів

– **Сторінка товару (адміністратор).** На цій сторінці адміністратор має змогу редагувати інформацію про товар, а також підключати до процесу стеження за товаром магазини.

– **Сторінка користувачів (адміністратор).** Тут відображається список усіх користувачів які зареєстровані на сервісі. Їх можна редагувати та видаляти.

– **Сторінка користувача (адміністратор).** На цій сторінці адміністратор має змогу переглянути та відредагувати дані про користувача,

а також, переглянути усі його підписки на товари, і при необхідності, завершити підписку.

– **Сторінка магазинів (адміністратор).** На цій сторінці (рис. 4.10) можна побачити список усіх відстежуваних магазинів з інформацією про процес відстеження.

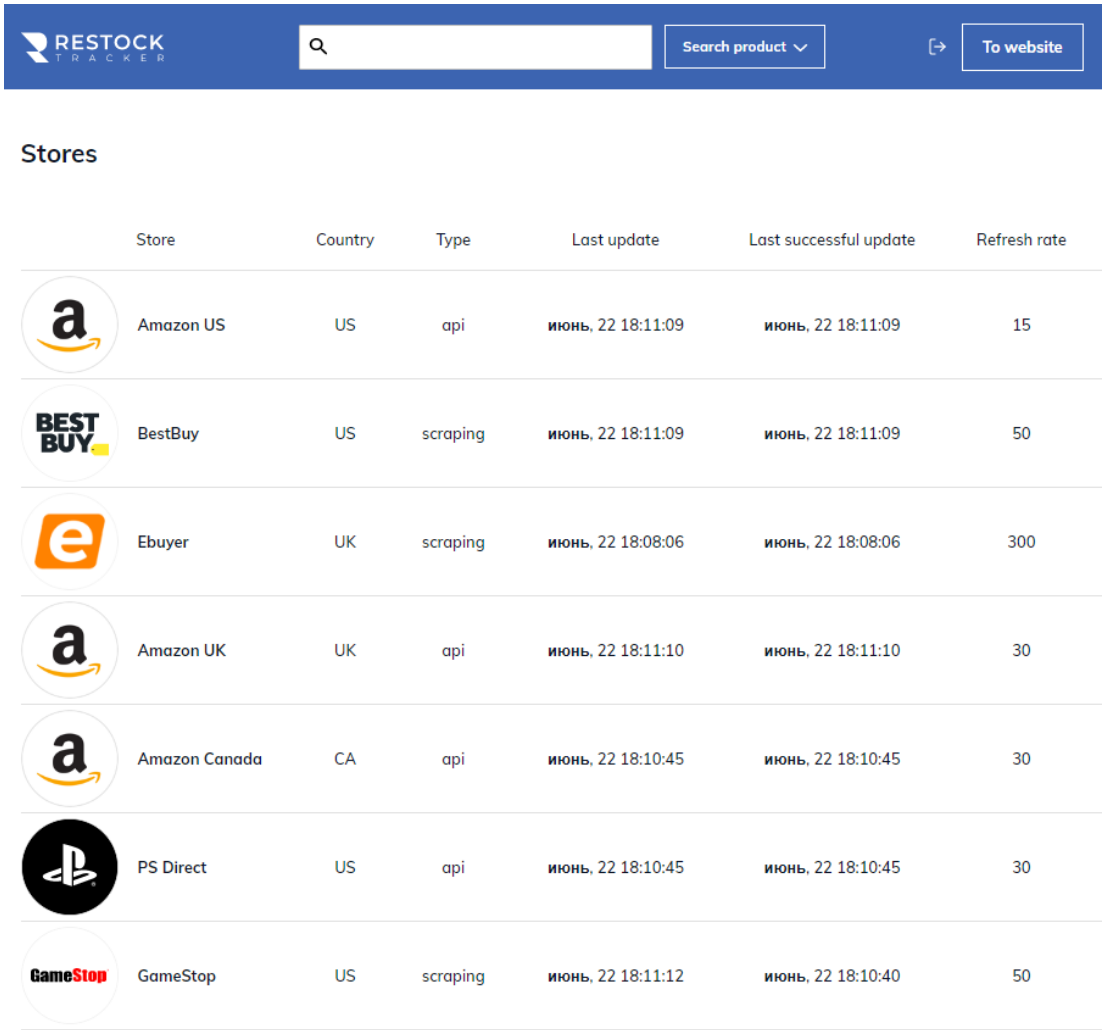

Рисунок 4.11 – Сторінка магазинів

– **Сторінка коментарів(адміністратор).** Представлена полем у яке можна занести список заборонених слів і словосполучень, які не можна використовувати користувачам при написанні коментарів.

– **Сторінка контенту(адміністратор).** На цій сторінці адміністратор має можливість змінювати контент на сторінках враховуючи обрану країну.

4.3 Розробка серверної частини застосунку

4.3.1 Розробка API на базі NestJS

За документацією NestJS пропонує модульну архітектуру, за якої застосунок розбивається на модулі, кожен з яких має свою зону відповідальності. При такому підході проект буде добре структурованим та в ньому буде легко знайти необхідні файли з кодом.

Для роботи у режимі типового API необхідно розібратися як функціонує подібний застосунок при використанні даного фреймворку. Далі наведемо основні структурні одиниці програми на NestJS:

– **Контролери.** Контролери відповідають за обробку вхідних запитів і повернення відповідей клієнту. Мета контролера — отримувати конкретні запити до програми. Механізм маршрутизації контролює, який контролер отримує які запити. Часто кожен контролер має більше одного маршруту, і різні маршрути можуть виконувати різні дії [14].

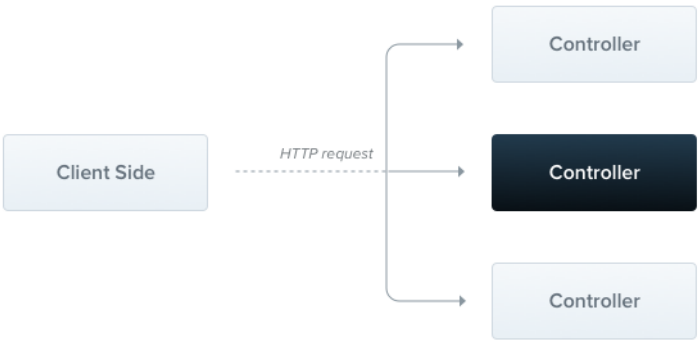

Рисунок 4.12 – Принцип роботи контролера

– **Провайдери.** Провайдери є основним поняттям у Nest. Багато базових класів Nest можуть розглядатися як провайдери – сервіси, репозиторії, фабрики, помічники тощо. Основна ідея провайдера полягає в тому, що його можна ввести як залежність; це означає, що об'єкти можуть створювати різні відносини один з одним, а функцію "підключення" екземплярів об'єктів можна значною мірою делегувати системі середовища виконання Nest.

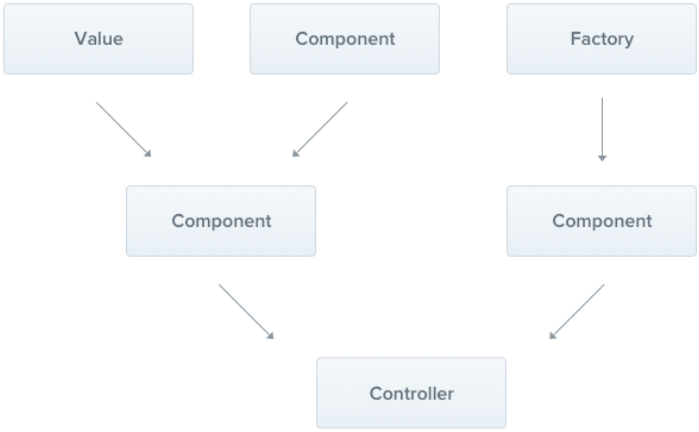

Рисунок 4.13 – Взаємодія провайдерів

– **Модулі.** Модуль — це клас, анотований декоратором @Module(). Декоратор @Module() надає метадані, які Nest використовує для організації структури програми.

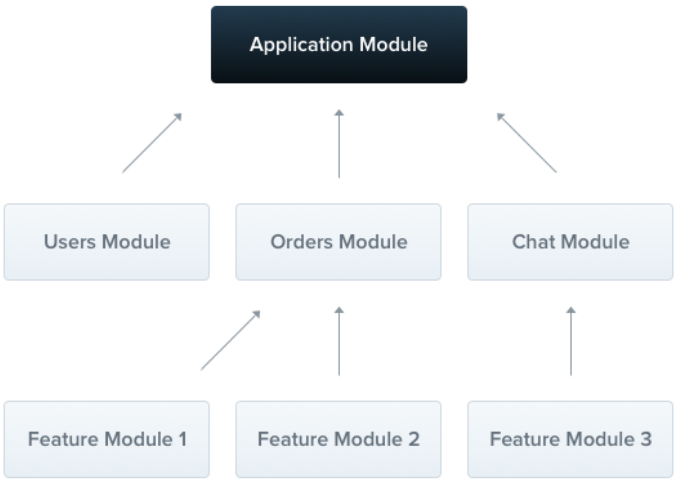

Рисунок 4.14 – Структура модулів

– **Проміжне програмне забезпечення (Middleware).** Проміжне програмне забезпечення – це функція, яка викликається перед обробником маршруту. Функції проміжного програмного забезпечення мають доступ до об’єктів запиту та відповіді, а також до функції проміжного програмного забезпечення next() у циклі запит-відповідь програми. Наступна функція проміжного програмного забезпечення зазвичай позначається змінною з іменем next.

Рисунок 4.15 – Принцип роботи middleware

– **Охоронці (Guards).** Охоронці мають єдину відповідальність. Вони визначають, чи буде даний запит оброблений обробником маршруту чи ні, залежно від певних умов (наприклад, дозволів, ролей, списків керування доступом тощо), присутніх під час виконання. Це часто називають авторизацією.

Рисунок 4.16 – Принцип роботи охоронців

Розглянемо приклад обробки одного з запитів, що може надійти до API застосунку. Для прикладу візьмемо запит на видалення трекеру.

Щоб його здійснити необхідно буде відправити DELETE запит за наступним маршрутом – /api/trackers/:id.

Спочатку запит оброблюється маршрутизатором NestJS, щоб визначити який метод якого контролеру необхідно викликати. Після чого система передає виконання процесу контролеру TrackersController та його методу remove (рис. 4.17).

```
@Delete( path: ':id')
@UseGuards(AuthGuard(), AdminGuard)
async remove(@Param( property: 'id') id: string) {
  return await this.trackersService.remove(+id);
}
```

Рисунок 4.17 – Метод remove у контролері

Але як бачимо у даному маршруті використовується 2 охоронці – AuthGuard та AdminGuard. Перший перевіряє чи авторизований користувач у системі, а другий – чи є користувач адміністратором. Якщо ці умови не будуть виконані система поверне відповідь з помилкою.

Якщо ж усе добре, то далі відпрацює метод remove (рис. 4.18) у сервісі TrackersService. Задача цього класу просто розгрузити контролер, щоб у ньому не було багато зайвого коду.

```
async remove(id: number) {
  return this.trackerRepository.destroy( options: {
    where: { id: id }
  });
}
```

Рисунок 4.18 – Метод remove у сервісі

Метод remove у свою чергу викликає функцію видалення запису з бази даних у моделі Sequelize.

Після чого до клієнта повертається відповідь з інформацією про виконаний запит.

4.3.2 Розробка сервісу стеження за наявністю товарів

Відповідно до вимог необхідно було розробити сервіс який має стежити за наявністю товарів у інтернет-магазинах.

Спочатку необхідно розділити усі магазини на ті, що спостерігаються за допомогою звернень до API та на ті, що будуть використовувати для цього парсинг веб-сторінок. Це потрібно зробити, адже принцип роботи у них буде відрізатися.

Після цього створюємо інтерфейси для обох випадків стеження – ApiInterface та ScraperInterface. Це необхідно для того, щоб чітко визначити які методи повинні використовувати скрапери та наддасть нам переваги поліморфізму. На наступних рисунках наведено код інтерфейсів.

```
export interface ApiInterface {
  stock: Stock;
  getData();
  sendApiRequest(): Promise<any>;
}
```

Рисунок 4.19 – Інтерфейс ApiInterface

```
export interface ScraperInterface {
  stock: Stock;
  getData();
  scrap(tracker: Tracker);
  parsePage(body);
}
```

Рисунок 4.20 – Інтерфейс ScraperInterface

Усі класи які будуть створені для скрапінгу магазинів будуть реалізовувати один з цих інтерфейсів.

Як ми бачимо, обидва інтерфейси мають метод getData. Саме він і буде викликатися для запуску ітерації стеження за товарами у магазині. Для того щоб він викликався автоматично через визначений період часу будемо використовувати cron.

Cron дозволяє користувачам Linux і Unix запускати команди або сценарії в певну дату і час. Ви можете запланувати періодичне виконання сценаріїв. Cron є одним із найкорисніших інструментів у операційних системах Linux або UNIX.

Фреймворк NestJS має вбудований функціонал для роботи з cron. Він називається Планування завдань (Task scheduling).

Планування завдань дозволяє запланувати виконання довільного коду (методів/функцій) у фіксовану дату/час, через повторювані інтервали або один раз через певний інтервал. У світі Linux це часто вирішується за допомогою таких пакетів, як cron, на рівні ОС. Для програм Node.js існує кілька пакетів, які емулюють функціональні можливості, схожі на cron. Nest

надає пакет @nestjs/schedule, який інтегрується з популярним пакетом Node.js node-cron.

Для початку необхідно створити метод який буде відпрацьовувати кожного разу при запуску cron. Для запуску скраперів які використовують API таким методом буде api (рис. 4.21) класу CronService.

```
@Cron(CronExpression.EVERY_SECOND)
async api() {
    const isScraper = this.configService.get<number>('scraper');

    if (isScraper != 1) {
        return;
    }

    await this.apiProducerService.run( className: "AmazonApi");
    await this.apiProducerService.run( className: "BestbuyApi");
    await this.apiProducerService.run( className: "AmazonUkApi");
    await this.apiProducerService.run( className: "AmazonCaApi");
    await this.apiProducerService.run( className: "PsDirectApi");
    await this.apiProducerService.run( className: "WalmartApi");
    await this.apiProducerService.run( className: "NeweggApi");
    await this.apiProducerService.run( className: "NeweggCaApi");
}
```

Рисунок 4.21 – Метод запуску скраперів на виконання

Хоч cron і запускаєтья кожну секунду, для кожного скраперу є свій власний таймер, який відраховує час до запуску. Це було зроблено для того, щоб мати можливість виставляти різний час скрапінгу для кожного магазину.

Щоб система відпрацьовувала коректно необхідно було впровадити механізм черг. На щастя він також є у Nest і реалізується за допомогою Redis.

Redis – це сховище з відкритим вихідним кодом (ліцензія BSD), сховище структури даних у пам’яті, яке використовується як база даних, кеш-пам’ять, посередник повідомлень і потокова система. Redis надає такі структури даних, як рядки, хеші, списки, набори, відсортовані набори із запитами діапазону, растрові зображення, гіперлогові журнали, геопросторові індекси та потоки. За допомогою Redis можна організувати черги [22].

Черги – це потужний шаблон проектування, який допоможе впоратися зі стандартними проблемами масштабування додатків і продуктивності. Для використання черг викликів api скраперів необхідно створити класи ApiConsumer та ApiProducer. Код цих класів наведено нище.

```
@Processor( queueName: 'api-fetchers')
export class ApiConsumer {

    constructor(private apiService: ApiService) {
    }

    @Process( name: 'run-api-fetcher')
    async readOperationJob(job:Job<unknown>){
      await this.apiService.run(job.data);
    }
}
```

Рисунок 4.22 – Клас ApiConsumer

```
@Injectable()
export class ApiProducerService {
    constructor(@InjectQueue( name: 'api-fetchers') private queue: Queue) {}

    async run(className){
        await this.queue.add('run-api-fetcher',{
            api: className
        }, {
            delay: 0
        });
    }
}
```

Рисунок 4.23 – Клас ApiProducer

Далі необхідно вказати в конфігурації дані серверу Redis і викликати метод run для кожного скраперу при запуску методу api.

При роботі з API магазину все відбувається досить легко. Необхідно просто зробити запит та розібрати дані, які прийшли у відповідь.

А от процес парсингу може бути досить складним. Перша проблема з якою можна зустрітися – це блокування. Деякі магазини мають систему захисту від подібних дій, тому необхідно було знайти спосіб обійти цей захист.

Найпростіший спосіб – використання проксі сервісу з ротацією ip адрес. Після аналізу низки подібних сервісів було вирішено використати Zenscrape. Він надає зручний інтерфейс для скрапінгу веб-сторінок.

Щоб отримати дані необхідно зробити запит на API сервісу(рис. 4.24) передавши у параметрі адресу сторінки. Якщо цього буде замало для отримання бажаного результату можна погратися з параметрами і додати затримку перед парсингом, рендер js, або premium режим.

```
this.httpService.get( url: `https://app.zenscrape.com/api/v1/get?&url=${tracker.product_url}`, config: {
  headers: {
    'apikey': this.configService.get<string>('zenscrape.apiKey')
  }
}).pipe(take( count: 1)) Observable<AxiosResponse<any>>
```

Рисунок 4.24 – Приклад запиту до Zenscrape

Інша проблема – це коли недостатньо просто завантажити сторінку, але необхідно виконати на ній якісь дії, щоб відобразити той контент який нас цікавить. Наприклад, натиснути кнопку, закрити popup, обрати країну та інше.

У цьому випадку можна скористуватися бібліотекою puppeteer. Puppeteer — це бібліотека Node, яка надає високорівневий API для керування Chrome або Chromium через DevTools Protocol. За замовчуванням Puppeteer працює у режимі headless, але його можна налаштувати на повний (non-headless) Chrome або Chromium.

За допомогою puppeteer ми можемо виконувати різні маніпуляції на сторінці після чого отримувати необхідний контент.

Останнім, але не менш важливим етапом є відправка повідомлень користувачам про те, що товар з’явився на складі. Повідомлення мають бути двух типів – емейл та браузерні.

Для першого типу було вирішено використовувати сервіс Postmark. Він надає зручний інтерфейс для відправки великої кількості повідомлень з великою швидкістю.

Для відправки браузерних повідомлень було обрано FCM. Firebase Cloud Messaging (FCM) – це кросплатформне рішення для обміну повідомленнями, яке дозволяє надійно та безкоштовно надсилати браузерні повідомлення [23].

4.4 Розгортання застосунку на сервері

Однією з найважливіших вимог до системи була відмовостійкість при напливах трафіку під час дропу товару. Під час великого завантаження потужності сервера могли не витримати і почати повільно працювати та повертати користувачам 504 помилку (Gateway timeout) або зовсім падати повертаючи 502 (Bad Gateway).

Для вирішення цієї проблеми необхідно було скористатися надійними та перевіреними рішеннями для розгортання застосунку на сервері. Відомою компанією яка може надавати такий сервіс є DigitalOcean.

DigitalOcean, приватна компанія зі штаб-квартирою в Нью-Йорку, забезпечує відносно просту хмарну платформу для розгортання, управління та масштабування програм будь-якого розміру, усуваючи тертя інфраструктури та забезпечуючи передбачуваність, щоб розробники та їхні команди могли витрачати більше часу на створення програмного забезпечення. які люблять клієнти. DigitalOcean має 12 центрів обробки даних по всьому світу в 8 регіонах. Очолює компанію генеральний директор Марк Темплтон [24].

Основним продуктом даної компанії є дроплети. DigitalOcean Droplets – це віртуальні машини (VM) на базі Linux, які працюють поверх віртуалізованого обладнання. Кожен створений дроплет – це новий сервер, який ви можете використовувати як самостійний, так і як частину більшої хмарної інфраструктури [25]. Їх і було вирішено використовувати для розгортання проекту. На рис. 4.25 зображено процес створення нового дроплету.

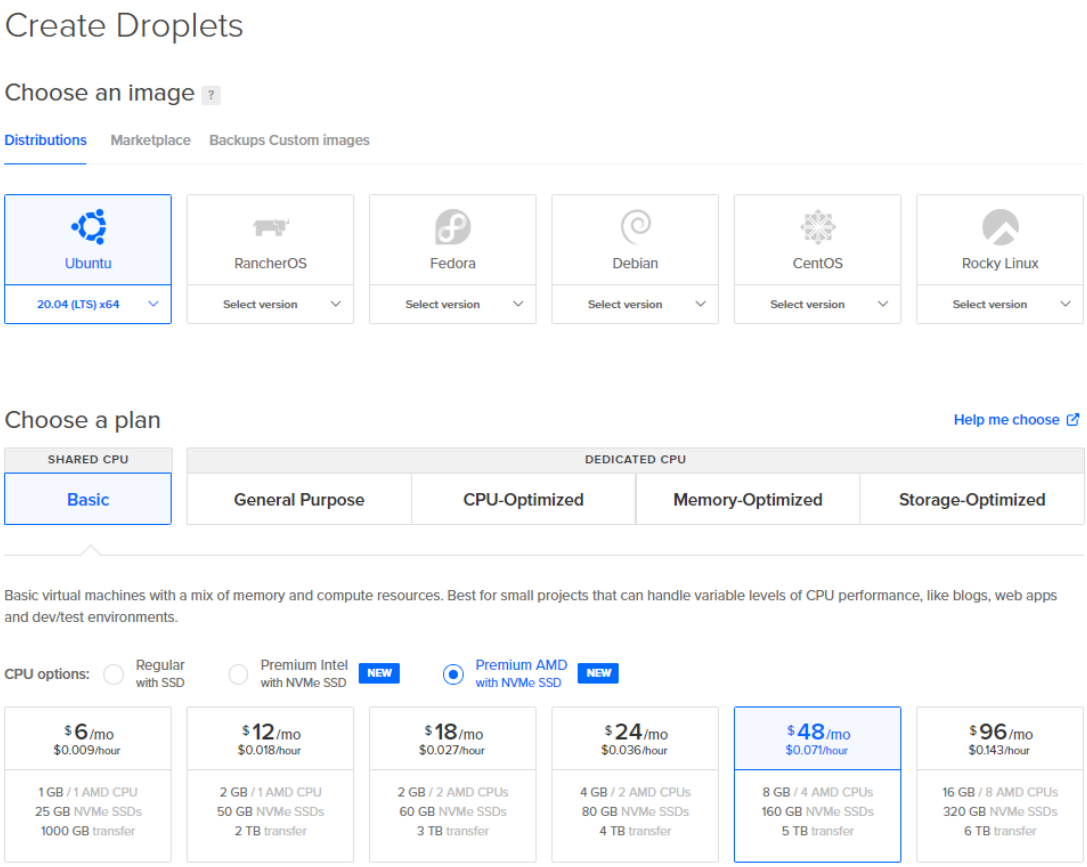

Рисунок 4.25 – Створення дроплету

Провівши тестування на завантаження та відмово стійкість виявилося, що одного дроплету замало, щоб витримувати великий наплив трафіку в момент дропу товару. Для вирішення цієї проблеми було вирішено провести горизонтальне масштабування.

Горизонтальне масштабування – розбиття системи на дрібніші структурні компоненти та рознесення їх по окремих фізичних машинах (або їх групах), та (або) збільшення кількості серверів, що паралельно виконують одну й ту саму функцію.

Першим етапом горизонтального масштабування було винесення на окремі дроплети бази даних, сховища та сервісу відстеження за наявністю товару. Тепер, кожна з цих одиниць розташовувалась на окремому сервері і не навантажувала роботу веб-застосунку, що обробляв запити користувачів.

Далі було вирішено створити ще 7 дроплетів саме для розгортання веб-застосунку. На цих дроплетах розміщувалися однакові застосунки, але у

такий спосіб вони розділяли між собою навантаження. Крім того, для розподілення навантаження між дроплетами було вирішено використати Load Balancer. Це високо доступний продукт, який дає змогу розробникам і підприємствам розподіляти трафік по своїй інфраструктурі, щоб досягти 100-відсоткової безвідмовної роботи своїх виробничих навантажень. Список створених дроплетів зображено на рис. 4.26.

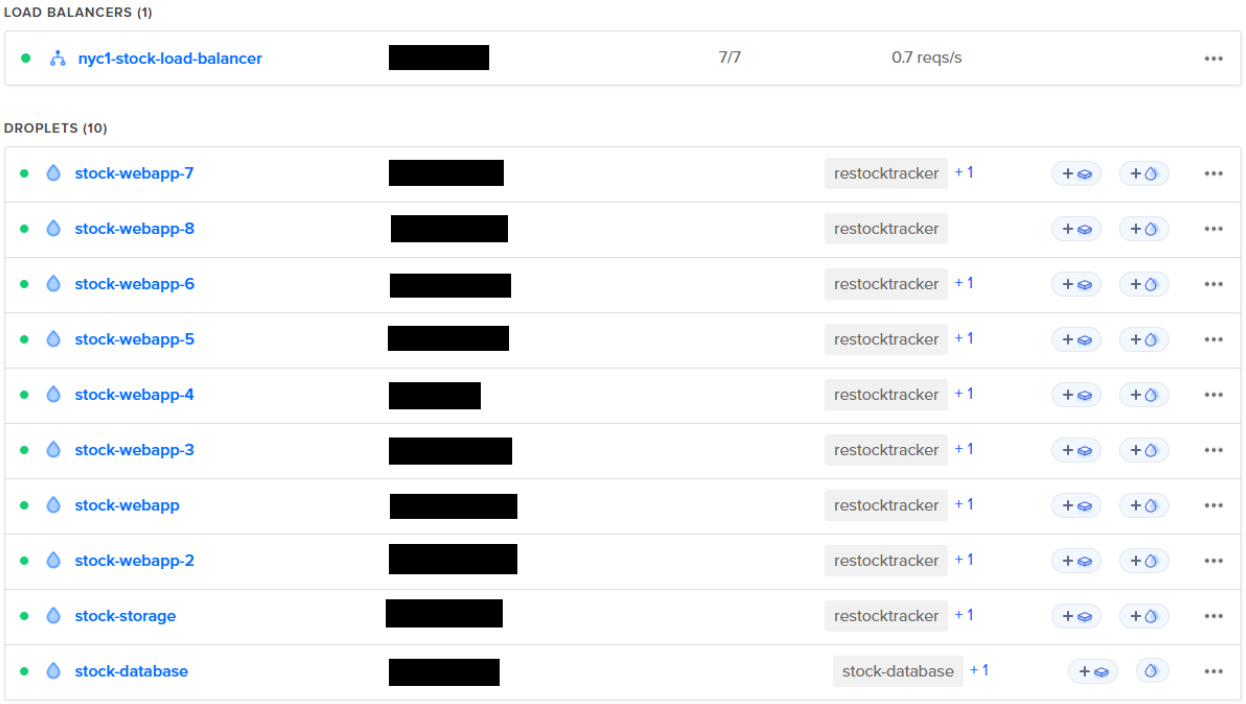

Рисунок 4.26 – Список дроплетів

Для зручного та швидкого розгортання та внесення змін на сервер було вирішено скористатися технологією Docker.

Docker – це набір платформ як сервісних продуктів, які використовують віртуалізацію на рівні ОС для доставки програмного забезпечення в пакетах, які називаються контейнерами.

Контейнер – це стандартна одиниця програмного забезпечення, яка упаковує код і всі його залежності, щоб програма швидко й надійно працювала від одного обчислювального середовища до іншого. Образ контейнера Docker – це легкий, автономний, виконуваний пакет програмного забезпечення, який включає все необхідне для запуску програми: код, час виконання, системні інструменти, системні бібліотеки та налаштування.

Зображення контейнерів стають контейнерами під час виконання, а у випадку контейнерів Docker – зображення стають контейнерами, коли вони запускаються на Docker Engine. Доступне як для програм на базі Linux, так і для Windows, контейнерне програмне забезпечення завжди буде працювати однаково, незалежно від інфраструктури. Контейнери ізолюють програмне забезпечення від середовища і гарантують, що воно працює рівномірно, незважаючи на відмінності, наприклад, між розробкою та стадією. На рис. 4.27 наведено код докер файлу для бекенд частини застосунку.

```
# FROM public.ecr.aws/bitnami/node:14-prod
FROM node:14

RUN apt-get update -y
RUN apt-get install -y build-essential cmake autoconf automake libtool nasm make libpng-dev gcc g++ openssl curl git

# Install Chrome for Selenium
RUN curl https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb -o /chrome.deb
RUN dpkg -i /chrome.deb || apt-get install -yf
RUN rm /chrome.deb

# Install chromedriver for Selenium
RUN curl https://chromedriver.storage.googleapis.com/2.31/chromedriver_linux64.zip -o /usr/local/bin/chromedriver
RUN chmod +x /usr/local/bin/chromedriver

WORKDIR /app

RUN npm cache clean --force

COPY package*.json ./
RUN npm install
COPY . .

RUN npm run build

EXPOSE 4000

# start command
CMD [ "node", "--max-http-header-size=80000000000000", "dist/main" ]
```

Рисунок 4.27 – Докерфайл

Крім того, DigitalOcean надає інструмент для зручної роботи з докер контейнерами, який називається реєстр контейнерів. Реєстр контейнерів DigitalOcean (DOCR) – це приватний реєстр образів Docker з підтримкою додаткового інструментарію, який забезпечує інтеграцію з вашим середовищем Docker і кластерами DigitalOcean Kubernetes. Реєстри DOCR є приватними та розміщені в центрах обробки даних, де керують кластерами DigitalOcean Kubernetes для безпечного, стабільного та продуктивного розгортання зображень у ваші кластери.

Також, для підвищення безпеки та надійності сервісу було вирішено використовувати сервіс Cloudflare. Cloudflare – це безкоштовна система, яка діє як проксі між клієнтом та сервером. Діючи як проксі-сервер, Cloudflare кешує статичний вміст сайту, згодом зменшуючи кількість запитів до серверів, дозволяючи відвідувачам отримати доступ до сайту.

Висновки до розділу 4

У розділі описані усі етапи розробки програмного забезпечення, рішення які приймалися в процесі розробки та процес інтеграції сторонніх сервісів.

Було описано принцип роботи з базою даних за допомогою ORM Sequelize, використання міграцій, сидів, моделей та засоби виконання запитів до БД.

Розписана структура сторінок та інших графічних елементів візуальної частини застосунку. Описано процес розробки API застосунку та створення сервісу стеження за наявністю товарів на маркетплейсах.

У останньому пункті розділу наведено процес розгортання застосунку на сервері та розглянуті технології які для використовувалися для цього.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра розроблено програмне забезпечення стеження за наявністю товару на маркетплейсах. Було проаналізовано предметну область та сформовані функціональні та не функціональні вимоги до застосунку. Досліджено методи та засоби збирання інформації на веб-ресурсах, проаналізовано аналогічне ПЗ, виявлено його переваги та недоліки.

Перед початком розробки проведено моделювання та декомпозицію предметної області. За допомогою графічної нотації UML було побудовано діаграму варіантів використання системи, концептуальну, логічну та фізичні моделі, і діаграму класів застосунку.

Далі було проведено аналіз існуючих засобів розробки веб-застосунків та визначено основний технологічний стек. Відповідно до обраної архітектури програмного забезпечення були обрані наступні технології: фреймворк Nuxt.js для розробки візуальної частини застосунку, фреймворк NestJS було використано для серверної частини, а для роботи з базою даних – СКБД MySQL.

У останньому розділі було описано процес розробки програмного продукту. А саме принцип роботи з базою даних за допомогою ORM Sequelize, описано структуру сторінок і графічних елементів візуальної частини застосунку, описано процес розробки API та системи стеження за товарами, а також, процес розгортання застосунку на сервері.

Результатом даної роботи є діючий застосунок, який забезпечує високий рівень надійності та безпеки, відповідає усім вимогам які були сформовані до початку розробки та надає якісний сервіс для його користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is UML | Unified Modeling Language. URL: https://www.uml.org/what-is-uml.htm. (дата звернення: 25.05.2022)
2. Фаулер М. UML. Основы. 3 издание. 2005
3. Use-case diagrams - IBM Documentation. URL: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-use-case. (дата звернення: 27.05.2022)
4. Thomas Connolly, Carolyn Begg. Database Systems: A Practical Approach to Design, Implementation, and Management. 2018.
5. Martin Fowler. UML Distilled. 2003.
6. ER Diagram (ERD) - Definition & Overview | Lucidchart. URL: https://www.lucidchart.com/pages/er-diagrams. (дата звернення: 28.05.2022).
7. What is 3-tier Application Architecture? URL: https://www.techtarget.com/searchsoftwarequality/definition/3-tier-application. (дата звернення: 01.06.2022).
8. What is Three-Tier Architecture | IBM. URL: https://www.ibm.com/cloud/learn/three-tier-architecture. (дата звернення: 01.06.2022).
9. Front End Development: Key Technologies and Concepts | AltexSoft. URL: https://www.altexsoft.com/blog/front-end-development-technologies-concepts/. (дата звернення: 02.06.2022).
10. Comparing the most popular frontend JavaScript frameworks. URL: https://www.educative.io/blog/compare-popular-frontend-javascript-frameworks. (дата звернення: 02.06.2022).
11. Devloop || Зачем исплоьзовать Nuxt.js? URL: https://devloop.pro/ru/blog/why-i-use-nuxt. (дата звернення: 02.06.2022).
12. Pros and Cons of Node.js Web App Development | AltexSoft. URL: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-node-js-web-app-development/. (дата звернення: 03.06.2022).

13. Introduction to Node. URL: https://nodejs.dev/learn/introduction-to-nodejs. (дата звернення: 03.06.2022).
14. Documentation | NestJS - A progressive Node.js framework. URL: https://docs.nestjs.com/. (дата звернення: 04.06.2022).
15. Розробка додатків на NestJS. URL: https://brander.ua/technologies/nestjs. (дата звернення: 05.06.2022).
16. SASS vs LESS | Top 6 Most Useful Differences To Learn. URL: https://www.educba.com/sass-vs-less/. (дата звернення: 05.06.2022).
17. Database Management Systems (DBMS) Comparison: MySQL, PostgreSQL, and more | AltexSoft. URL: https://www.altexsoft.com/blog/business/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/. (дата звернення: 07.06.2022).
18. SQL vs NoSQL: 5 Critical Differences | Integrate.io. URL: https://www.integrate.io/blog/the-sql-vs-nosql-difference/.(дата звернення: 07.06.2022).
19. What is object-relational mapping (ORM)? - Definition from WhatIs.com. URL: https://www.theserverside.com/definition/object-relational-mapping-ORM. (дата звернення: 08.06.2022).
20. Sequelize | Feature-rich ORM for modern TypeScript & JavaScript. URL: https://sequelize.org/. (дата звернення: 08.06.2022).
21. How To Add Jobs To cron Under Linux or UNIX - nixCraft. URL: https://www.cyberciti.biz/faq/how-do-i-add-jobs-to-cron-under-linux-or-unix-oses/. (дата звернення: 09.06.2022).
22. Introduction to Redis | Redis. URL: https://redis.io/docs/about/. (дата звернення: 10.06.2022).
23. Firebase Cloud Messaging | Firebase Documentation. URL: https://firebase.google.com/docs/cloud-messaging. (дата звернення: 11.06.2022).

24. DigitalOcean: Product Overview and Insight - eWEEK. URL: https://www.eweek.com/it-management/digitalocean-product-overview-and-insight/. (дата звернення: 12.06.2022).
25. Droplets :: DigitalOcean Documentation. URL: https://docs.digitalocean.com/products/droplets/. (дата звернення: 13.06.2022).

ДОДАТОК

ПРОГРАМНИЙ КОД ЗАСТОСУНКУ

Класс ProductsController

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  UseInterceptors,
  UploadedFile,
  Query, UseGuards,
} from '@nestjs/common';
import { ProductsService } from './products.service';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { Product } from "./entities/product.entity";
import { FileInterceptor } from "@nestjs/platform-express";
import { diskStorage } from "multer";
import * as fs from "fs";
import { AuthGuard } from '@nestjs/passport';
import { AdminGuard } from '../auth/guards/admin.guard';

@Controller('products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService)
{}

  @Post()
  @UseGuards(AuthGuard(), AdminGuard)
  async create(@Body() createProductDto: CreateProductDto) {
    return await this.productsService.create(createProductDto);
  }

  @Post('uploads')
  @UseGuards(AuthGuard(), AdminGuard)
  @UseInterceptors(
    FileInterceptor('image', {
    storage: diskStorage({
      destination: './public/uploads/temp',
      filename: (req, file, cb) => {
        cb(null, file.originalname)
      }
    }),
  }))
  uploadFile(@UploadedFile() file: Express.Multer.File) {}
```

```
  @Get('/search')
  async search(@Query('value') value: string):
Promise<Product[]> {
    return await this.productsService.search(value);
  }

  @Get('/country/:countryId/search')
  async searchByCountry(@Query('value') value: string,
@Param('countryId') countryId: number): Promise<Product[]> {
    return await this.productsService.searchByCountry(value,
countryId);
  }

  @Get('filters')
  async findByFilters(
    @Query('country') countryId,
    @Query('orderBy') orderBy,
    @Query('sortDir') sortDir,
    @Query('offset') offset
  ): Promise<{ rows: Product[]; count: number }> {
    return await this.productsService.findByFilters(countryId,
orderBy, sortDir, +offset);
  }

  @Get('/limit/:limit')
  async findWithLimit(@Param('limit') limit: string):
Promise<Product[]> {
    return await this.productsService.findWithLimit(+limit);
  }

  @Get('/country/:countryId')
  async findByCountry(@Param('countryId') countryId: string):
Promise<Product[]> {
    return await this.productsService.findByCountry(+countryId);
  }

  @Get('/country/:countryId/limit/:limit')
  async findByCountryWithLimit(@Param('countryId') countryId:
string, @Param('limit') limit: string): Promise<Product[]> {
    return await
this.productsService.findByCountryWithLimit(+countryId, +limit);
  }

  @Get()
  async findAll(): Promise<Product[]> {
    return await this.productsService.findAll();
  }

  @Get('/:id')
  async findOne(@Param('id') id: string): Promise<Product> {
    return await this.productsService.findOne(+id);
  }
```

```
  @Get('/slug/:slug/country/:countryId')
  async findBySlug(@Param('slug') slug: string,
@Param('countryId') countryId: string) {
    return await this.productsService.findBySlug(slug,
+countryId);
  }

  @Patch(':id')
  @UseGuards(AuthGuard(), AdminGuard)
  async update(@Param('id') id: string, @Body()
updateProductDto: UpdateProductDto) {
    return await this.productsService.update(+id,
updateProductDto);
  }

  @Delete('/uploads/:image')
  @UseGuards(AuthGuard(), AdminGuard)
  async removeImage(@Param('image') image: string) {
    fs.unlink(`./public/uploads/products/${image}`, (err) => {
      if (err) console.log('File not was deleted');
    });
  }

  @Delete(':id')
  @UseGuards(AuthGuard(), AdminGuard)
  async remove(@Param('id') id: string) {
    return await this.productsService.remove(+id);
  }

  @Get('/countries/slug/:slug')
  async findCountriesBySlug(@Param('slug') slug:string) {
    return await this.productsService.findCountriesBySlug(slug);
  }
}
```

Клас ProductsService

```
import { HttpException, HttpStatus, Inject, Injectable } from
"@nestjs/common";
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { Product } from "./entities/product.entity";
import { Country } from "../countries/entities/country.entity";
import { Op } from "sequelize";
import { Comment } from "../comments/entities/comment.entity";
import { Subscription } from
"../subscriptions/entities/subscription.entity";
import * as fs from "fs";
import * as path from "path"
import { User } from '../users/entities/user.entity';
import { GroupsService } from '../groups/groups.service';
import { ImageService } from '../image/image.service';
```

```
import * as randomstring from 'randomstring';
import { Identifier } from
"src/app/identifiers/entities/identifier.entity";
import { Stock } from "src/app/stocks/entities/stock.entity";
import { Tracker } from
"src/app/trackers/entities/tracker.entity";
import { CountriesService } from
'../countries/countries.service';

@Injectable()
export class ProductsService {
  constructor(
    @Inject('PRODUCT_REPOSITORY') private productRepository:
typeof Product,
    private countryService: CountriesService,
    private groupsService: GroupsService,
    private imageService: ImageService
    ) {}

  async create(createProductDto: CreateProductDto) {
    if (createProductDto.image !== '') {
      const country = await
this.countryService.findOne(createProductDto.country_id);

      const fileName = randomstring.generate(64);
      const countryCode = country.code.toLowerCase();
      const extname = path.extname(createProductDto.image);
      const dirPath =
`./public/uploads/products/${countryCode}/${createProductDto.slu
g}`;

      if (!fs.existsSync(dirPath)) {
        fs.mkdirSync(dirPath);
      }

      fs.rename(
        `./public/uploads/temp/${createProductDto.image}`,
        `${dirPath}/${fileName}${extname}`,
        async () => {
          if (extname !== '.png') {
            await this.imageService.convertToPng(dirPath,
fileName, extname);
          }
          await
this.imageService.convertToPngWithCompression(dirPath, fileName,
extname);
          await
this.imageService.convertToWebpWithCompression(dirPath,
fileName, extname);
          await this.imageService.createThumbnail(dirPath,
fileName, extname);
          await this.imageService.createWebpThumbnail(dirPath,
```

```
fileName, extname);
        }
      );

      createProductDto.image = fileName;
    }

    const product = await
this.productRepository.create<Product>({...createProductDto});

    if (product) {
      return {
        status: HttpStatus.CREATED,
        message: 'Product created successfully.',
        product
      }
    }

    return new HttpException('Error! Cannot create product',
HttpStatus.INTERNAL_SERVER_ERROR);
  }

  async search(value): Promise<Product[]> {
    return this.productRepository.findAll<Product>({
      where: {
        title: {
          [Op.like]: `%${value}%`
        }
      },
      limit: 5
    })
  }

  async searchByCountry(value: string, countryId: number):
Promise<Product[]> {
    return this.productRepository.findAll<Product>({
      where: {
        title: {
          [Op.like]: `%${value}%`
        },
        country_id: countryId
      },
      limit: 5
    })
  }

  async findByFilters(countryId: number, orderBy: string,
sortDir: string, offset: number): Promise<{ rows: Product[];
count: number }> {
    return this.productRepository.findAndCountAll<Product>({
      limit: 10,
      offset,
```

```
      include: { model: Country, as: 'country', attributes:
['id', 'code'] },
      attributes: ['id', 'title', 'nickname', 'image', 'slug',
'show'],
      where: { country_id: countryId },
      order: [ [ orderBy, sortDir ] ]
    });
  }

  async findWithLimit(limit: number): Promise<Product[]> {
    return this.productRepository.findAll<Product>({
      limit: limit,
      order: [ [ 'createdAt', 'DESC' ] ]
    });
  }

  async findByCountry(countryId: number): Promise<Product[]> {
    return await this.productRepository.findAll<Product>({
      include: { model: Country, as: 'country' },
      where: { country_id: countryId }
    })
  }

  async findByCountryWithLimit(countryId: number, limit:
number): Promise<Product[]> {
    return await this.productRepository.findAll<Product>({
      include: { model: Country, as: 'country', attributes:
['id', 'code'] },
      limit: limit,
      where: { country_id: countryId, show: true },
      order: [ [ 'createdAt', 'DESC' ] ]
    })
  }

  async findAll(): Promise<Product[]> {
    return this.productRepository.findAll<Product>({
      include: { model: Country, as: 'country', attributes:
['id', 'code'] },
      attributes: ['id', 'title', 'nickname', 'image', 'slug',
'show']
    });
  }

  async findBySlug(slug: string, countryId: number) {
      return await this.productRepository.findOne<Product>({
      include: [{
          model: Tracker, as: 'trackers',
          include: [{
            model: Stock, as: 'stock', attributes: ['id',
'name', 'image', 'url'],
            include: [
              { model: Identifier, as: 'identifier', attributes:
```

```
['name'] },
              { model: Country, as: 'country', attributes:
['code'] }
            ]
          }],
          where: { is_tracked: true },
          required: false
        }, {
          model: Comment, as: 'comments', include: [{
            model: User, as: 'user', attributes: ['id',
'username', 'image']
        }]
        }, {
          model: Country, as: 'country',
        }
      ],
      where: { slug: slug, country_id: countryId },
      order: [
        [
          { model: Comment, as: 'comments' },
          'createdAt',
          'DESC'
        ]
      ]
    });
  }

  async findCountriesBySlug(slug) {
    const products = await
this.productRepository.findAll<Product>({
      include: { model: Country, as: 'country' },
      where: { slug: slug }
    });

    return products.map(p => p.country);
  }

  async findOne(id: number): Promise<Product> {
    return this.productRepository.findOne<Product>({
      include: { model: Country, as: 'country' },
      where: { id: id }
    });
  }

  async update(id: number, updateProductDto: UpdateProductDto) {
    const product = await
this.productRepository.findOne<Product>({
      include: { model: Country, as: 'country' },
      where: {
        id: id
      }
    });
```

```
    if (updateProductDto.slug !== product.slug) {
      fs.rename(

`./public/uploads/products/${product.country.code.toLowerCase()}
/${product.slug}`,

`./public/uploads/products/${product.country.code.toLowerCase()}
/${updateProductDto.slug}`,
        err => {
           if (err) {
             console.error('Cannot rename image folder', err);
           }
        });
    }

    if (updateProductDto.image !== product.image) {
      if (updateProductDto.image !== '') {
        const extname = path.extname(updateProductDto.image);
        const dirPath =
`./public/uploads/products/${product.country.code.toLowerCase()}
/${updateProductDto.slug}`;
        const fileName = randomstring.generate(64);


fs.rmdirSync(`./public/uploads/products/${product.country.code.t
oLowerCase()}/${updateProductDto.slug}`, {
          recursive: true
        });

        if (!fs.existsSync(dirPath)) {
          fs.mkdirSync(dirPath);
        }

        fs.rename(
          `./public/uploads/temp/${updateProductDto.image}`,
          `${dirPath}/${fileName}${extname}`,
          async () => {
            if (extname !== '.png') {
              await this.imageService.convertToPng(dirPath,
fileName, extname);
            }
            await
this.imageService.convertToPngWithCompression(dirPath, fileName,
extname);
            await
this.imageService.convertToWebpWithCompression(dirPath,
fileName, extname);
            await this.imageService.createThumbnail(dirPath,
fileName, extname);
            await this.imageService.createWebpThumbnail(dirPath,
fileName, extname);
```

```
          }
        );

        updateProductDto.image = fileName;
      } else {
        try {

fs.rmSync(`./public/uploads/products/${product.country.code.toLo
werCase()}/${product.slug}`, { recursive: true, force: true });
        } catch (e) {
          console.log('error', e);
        }
      }
    }

    const res = await
this.productRepository.update(updateProductDto,{
      where: { id: id }
    });

    if (res) {
      return {
        status: HttpStatus.OK,
        message: 'Product updated successfully.'
      }
    }

    return new HttpException('Error! Cannot update product.',
HttpStatus.INTERNAL_SERVER_ERROR);
  }

  async remove(id: number) {
    const product = await this.findOne(id);


fs.rmdirSync(`./public/uploads/products/${product.country.code.t
oLowerCase()}/${product.slug}`, {
      recursive: true
    });

    const groups = await this.groupsService.getByProductId(id);

    const res = await this.productRepository.destroy({
      where: { id: id }
    });

    if (res) {
      for (const group of groups) {
        await this.groupsService.remove(group.id);
      }

      return {
```

```
        status: HttpStatus.OK,
        message: 'Product deleted successfully'
      }
    }

    return new HttpException('Error! Cannot remove product.',
HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Клас ArgosScraper

```
import { Injectable } from '@nestjs/common';
import { TrackersService } from
"../../app/trackers/trackers.service";
import * as cheerio from "cheerio";
import { Tracker } from
"../../app/trackers/entities/tracker.entity";
import { ConfigService } from "@nestjs/config";
import { UsersService } from "../../app/users/users.service";
import { ScrapResultDto } from "../dto/scrap-result.dto";
import { ScraperLogger } from "../../logger/scraper.logger";
import { MailService } from "../../mail/mail.service";
import { NotificationsService } from
'../../app/notifications/notifications.service';
import { take } from 'rxjs/operators';
import { ScraperInterface } from '../scraper.interface';
import { Stock } from '../../app/stocks/entities/stock.entity';
import { StocksService } from '../../app/stocks/stocks.service';
import { HttpService } from '@nestjs/axios';
import { ScraperHelper } from '../scraper.helper';

@Injectable()
export class ArgosScraper implements ScraperInterface {
  stock: Stock;
  trackers: Tracker[];
  scrapResult: ScrapResultDto;

  constructor(
    private scraperLogger: ScraperLogger,
    private usersService: UsersService,
    private trackersService: TrackersService,
    private configService: ConfigService,
    private mailService: MailService,
    private notificationsService: NotificationsService,
    private httpService: HttpService,
    private stocksService: StocksService,
    private scraperHelper: ScraperHelper,
    ) {
  }

  async getData(): Promise<ScrapResultDto> {
```

```
    this.stock = await this.stocksService.findOne(10);

    this.stock.time_to_scrape = this.stock.time_to_scrape === 0
? this.stock.update_rate : --this.stock.time_to_scrape;
    this.stock.save();

    this.scrapResult = new ScrapResultDto();
    this.scrapResult.stock = this.stock;
    this.scrapResult.isSuccess = true;

    this.trackers = await
this.trackersService.findByStock(this.stock.id);

    if (this.trackers.length > 0 && this.stock.time_to_scrape
=== 0) {
      const date = new Date().toISOString().slice(0,
19).replace('T', ' ');
      console.log(`${date} | ${this.stock.name} scraping
started...`);

      for (const tracker of this.trackers) {
        this.scrap(tracker);
      }
    }

    return this.scrapResult;
  }

  async scrap(tracker) {
    if (tracker.last_track >= 0) {


this.httpService.get(`https://app.zenscrape.com/api/v1/get?&url=
${tracker.product_url}&render=true&premium=true`, {
        headers: {
          'apikey':
this.configService.get<string>('zenscrape.apiKey')
        }
      }).pipe(take(1))
        .toPromise()
        .then(async response => {
          if (response.status === 200) {

            console.log(`scraped => tracker: ${tracker.id},
stock: ${this.stock.name}`);

            const parseResult = await
this.parsePage(response.data);

            const isTitleUpdated = tracker.product_title !==
parseResult.title && parseResult.title !== null;
            const isPriceUpdated = tracker.price !==
```

```
parseFloat(parseResult.price) && parseResult.price !== null;
            const isInStockUpdated = tracker.in_stock !==
parseResult.availability && parseResult.availability !== null;

            if (isPriceUpdated || isInStockUpdated ||
isTitleUpdated) {
              tracker.price = isPriceUpdated ? parseResult.price
: tracker.price;
              tracker.product_title = isTitleUpdated ?
parseResult.title : tracker.product_title;

              if(tracker.in_stock && isInStockUpdated) {
                if(!tracker.updated_time) {
                  tracker.updated_time = ((new
Date().toISOString()));
                }
              }

              const isAwaited = !tracker.updated_time || await
this.scraperHelper.isTimePassed(tracker.updated_time,ScraperHelp
er.timeToUpdate);
              if (isAwaited) {
                tracker.in_stock = isInStockUpdated ?
parseResult.availability: tracker.in_stock;
                tracker.updated_time = null;
                if (parseResult.availability && isInStockUpdated
&& tracker.is_changed) {
                  tracker.last_out = new Date().toISOString();
                } else if (!parseResult.availability &&
isInStockUpdated && tracker.is_changed) {
                  tracker.last_stock = new Date().toISOString();
                }
              }

              tracker.is_changed = true;

              const isTimePassed = await
this.scraperHelper.isTimePassed(tracker.last_notice,20);

              if (isTimePassed && isInStockUpdated &&
parseResult.availability) {
                tracker.last_notice = (new
Date()).toISOString();
              }

              await tracker.save();

              if (isTimePassed && isInStockUpdated &&
parseResult.availability) {
                await
this.notificationsService.sendNotifications(tracker);
              }
```

```
              console.log(`updated => tracker: ${tracker.id},
stock: ${this.stock.name}`);
            }

            const date = new Date().toISOString().slice(0,
19).replace('T', ' ');


this.stocksService.updateAfterScraping(this.stock.id, {
              last_successful_update: date,
              last_update: date
            });

            await
this.scraperLogger.logScrapResult(this.stock.id, {
              date: new Date().toISOString().slice(0,
19).replace('T', ' '),
              status: 'success',
              product: tracker.product,
              payload: `title: ${parseResult.title}, price:
${parseResult.price}, availability:
${parseResult.availability}`,
            });

          } else {
            const date = new Date().toISOString().slice(0,
19).replace('T', ' ');
            console.error(`failed => tracker: ${tracker.id},
stock: ${this.stock.name}`);

this.stocksService.updateAfterScraping(this.stock.id, {
              last_update: date
            });
            await
this.scraperLogger.logScrapResult(this.stock.id, {
              date,
              status: 'error',
              product: tracker.product,
              payload: `Response status: ${response.status},
message: ${response.statusText}`,
            });
          }
        }).catch(async error => {

          if (error.response.status === 404) {
            if (tracker.in_stock || tracker.in_stock === null) {
              tracker.in_stock = false;
              tracker.is_changed = true;

              if (tracker.in_stock) {
                tracker.last_stock = new Date().toISOString();
```

```
              }

              await tracker.save();
            }
          }

          const date = new Date().toISOString().slice(0,
19).replace('T', ' ');
          console.error(`failed => tracker: ${tracker.id},
stock: ${this.stock.name}`);
          this.stocksService.updateAfterScraping(this.stock.id,
{
            last_update: date
          });
          await this.scraperLogger.logScrapResult(this.stock.id,
{
            date: new Date().toISOString().slice(0,
19).replace('T', ' '),
            status: 'error',
            product: tracker.product,
            payload: error.message
          });
      });
    }
  }

  async parsePage(body) {
    const parseResult = {
      title: null,
      price: null,
      availability: null
    }

    const $ = cheerio.load(body);

    const notAvailableSelector = $('h1#h1title');

    if (notAvailableSelector.length > 0) {
      parseResult.availability = false;
      return parseResult;
    }

    const titleSelector = $('h1 > span');
    const priceSelector = $('.pdp-pricing-module h2');
    const addToCartBtnSelector = $('.add-to-trolley-main
button');
    const outOfStockImgSelector = $('div[data-el=badges] >
img');
    const outOfStockImgAlt = outOfStockImgSelector.length > 0 ?
outOfStockImgSelector.attr('alt') : null;

    if (addToCartBtnSelector.length > 0 &&
```

```
outOfStockImgSelector.length > 0) {
      parseResult.availability = outOfStockImgAlt !==
'out_of_stock';
    }

    parseResult.price = priceSelector.length > 0 ?
parseFloat(priceSelector.text().slice(1)) : null;
    parseResult.title = titleSelector.length > 0 ?
titleSelector.first().text().trim() : null;

    return parseResult;
  }
}
```