КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультету радіофізики, електроніки та комп'ютерних систем
Кафедра комп'ютерної інженерії

АВТОМАТИЗОВАНА РОЗПОДІЛЕНА СИСТЕМА АГРЕГАЦІЇ ПРОПОЗИЦІЙ ТОВАРІВ

Дипломна робота бакалавра
студента 4 року навчання
Спеціальність: 123 «Комп'ютерна інженерія»
Артема ШЕВЧЕНКО

Науковий керівник
кандидат фіз.-мат. наук Юрій БОЙКО
доцент кафедри комп'ютерної інженерії

Рецензент
доктор фіз.-мат. наук ЄВГЕН ІВОХІН
професор кафедри системного аналізу та теорії прийняття рішень

До захисту допускаю
завідувач кафедрою
кандидат фіз.-мат. наук Юрій БОЙКО
доцент кафедри комп'ютерної інженерії

Ухвалено на засіданні кафедри "____"_____2022 р.,
протокол № ____

Київ 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ORM	- Object-Relational Mapping
EF	- Entity Framework
AMQP	- Advanced Message Queuing Protocol
REST	- Representational State Transfer
ASP	- Active Server Pages
HTTP	- HyperText Transfer Protocol
SQL	- Structured query language
JSON	- JavaScript Object Notation
JSONB	- JavaScript Object Notation Binary
XML	- Extensible Markup Language
API	- Application Programming Interface
СУБД	- Система керування базами даних

ВСТУП

За останні декілька років, у зв’язку з пандемією коронавірусу та масовим переходом населення на безконтактні способи взаємодії, ринок електронної комерції показав вражаюче зростання по всьому світу. З року в рік частка товарів, що продаються онлайн, тільки збільшується і, на даний момент, більшість напрямків комерції використовує переваги інтернет-торгівлі. Основу такого типу продажів складає роздрібна торгівля, що спрямована на кінцевого споживача. Саме вона дозволяє максимізувати застосування індивідуального підходу до кожного користувача, а також різко зменшити час на знайомство з товаром, аналіз ринкових цін та оформлення покупки. З іншого боку інтернет торгівля дозволяє магазинам значно зменшити витрати на підтримку фізичних магазинів та полегшити логістику.

Висока ефективність та відсутність уніфікації такого типу продажів призвела до того, що ринок онлайн торгівлі став дуже фрагментованим. Щорічне зростання ринку онлайн торгівлі знаходиться в межах 25 - 35%. Таке стрімке зростання ринку призводить до закономірного збільшення кількості компаній, які займаються роздрібною торгівлею. Зараз існують тисячі незалежних майданчиків, на яких можуть продаватися унікальні товари. З часом кількість торговельних майданчиків лише збільшується, що призводить до пропорційного збільшення кількості затраченого часу та складності пошуку відповідного товару. З проведеного дослідження ринку було виявлено, що схожого сервісу, який би відповідав всім вимогам, ще не представлено у загальному доступі. Фрагментованість ринку та відсутність прямої конкуренції відкриває нові можливості для створення сервісів агрегації пропозицій товарів із різних торгових майданчиків.

Метою роботи є створення розподіленої системи з можливістю автоматизованого збору, обробки та зберігання інформації про унікальні пропозиції товарів від різних виробників та маркетплейсів. Архітектура

такої системи дозволить ефективно розподіляти навантаження між сервісами та хостами та оперативно переналаштовувати вже існуючі та додавати нові маркетплейси для збору даних.

Практична значимість роботи визначається можливістю використання автоматизованих систем для зменшення часу, який користувач витрачає на пошук певного товару, порівняння ціни в різних торгівельних мережах, отримання повної інформації про товар та ознайомлення його з досвідом використання цього товару іншими покупцями.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ СИСТЕМ АГРЕГАЦІЇ ПРОПОЗИЦІЙ ТОВАРІВ

1.1 Автоматизована система агрегації пропозицій товарів

Рішенням проблеми фрагментації ринку роздрібної торгівлі через інтернет може бути система агрегації пропозицій товарів. Більшою мірою, але не виключно, ця система направлена саме на спеціалізовані електронні магазини, які пропонують унікальні товари від окремого виробника. В цьому випадку взаємодія учасників процесу агрегації є інформаційним процесом, який тісно пов'язаний зі створенням, обміном, зберіганням і використанням різної інформації. Функціонування такої системи вимагає обов’язкової присутності таких компонентів:

- програмно-апаратного комплексу підтримки інформаційного процесу;
- сервісу обміну повідомленнями між частинами розподіленої системи;
- наповнення інформаційних ресурсів, з якими відбувається взаємодія;
- сервісу взаємодії користувачів з самим середовищем.

Використовуючи системний підхід до побудови такого роду сервісу, ми можемо представити його як сукупність незалежних частин, кожна з яких вирішує унікальний комплекс поставлених завдань. Побудова сучасної автоматизованої системи агрегації пропозицій товарів є нелегким завданням, яке вимагає великої кількості залученої інтелектуальної праці та використання сучасних технологій проектування та розробки.

В Україні вже існують програмні комплекси, які спрямовані, в тій чи іншій мірі, на подолання поставленого завдання, але жоден з них не вирішує її повністю. Найбільшим гравцем, який має схожий продукт, є компанія Lamoda. Ця система надає можливість великій кількості продавців використовувати єдиний маркетплейс, щоб надавати кінцевим покупцям

свої товари. Але, при цьому, система Lamoda є повністю закритою. Це накладає деякі обмеження:

- компанії, що хочуть почати використовувати торгівельний майданчик, повинні пройти повну модерацію, перед тим як отримати змогу почати ним користуватися. При цьому зберігається велика ймовірність, що компанії буде відмовлено, якщо управління маркетплейсу вирішить, що надавати право користування економічно не вигідно;
- продавці не мають можливості додавати пропозиції товарів з вже існуючих торговельних майданчиків. Через це вони повинні або підтримувати одночасно два або більше сервіси, або перейти повністю на платформу Lamoda.

Наведені причини не дозволяють сказати, що існує рішення наявної проблеми. Тому доцільною є розробка системи, що дозволить в повній мірі вирішити поставлені задачі.

1.2 Електронний торгівельний майданчик

Електронний торгівельний майданчик — один з типів веб-сайтів електронної комерції, на якому інформацію про товари надають треті сторони, які водночас і займаються питаннями отримання, розподілу та доставки товарів. Таким чином маркетплейси є основним типом багатоканальної контент-стратегії, яку можуть використовувати організації для значного покращення досвіду використання та взаємовідносин зі своїми покупцями.

В такого роду торгівельних майданчиках вся логіка оформлення замовлень, внесення платежів та проведення транзакцій обробляється оператором маркетплейсу. Лише після повного проведення процесу замовлення доставляється до кінцевої компанії, яка займається роздрібною або оптовою торгівлею. При цьому, за проведення такого роду транзакцій, торгові майданчики беруть певну націнку від вартості кожного

оформленого товару.

Загалом, оскільки електронні торговельні майданчики об’єднують велику кількість постачальників, доступність товарів на такому сайті більша, ніж в звичайних інтернет-магазинах окремих постачальників. [1]

1.3 Спеціалізований електронний магазин

Електронний магазин — тип веб-сайтів електронної комерції, інформацію про товари на якому надає сам виробник товару. Зазвичай, такого роду сервіси використовуються великими компаніями, які в змозі запропонувати кінцевому покупцю широкий асортимент товарів, що можуть в повній мірі покрити деякий напрям потреб. [2]

Унікальною особливістю такої платформи є тісний зв’язок кожного етапу оформлення замовлення. Досягається це за рахунок того, що всі процеси відбуваються на потужностях однієї компанії.

Позитивною стороною такого підходу є більш тісна взаємодія покупця та продавця. Користувач може отримати максимально доступну інформацію про товар, також, зазвичай, на таких сервісах товари продаються за значно нижчими цінами, адже маркетплейс не вносить додаткової націнки на товар. Негативною стороною може бути те, що покупець повинен відвідувати десятки сайтів різних компаній, щоб знайти товар, що в повній мірі здатен покрити його потребу.

РОЗДІЛ 2. ТЕХНІЧНА БАЗА ДЛЯ РОЗРОБКИ АВТОМАТИЗОВАНОЇ РОЗПОДІЛЕНОЇ СИСТЕМИ АГРЕГАЦІЇ ПРОПОЗИЦІЙ ТОВАРІВ

2.1 Модульна система розробки програмного забезпечення .NET та платформа ASP.NET

.NET — це відкритий програмний фреймворк, створений компанією Microsoft. Ця платформа застосовується для створення програмних систем в одному з чотирьох сценаріїв:

- веб-програма ASP.NET
- програма командного рядка
- бібліотека платформи .NET
- програма універсальної платформи Windows

Цей програмний засіб реалізує можливість роботи одночасно на системах сімейств Windows, macOS, та більшості дистрибутивів Linux. Також фреймворк підтримує більшість розповсюджених архітектур центральних процесорів [3].

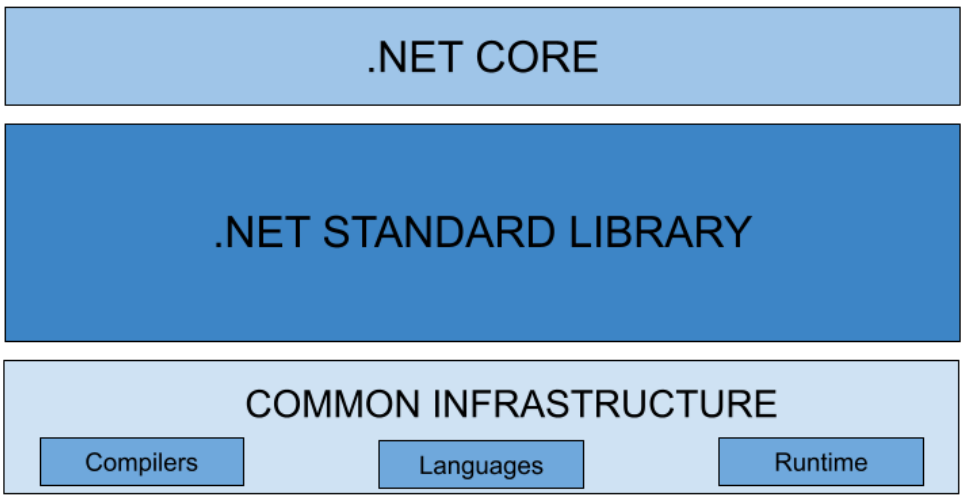

Рис. 1.1 – Структура системи .NET

В той же час, архітектура системи передбачає повну модульність. Це надає можливість, залежно від потреб розробника, в будь-який момент

змінювати середовище виконання, компілятор, основні та додаткові бібліотеки. Самі бібліотеки, при цьому, також модульні. У зв’язку з цим є можливість отримати лише ту частину, яка нам потрібна для виконання поставленого завдання, через систему поширення пакетів NuGet. Такий архітектурний стиль дозволяє дуже точно налаштовувати роботу всієї системи і досягати високої продуктивності [4].

Ключові можливості, що надаються платформою:

- Підтримка великої кількості мов програмування. Застосунок .NET може створюватися з використанням широкої кількості мов програмування;
- Загальний механізм виконання, що розподіляється між усіма мовами програмування, що підтримуються платформою. Це призводить до підтримки одного виділеного набору типів, що підтримується всіми мовами програмування;
- Загальна бібліотека класів. Фреймворк підтримує тисячі попередньо визначених типів, які реалізують більшість можливостей, які можуть знадобитись розробнику під час написання програмного забезпечення;
- Спрощена модель розгортання. Платформа .NET дозволяє декільком версіям одного програмного забезпечення існувати та виконуватися незалежно на одному комп'ютері.

Платформа ASP.NET містить функції низького рівня, що необхідні для отримання, обробки та відправки HTTP запитів. Фреймворк підтримує різні варіанти HTTP-серверів та постачається з набором компонентів проміжного рівня, які полегшують завдання обробки запитів користувачів. Загальна інфраструктура платформи побудована поверх .NET, щоб забезпечити можливість міжплатформної розробки застосунків та служб з використанням стилю REST.

У випадку використання web Api, для обробки запитів застосовуються контроллери. Кожен контроллер представляє собою елемент проміжного рівня, який на вхід отримує дані запиту користувача. Після отримання виклику, відбувається початкова валідація даних та, у випадку успіху, передача керування іншим компонентам системи, що, залежно від бізнес-процесу, виконують потрібні перетворення та генерацію відповіді на запит. Згенерований результат повертається у контроллер та надсилається клієнту.

Основною ідеєю такої архітектури є самодостатність, що відповідає принципу розподілу відповідальності. Кожен контроллер відповідає лише за певний тип запитів. При цьому, бізнес-логіка може бути взагалі не пов’язаною з певним контролером. З таким чітким розподілом відповідальності, з’являється можливість швидкої заміни різних частин застосунку, зменшується складність підтримки та збільшується швидкість розробки нових модулів системи.

Ключові можливості, які надаються фреймворком:

- вбудована підтримка ін'єкції залежностей. Така система представляє собою механізм підтримки слабкої зв’язаності між об’єктами. Замість створення залежних об’єктів безпосередньо або передачі методам специфічних реалізацій класи і методи проектуються так, щоб отримувати типи інтерфейсів. В такому випадку можуть передаватися будь-які взаємозамінні реалізації інтерфейсів, що значно збільшує гнучкість системи;
- інтегрована система конфігурації. Такий механізм дозволяє застосункам взаємодіяти з налаштуваннями середовища машини значно полегшує конфігураційний процес;
- модульний конвеєр запитів. Відповідно до основних принципів платформи, кожен обробник запитів додається лише явно.

Такий підхід дозволяє застосункам бути максимально легкими та використовувати лише потрібні частини;

- вбудована система керування аутентифікацією та авторизацією [5].

2.2 Набір продуктів для автоматизації розгортання та роботи з додатками в середовищах з підтримкою віртуалізації Docker

Docker — відкрите програмне забезпечення, що представляє собою набір сервісів для віртуалізації на рівні операційної системи. При цьому, кожен застосунок упаковується в індивідуальний контейнер. Контейнери ізольовані один від одного та інкапсулюють повний набір залежностей, робочих бібліотек та файлів конфігурації. Легкість кожного з контейнерів та використання лише емуляції рівня операційної системи дозволяє досягнути більшої продуктивності при використанні меншої кількості ресурсів, у порівнянні зі звичайними віртуальними машинами.

Основною перевагою використання сервісу docker є можливість міграції контейнерів між середовищами із застосуванням мінімальних зусиль. При цьому зовсім не важливо, яка операційна система використовується на машині та який набір програмного забезпечення вона підтримує.

Архітектура сервісу Docker складається з двох частин:

- сервер контейнерів;
- клієнтські засоби з інтеграцією до командного рядка.

Сервер забезпечує повну ізоляцію контейнерів на рівні файлової системи, процесів та мережі. Рівень файлової системи реалізується створенням, для кожного з контейнерів, унікального кореневого каталогу. При цьому, відбувається повне відтворення директорій файлової системи, що потрібні для правильного функціонування контейнеризованих програмних засобів.

Рівень процесів надає можливість точного керування доступами та розподілом ресурсів у кожного працюючого екземпляра програмного забезпечення, в той час, як рівень мережі надає можливість кожному з контейнерів звертатися лише до простору імен і відповідним віртуальним мережевим інтерфейсам, що йому належать.

Набір клієнтських засобів надає зручний інтерфейс командного рядка для створення, налаштування, запуску, контролю та зупинки контейнерів. При цьому підтримується можливість використання спеціального сценарного файлу, що надає можливість виконувати попередньо створені послідовності налаштувань. Кожна команда може застосовуватися як до локального серверу контейнерів, так і до будь-якого доступного по мережі. [6]

2.3 Реляційна система керування базами даних

Реляційна СУБД — сукупність програмних засобів, що забезпечують керування створенням та використанням баз даних системи, що виконує керування реляційними базами даних. Реляційні дані моделюють у вигляді таблиць, доступ до яких, зазвичай, відбувається через запити мовою SQL. Доступ до цих даних забезпечується внутрішньою системою, яка дозволяє взаємодіяти з однією або декількома базами даних, та надає можливість доступу до інформації, що в них міститься. При цьому системою надаються методи ефективного вводу, обробки, оновлення та видалення даних.

Модель даних організовує інформацію в таблиці, кожна з яких складається з рядків і стовпців. Кожен з рядків, при цьому, відображає один унікальний запис, який містить первинний ключ, що в повній мірі забезпечує можливість ідентифікації окремої частини даних в таблиці. Наявність унікального ключа надає можливість запису однієї таблиці бути пов’язаним з записом іншої, через посилання на первинний ключ батьківського запису. [7]

2.4 Технологія об'єктно-реляційної проекції Entity Framework

Entity framework — відкрита об’єктно орієнтована технологія доступу до даних, створена компанією Microsoft для системи .NET. Представляє собою рішення ORM, що надає можливість взаємодії з інформацією, що зберігається в підключеній базі даних, через конструкції мови програмування платформи .NET. Це надає змогу розробникам працювати з інформацією таблиць баз даних у такий самий спосіб, як вони працюють зі звичайними об’єктами вибраної мови програмування. Такий рівень абстракції дозволяє значно зменшити складність розробки та підтримки програмного коду продукту.[8]

При ініціалізації проекту з EF відбувається створення об'єктної системи моделей, яка повністю відповідає внутрішній структурі бази даних, з якою відбувається робота.

Доступ до даних в EF виконується за допомогою моделей даних. Кожна модель складається з класів сутностей і об’єкта контексту, який відповідає певному сеансу з базою даних. Саме цей контекст надає можливість виконувати операції з базою даних. Класи сутностей містять набори властивостей, кожна з яких відповідає певному стовпчику з таблиці. При цьому тип властивості жорстко пов’язаний з типом записів, які він презентує в базі даних. Для більш тонкого налаштування використовуються анотації сутностей. Вони надають можливість додаткового опису стовпчиків таблиці, як, наприклад, максимальна довжина запису, чи є він обов’язковим для заповнення, чи є він первинним ключем та інше.

Виходячи з того, що реляційні дані представлені у формі об’єктів мови програмування, з’являється можливість використовувати технологію LINQ для формування запитів до бази даних. Це спрощує сприйняття програмного коду та надає можливість привести загальний вигляд запитів до аналогічних мовою SQL.

2.5 Брокер повідомлень RabbitMQ

RabbitMQ — відкрита система обміну повідомленнями, побудована з використанням протоколу AMQP. Представляє собою набір сервісів, що надають можливість швидкої комунікації між елементами розподіленої системи.

Ця система використовується для налаштування ефективного обміну даними між компонентами розподіленої системи. Основна ідея полягає в тому, що кожен незалежний елемент системи може обмінюватися довільними повідомленнями через спеціальний брокер, що виконує перетворення, маршрутизацію та доставку повідомлень. При цьому повідомлення відразу не надсилається до певного споживача, а направляється до спеціального обміну, який залежно від супровідних даних повідомлення, вже сам розподіляє та маршрутизує потоки даних. Такий рівень абстракції дозволяє видавцям та споживачам повідомлень навіть не знати про наявність та логіку роботи один одного.

Система RabbitMQ підтримує такі варіанти обміну:

- обмін один до одного. При такому типі обміну один видавець повідомлень обмінюється лише з одним споживачем;
- обмін один до декількох. Цей тип обміну дозволяє надсилати одне повідомлення відразу декільком, чітко визначеним, споживачам;
- обмін один до всіх. В цьому режимі повідомлення доставляються відразу всім клієнтам, що налаштовані на відповідний прийом даних.[9]

Завдяки своїй легкості та високій швидкості RabbitMQ підходить як для невеликих систем з декількома сервісами, так і для розподілених по всьому світу мереж з тисячами споживачів.

2.6 Архітектурний стиль побудови систем REST

REST — архітектурний стиль побудови розподілених систем, при якому з кожним запитом передається самоописний стан. Головним призначенням цього підходу є полегшення взаємодії різних компонентів системи через обов'язкову реалізацію чітко визначених обмежень та правил.[10]

Побудова такого типу системи можлива при виконанні двох правил:

- модель взаємодії клієнт-сервер. Розподіл цих сутностей дозволяє значно спростити складність розвитку та підтримки кожного з компонентів. Також це надає унікальну можливість незалежного розвитку кожного з компонентів;
- відмова від зберігання стану сервером. Це правило є основою реалізації відповідного способу взаємодії цих сутностей . При цьому, клієнт будує запити таким чином, щоб сервер міг отримати всю необхідну інформацію з тіла запиту на сформувати валідну відповідь. При цьому, вся необхідна інформація про стан знаходиться саме на клієнті, який, в міру необхідності, сам ініціалізує взаємодію з сервером для отримання інформації.

Додатковою особливістю є використання багатошарової структури мережі, за якої запит може проходити необмежену кількість сервісів до того, як буде оброблений кінцевим сервером. В цьому випадку клієнт може навіть не знати, що відповідь надійшла через деяку кількість проміжних вузлів. Це надає можливість використання проміжного програмного забезпечення для підвищення безпеки, балансування навантаження та кешування відповідей.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЕЛЕМЕНТІВ РОЗПОДІЛЕНОЇ СИСТЕМИ АГРЕГАЦІЇ ПРОПОЗИЦІЙ ТОВАРІВ

Метою роботи є розробка автоматизованої системи агрегації пропозицій одягу з використанням даних, що публічно доступні на спеціалізованих інтернет-майданчиках. Об’єм даних про товари на різних площадках може бути досить великим, що буде призводити до великих часових затримок відповідей на запити користувачів. З цієї причини було прийнято рішення формувати локальну копію упорядкованих даних, саме до яких будуть направлені всі запити користувачів. Самі дані будуть оновлюватися за чітко визначеним розкладом.

Розроблена система буде складатися з чотирьох незалежних частин, кожна з яких буде виконувати чітко визначену роль. Це надасть можливість з легкістю масштабувати сервіси при збільшенні навантаження та додавати нові обробники даних не торкаючись роботи інших. Основні компоненти програмного продукту:

- проксі-сервер nginx;
- основний сервер з бізнес-логікою AZZCore;
- масив аналізаторів пропозицій товару з відкритих джерел;
- система управління базами даних PostgreSQL.

Загальну структуру побудованої системи можна побачити на Рис.2.1

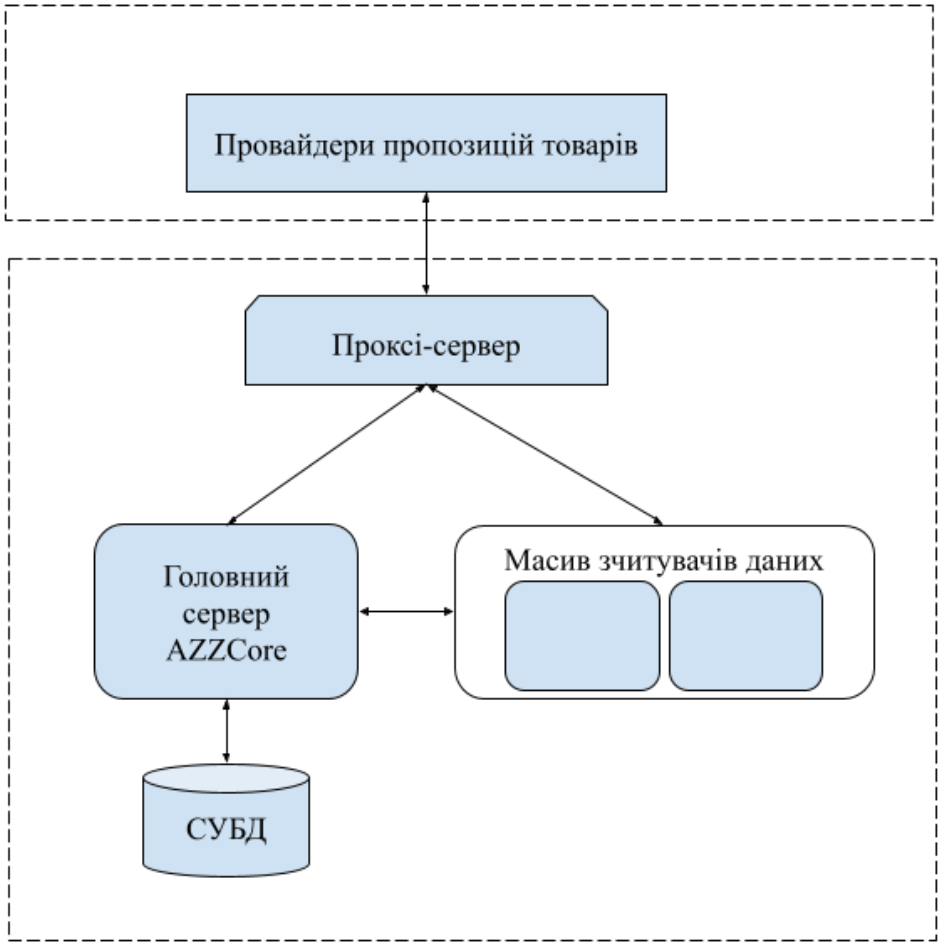

Рис. 2.1 – Загальна структура системи

3.1 Рішення для зчитування даних про товари з відкритих джерел

Отримання даних з загальнодоступних маркетплейсів можливе через надані відкриті API, або через прямий парсинг даних. Більшість з представлених на ринку торговельних майданчиків не надають відкриті методи API. З цієї причини було обрано саме варіант прямого парсингу даних, як основний для задачі отримання пропозицій товарів. Хоча, архітектура побудованої системи дозволяє у будь-який час перейти на використання наданих програмних інтерфейсів для накопичення потрібного асортименту.

Для отримання даних з сервісів провайдерів пропозицій було створено автоматизований масив зчитувачів даних. Кожен з елементів наявного масиву представляє собою окремий мікросервіс, який за командою керуючого сервера виконує звернення на отримання даних про пропозиції одягу до сервера чітко визначеного маркетплейсу. Дані при кожному зверненні передаються у вигляді HTTP(HTTPS) запитів. Наступним кроком є обробка отриманих даних та приведення їх до єдиного стандарту, з яким працює наша система.

Основним процесом, який виконує кожен сервісів є саме перетворення даних. Очевидно, що залежно від типу структури отриманої від сервера інформації, вона потребує різних алгоритмів та підходів для обробки. Саме з цієї причини була обрана багатосервісна архітектура, за якої для кожного з серверів-надавачів даних використовується окремий сервіс. Це надає можливість значно спростити загальну логіку обробки та збільшити ефективність кожного окремого сервісу.

Обробка даних передбачає чотири етапи. На першому етапі з загального потоку даних виділяється лише та інформація, що нас цікавить. Другим етапом є валідація даних. Третім етапом є приведення даних у відповідність до модельної структури нашої системи. На останньому етапі оброблені дані надсилаються до головного сервера.

Весь обмін даними між кожним з мікросервісів та головним сервером відбувається за допомогою брокера повідомлень rabbitMQ. Такий тип обміну повідомленнями було обрано через його відмовостійкість, високу швидкість роботи, велику пропускну здатність та можливість збереження черги повідомлень навіть у випадку припинення роботи сервісу.

В моменти часу між запитами головного сервера на отримання даних, сервіси знаходяться в очікуючому режимі. Загальна схема роботи сервіса представлена на Рис.2.2

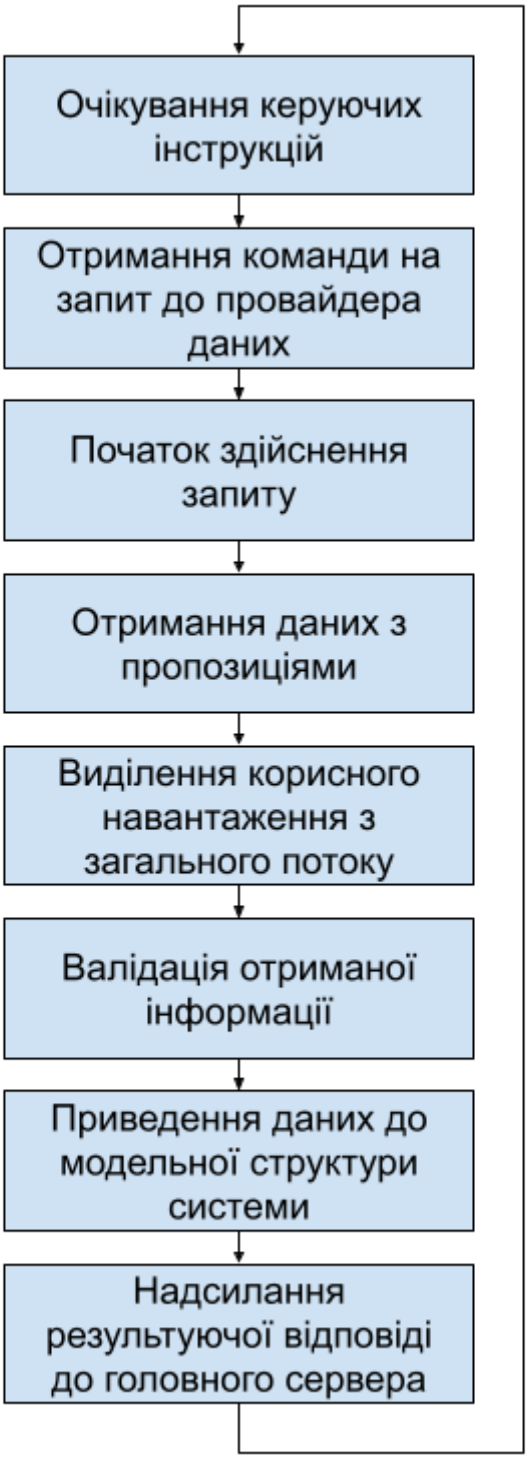

Рис. 2.2 – Послідовність роботи сервіса зчитування даних

3.2 Модуль тривалого зберігання інформації

Задача тривалого зберігання великої кількості даних про товари ставить високі вимоги перед системою управління базами даних. Головними вимогами для такого роду сервісу є висока швидкодія, переносимість між різними сімействами систем та відкритість. З цих причин була обрана СУБД PostgreSQL.

В основі цієї СУБД лежить відкрита кодова база, яка доповнюється та перевіряється на безпеку сотнями незалежних розробників з усього світу. Це надає можливість вільно використовувати обраний програмний продукт та бути впевненим в його безпеці та надійності. Сама СУБД підтримує роботу на системах сімейств Windows, macOS, BSD, Solaris та Linux. Швидкодія PostgreSQL є достатньою для зберігання даних розміром у десятки та сотні терабайт без видимих затримок у роботі. При цьому сам програмний продукт виходить за межі примітивних типів даних, які підтримують більшість сучасних реляційних СУБД, та надає можливість зберігати дані видів JSON/JSONB, XML, Key-value, Array. Це може значно спрощувати завдання збереження даних з масиву зчитувачів.

Перевагою використання системи зчитування та синхронізації даних за розкладом є можливість розміщення СУБД на тому самому хості, на якому будуть розміщуватися самі зчитувачі та головний сервер. Найбільш ресурсомісткі завдання, якими є зчитування даних віддалених майданчиків, їх обробка та створення локальної копії, можна, розподілити в часі так, щоб вони виконувалися в моменти найменшого навантаження на систему. При цьому зберігається висока швидкість доступу до бази даних через відсутність витрат на пересилання даних через мережу.

Розміщення СУБД в окремому контейнері хоча і приводить до зменшення швидкодії через додаткові витрати на віртуалізацію на 5-7%, додатково надає можливість легко переносити сервіс між різними хост-

системами як у зв’язці з іншими частинами системи агрегації пропозицій, так і окремо.

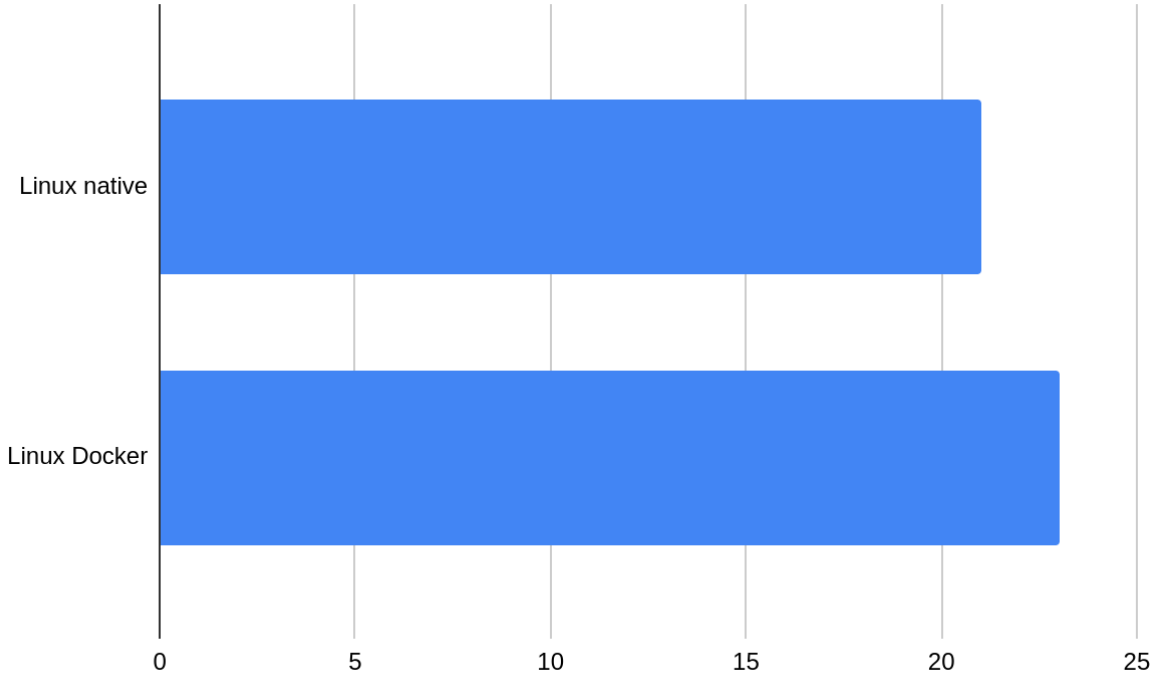

Рис. 2.3 – Час обробки в секундах задачі додавання 10000 рядків до бази даних, в залежності від використаного хосту

3.3 Сервер керування логікою роботи AZZCore

Для керування всією наявною в проекті бізнес-логікою було розроблено спеціальних сервер AZZCore з використанням фреймворку ASP.NET. Архітектурний стиль побудови сервісу надає можливість використання автономних модулів, кожен з яких виконує чітко визначену задачу та має можливість незалежного підключення. При цьому, використання інверсії залежностей дозволяє змінювати модуль на аналогічний, що реалізує відповідний інтерфейс, зміною однієї стрічки програмного коду.

Такий високий рівень гнучкості, та наявність вбудованих модулів, дозволяє реалізовувати на представленій платформі сервери для більшості

можливих завдань. Сам же AZZCore окрім наданих платформою ASP.NET, має 5 вбудованих додаткових частин:

- модуль роботи з файловим середовищем системи;
- модуль роботи з RabbitMQ;
- модуль роботи з базою даних;
- модуль планувальника завдань;
- модуль ведення логів

Основним завданням всієї системи агрегації пропозицій одягу є накопичення та зберігання даних. Для цього завдання було створено модуль роботи з базою даних з використанням Entity Framework. Це надає можливість в об'єктно-орієнтованому стилі звертатися до наявних даних з коду обробників сервера. При цьому використовується синтаксис звернення до вбудованих в мову об'єктів, що надає можливість розробки сервісу та побудови звернень до бази даних без знання мови SQL. Іншою перевагою є перекладання відповідальності контролю наявності ресурсів, та доступу до них на Entity Framework, який зарекомендував себе як швидкий та стабільний в тисячах інших програмних продуктів.

Специфікою пропозицій одягу є наявність великої кількості файлів зображень, які пов'язані з кожним з товарів. Спроба зберігання цих даних в одній директорії системи може приводити до пропорційного збільшення часу доступу до кожного з них. Рішенням цієї проблеми, яке було реалізоване в системі AZZCore, є спеціальний алгоритм побудови дерева папок та розподілу файлів між ними. Кожне дерево має налаштовану загальну кількість рівнів у глибину та директорій кожного рівня. При цьому, в кожній папці кожного рівня створюється такая сама кількість директорій, яка вказана в налаштуваннях. В отриманій структурі файли пропозицій зберігаються в найнижчому рівні, а їх шлях записується до відповідної таблиці бази даних. Використання такого алгоритму дозволяє значно зменшити швидкість доступу до файлів при великій їх кількості.

Іншою перевагою є можливість використання лінкування директорій, що дозволяє кожну з директорій розміщувати на різних фізичних носіях та навіть різних серверах.

Кожне з завдань отримання оновлених даних з серверів провайдерів пропозицій виконується за чітко визначеним розкладом. Для виконання задач, що потребують планування було створено модуль планувальника завдань. Основою цього модулю є відкрита система планування завдань Quartz.NET. Створений модуль надає можливість створення завдань, котрі будуть виконуватися за визначеним розкладом. Саме налаштування може виконуватися як у коді, створенням відповідних частин програми, так і в спеціальних файлах конфігурації. Сам синтаксис налаштування ідентичний планувальнику завдань Cron. Це дозволяє дуже точно налаштовувати час та розпорядок виконання кожного з завдань.

Весь обмін даними, що існує між головним сервером та масивом зчитувачів інформації з маркетплейсів, відбувається через відкритий брокер повідомлень RabbitMQ. Для полегшення використання цього інструменту було створено спеціалізований модуль серверу AZZCore. Брокер повідомлень надає можливості швидкого відправлення та отримання відповідних повідомлень. При цьому внутрішні сервіси RabbitMQ гарантують обов’язкову доставку повідомлення. Обмін даними відбувається через загальний exchange, в якому створюються черги повідомлень від головного сервера до кожного з сервісів-парсерів і навпаки. Це надає можливість серверу не очікувати відповіді кожного з мікросервісів, а обробляти відповідні дані лише по факту їх отримання. Іншою перевагою є можливість використання брокера повідомлень як балансувальника навантаження, що дозволить створити декілька екземплярів одного сервіса зчитування, які будуть одночасно отримувати дані різних типів пропозицій з одного майданчика. Також незаперечною перевагою є можливість винесення всього масиву мікросервісів на різні

хост-машини з мінімальною зміною налаштувань лише самого брокера повідомлень.

Для можливості спостереження за роботою головного сервера у будь-який момент часу, було створено модуль логування роботи на основі відкритої бібліотеки NLog.NET. Ця система дозволяє як з легкістю логувати всі типи повідомлень, які можуть виникати під час роботи сервера, так і працювати з великою кількістю провайдерів логування. До них входять: логування до консолі, логування до файлу та відправка логів через електронну пошту. Такий широкий спектр варіантів повністю покриває наші потреби. При цьому присутня зручна система налаштувань через файли конфігурації, яка дозволяє налаштовувати тип інформації, що збирається, її структуру, рівні повідомлень та цілі логування.

3.4 Компонент проксування запитів Nginx

Загальна безпека програмного продукту була одним із найбільших пріоритетів при проектуванні системи. Проксі-сервер є однією із ключових ланок її забезпечення. У нашому випадку використовується відкритий сервер nginx для фільтрації та обробки всієї інформації, якою виконується обмін між нашою системою та глобальною мережею. Це досягається за рахунок використання єдиної точки входу до системи, яка проходить через через налаштований проксі-сервер. При цьому з глобальної мережі неможливо отримати доступ напряму до СУБД, масиву парсерів та головного сервера. Таким чином забезпечується фільтрація невідповідного трафіка та захист від ddos атак. Додатковою перевагою nginx є можливість використання його як балансувальника навантаження.[11] Це дозволить розгортати одночасно декілька екземплярів всієї нашої системи, що надасть можливість, за потреби, різко збільшувати продуктивність системи.

Висновки

Аналіз ринку електронної торгівлі України показав:

- загальна кількість торговельних майданчиків в сфері пропозицій одягу наразі 12 компаній;
- існуючі майданчики потребують попередньої реєстрації та укладання договорів (Rozetka, Lamoda);
- відсутні маркетплейси, які надають повний доступ до програмних кодів використовуваних систем.

Виконане дослідження сфери пропозицій одягу доводить доцільність створення системи агрегації пропозицій товарів.

Набір технологій, використаних для реалізації проекту, забезпечує:

- можливість його масштабування на рівні окремих модулів та цілого застосунку;
- змогу опрацювання до тисячі одночасних запитів на один екземпляр системи;
- відкритість програмних кодів застосунку та його компонентів.

Розроблена система дозволяє зменшити час, який користувач витрачає на пошук необхідного товару, його порівняння з аналогами та ознайомлення з відгуками на 15-25%.

Під час виконання дипломної роботи було реалізовано набір основних модулів, що забезпечують роботу сервісу, та розроблено рішення для отримання даних пропозицій товарів.

Розгорнутий сервіс доступний за посиланням

Перелік посилань

1. Kestenbaum, Richard. What Are Online Marketplaces And What Is Their Future? [Електронний ресурс] – Режим доступу: https://www.forbes.com/sites/richardkestenbaum/2017/04/26/what-are-online-marketplaces-and-what-is-their-future/ (дата звернення 22.11.2021)
2. Online shopping [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Online_shopping (дата звернення 19.11.2021)
3. NET Goes Cross-Platform with .NET Core [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/archive/msdn-magazine/2016/april/net-core-net-goes-cross-platform-with-net-core (дата звернення 10.11.2021)
4. Troelsen, Andrew, Philip Japikse. Pro C# 7: With .NET and .NET Core 8th ed. Edition. 2017; P. 1316.
5. Adam Freeman. Pro ASP.NET Core 3 (Develop Cloud-Ready Web Applications Using MVC 3, Blazor, and Razor Pages) 8th ed. Edition. 2020; P. 1080.
6. Charles Anderson. Docker [Software engineering]. IEEE. 2015; 102 - c3
7. PostgreSQL 14.1 Documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/current/ (дата звернення 11.11.2021)
8. Entity Framework overview [Електронний ресурс] - Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview (дата звернення 17.11.2021)
9. AMQP 0-9-1 Model Explained [Електронний ресурс] - Режим доступу: https://www.rabbitmq.com/tutorials/amqp-concepts.html (дата звернення 18.2.2022)
10. Representational State Transfer (REST) [Електронний ресурс] - Режим доступу: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (дата звернення 19.2.2022)
11. Module ngx_http_upstream_module [Електронний ресурс] - Режим доступу: https://nginx.org/en/docs/http/ngx_http_upstream_module.html (дата звернення 15.2.2022)

ДОДАТКИ

Додаток А

Модуль створення файлового середовища проекту

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using ASSCore.Models.AppSettingsModels;
using IdentityServer4.Extensions;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace ASSCore.FileEnvironment.InitialFileEnvironmentPreparation
{
    public static class FileEnvironmentPreparer
    {
        public static async Task PrepareAsync(IApplicationBuilder app)
        {
            using var serviceScope = app.ApplicationServices.CreateScope();
            var configuration =
serviceScope.ServiceProvider.GetService<IConfiguration>();
            var logger =
serviceScope.ServiceProvider.GetService<ILogger<FileEnvironmentContext>>();

            //Block for calling custom preparers
            await CreateFolderStructure(configuration, logger);

        }

        private static async Task CreateFolderStructure(IConfiguration
configuration, ILogger logger)
        {
            var options = new FileStoreOptions();
            configuration.GetSection(FileStoreOptions.FileStore).Bind(options);
            var pathToInfoFile = Path.Combine(options.StorePath, "StoreInfo");

            if (!Directory.Exists(options.StorePath))
                throw new ArgumentException("FileStore.StorePath is not a directory
or not exists");

            // Check if filestore structure exists and the settings are the same
            if (!await IsFileStructureExistsAsync(options, pathToInfoFile))
            {
                var coreDirPaths = Enumerable
                    .Repeat(options.StorePath,
options.DirectoriesNumberAtLevel).ToList();
                for (var i = 0; i < options.NestingLevel; i++)
                {
                    var temp = new List<string>();
                    for (var k = 1; k < options.DirectoriesNumberAtLevel + 1; k++)
```

```
                    {
                        temp.AddRange(coreDirPaths.Select(dirPath =>
                            Directory.CreateDirectory(Path.Combine(dirPath,
k.ToString())).FullName));
                    }
                    coreDirPaths = temp.ToList();
                }

                string[] textToWrite = {$"StructureVersion:
{options.StructureVersion}"};
                await File.WriteAllLinesAsync(pathToInfoFile, textToWrite);

                logger.LogInformation("CreateFolderStructure execution finished.
File structure created successfully");
                return;
            }

            logger.LogInformation("CreateFolderStructure execution finished. File
structure already created, creation skipped");

            static async Task<bool> IsFileStructureExistsAsync(FileStoreOptions
options, string pathToInfoFile)
            {
                if (!File.Exists(pathToInfoFile))
                    return false;

                var fileData = await File.ReadAllLinesAsync(pathToInfoFile);
                if (fileData.IsNullOrEmpty())
                    return false;
                var storeVersion = fileData[0].Split(":")[1];
                return int.Parse(storeVersion) == options.StructureVersion;
            }
        }
    }
}
```

Додаток Б

Модель збереженого файлу

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ASSCore.Models
{
    public class StoredFile
    {
        [Key]
        [Required]
        public string Id { get; set; }

        /// <summary>
        /// Path to file in file storage
        /// </summary>
        [Required]
        public string Path { get; set; }

        /// <summary>
        /// File sha256 hash as base64 string
        /// </summary>
        [Required]
        public string Hash { get; set; }

        /// <summary>
        /// Filename extension. An identifier specified as a suffix to the
name of a computer file
        /// </summary>
        [Required]
        public string Extension { get; set; }

        /// <summary>
        ///
        /// </summary>
        [Required]
        public bool IsDeleted { get; set; }

        [Required]
        public DateTime CreateDate { get; set; }

        public DateTime DeletedDate { get; set; }
    }
}
```

Додаток В

Модель продукту

```
// ReSharper disable UnusedAutoPropertyAccessor.Global
// ReSharper disable PropertyCanBeMadeInitOnly.Global
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ASSCore.Models.Scraper.Entities
{
    public class Product
    {
        [Key]
        [Required]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public string Id { get; set; }

        /// <summary>
        /// Full product name
        /// </summary>
        [Required]
        public string Title { get; set; }

        /// <summary>
        /// Type of clothing to which the product belongs
        /// </summary>
        [Required]
        public string Type { get; set; }

        /// <summary>
        /// Collection to which the product belongs
        /// </summary>
        [Required]
        public string Category { get; set; }

        /// <summary>
        /// Product manufacturer
        /// </summary>
        [Required]
        public string Company { get; set; }

        /// <summary>
        /// product price
        /// </summary>
        [Required]
        public string Price { get; set; }

        /// <summary>
        /// Link to the product page on the manufacturer's website
        /// </summary>
        [Required]
        public string Href { get; set; }
    }
}
```

Додаток Г

Модуль початкової ініціалізації бази даних

```
using System;
using System.Linq;
using System.Threading.Tasks;
using ASSCore.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace ASSCore.Data.InitialDbPreparation
{
    public static class DbPreparer
    {
        /// <summary>
        ///     Creates the minimum necessary database payload for the server to
function properly
        /// </summary>
        public static async Task PrepareAsync(IApplicationBuilder app)
        {
            using var serviceScope = app.ApplicationServices.CreateScope();
            var db = serviceScope.ServiceProvider.GetService<ApplicationContext>();
            var logger =
serviceScope.ServiceProvider.GetService<ILogger<ApplicationContext>>();

            //Block for calling custom preparers
            ApplyMigrations(db, logger);
            SeedUsers(db, logger);
            //await SeedModelNameAsync(db, logger);
        }

        /// <summary>
        /// Adds new user entries to the database if the users table is empty
        /// </summary>
        /// <param name="dbContext">Database context from Entity framework</param>
        /// <param name="logger">Entity used to perform logging</param>
        private static void SeedUsers(ApplicationContext dbContext, ILogger logger)
        {
            logger.LogDebug("Start SeedUsersAsync execution");
            if (!dbContext.Users.Any())
            {
                dbContext.Users.Add(new User
                    {Locale = "UA", UserName = "Vasiliy", Email = "vacya@mail.ukr",
EmailConfirmed = true, PhoneNumber = "123123123", PhoneNumberConfirmed = true,
TwoFactorEnabled = false, LockoutEnabled = false, AccessFailedCount = 0});

                dbContext.SaveChanges();

                logger.LogDebug("SeedUsersAsync execution finished. Default users
were added successfully");
                return;
            }
```

```
            logger.LogDebug("SeedUsersAsync execution finished. There are already
users in the database, so no new ones have been added");
        }

        /// <summary>
        /// Applies all created but not executed migrations to the database
        /// </summary>
        /// <param name="dbContext">Database context from Entity framework</param>
        /// <param name="logger">Entity used to perform logging</param>
        private static void ApplyMigrations(ApplicationContext dbContext, ILogger
logger)
        {
            logger.LogDebug("Start ApplyMigrationsAsync execution");
            if (dbContext.Database.GetPendingMigrations().Any())
            {
                dbContext.Database.Migrate();
            }

        }
    }
}
```

Додаток Д

Файл конфігурації логера

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      throwConfigExceptions="true"
      autoReload="true"
      internalLogLevel="Info"
      internalLogFile="/var/log/azzcore/internal-nlog-AspNetCore.txt">

    <variable name="logFileName" value="/var/log/azzcore/azzCore-log-
${shortdate}.log"/>

    <!-- enable asp.net core layout renderers -->
    <extensions>
        <add assembly="NLog.Web.AspNetCore"/>
    </extensions>

    <!-- the targets to write to -->
    <targets>
        <!-- File Target for all log messages with full details -->
        <target xsi:type="File" name="allfile" fileName="${logFileName}"
                layout="${longdate}|${event-
properties:item=EventId_Id:whenEmpty=0}|${uppercase:${level}}|${logger}|${message}
${exception:format=tostring}" />

        <!--Console Target for all log messages with basic details -->
        <target xsi:type="Console" name="allConsole"
                layout="${level:lowercase=true}: ${logger}[0]${newline}
${message}${exception:format=tostring}" />
    </targets>

    <!-- rules to map from logger name to target -->
    <rules>
        <!--All logs, including from Microsoft-->
        <logger name="*" minlevel="Trace" writeTo="allfile" />
        <logger name="*" minlevel="Trace" writeTo="allConsole"/>
    </rules>
</nlog>
```

Додаток Е

Модуль зчитувача даних про товари Dr. Martens

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using hm_scrapper.Lib.DTO;
using HtmlAgilityPack;

namespace hm_scrapper.Lib.Scraper
{
    public class HMScraper
    {
        public static async Task<List<ProductDto>> Process()
        {
            const string url = @"https://www.drmartens.com";
            const string productPath = @"/us/en/mens/boots/c/02010000";

            var scraper = new HtmlWeb();
            var htmlDocument = await scraper.LoadFromWebAsync(url + productPath);
            var products =
                htmlDocument.DocumentNode.SelectNodes("//div[contains(@class,
'product__list__item__details')]");

            return products.Select(product => new ProductDto
                {
                    href = url +
product.SelectSingleNode("./a").Attributes["href"].Value,
                    price = product.SelectSingleNode(".//span[contains(@class,
'current-price')]").InnerHtml,
                    title = product.SelectSingleNode("./a").InnerHtml,
                    category = "men's boots",
                    type = "shoes",
                    company = "Dr. Martens"
                })
                .ToList();
        }
    }
}
```

Додаток Ж

Модуль зчитувача даних про товари Adidas

```
const IScraper = require('./IScraper')
const axios = require('axios')
const cheerio = require('cheerio')

class Scraper extends IScraper {
  /**
   * Scrap products data
   * @returns {Promise<ProductData[]>}
   */
  static async Process() {
    const url = 'https://www.adidas.ua'
    const type = '/vzuttya'
    /**
     * @type {ProductData[]}
     */
    const result = []

    const response = await axios.get(url + type)
    if (response.status !== 200) {
      throw new Error(`Unsuccessful call to server. Response code
${response.status}`)
    }

    const $ = cheerio.load(response.data)
    $('a.product__info').each(function (i, el) {
      const current = $(this)
      result.push({
        href: url + current.attr('href'),
        price: current.find('.price__first').text(),
        title: current.find('.product__title').text(),
        category: current.find('.product__category').text(),
        type: 'shoes',
        company: 'adidas'
      })
    })

    return result
  }
}

module.exports = Scraper
```

Додаток К

Модуль контексту RabbitMQ

```
using System;
using RabbitMQ.Client;
using Microsoft.Extensions.Logging;
using System.Collections.Concurrent;
using ASSCore.Models.AppSettings;
using Microsoft.Extensions.Configuration;
using RabbitMQ.Client.Events;

namespace ASSCore.RabbitMq
{
    public class RabbitMqContext : IRabbitMqContext
    {
        private readonly IModel _channel;
        private readonly IConnection _connection;
        private readonly ILogger<RabbitMqContext> _logger;
        private readonly RabbitMqOptions _rabbitMqOptions;
        private readonly BlockingCollection<byte[]> _respQueue = new();

        private string ReplyQueueName { get; set; }
        private string CorrelationId { get; set; }
        private bool IsRpcConfigured { get; set; }


        ~RabbitMqContext()
        {
            _channel.Dispose();
            _connection.Dispose();
        }

        public RabbitMqContext(IConfiguration configuration,
ILogger<RabbitMqContext> logger)
        {
            _logger = logger;
            _rabbitMqOptions = new RabbitMqOptions();

configuration.GetSection(_rabbitMqOptions.BlockName).Bind(_rabbitMqOptions);

            var factory = new ConnectionFactory
            {
                HostName = _rabbitMqOptions.HostName,
                Port = _rabbitMqOptions.Port
            };
            _connection = factory.CreateConnection();
            _channel = _connection.CreateModel();

            _channel.ExchangeDeclare(
                _rabbitMqOptions.Exchange,
                _rabbitMqOptions.ExchangeType,
                _rabbitMqOptions.ExchangeProperties.Durable,
                _rabbitMqOptions.ExchangeProperties.AutoDelete
            );
        }

        public void Publish(string routingKey, IBasicProperties basicProperties,
byte[] body)
```

```
        {
            _channel.BasicPublish(
                _rabbitMqOptions.Exchange,
                routingKey,
                basicProperties,
                body
            );
        }

        public byte[] Call(string routingKey, IBasicProperties basicProperties,
byte[] body)
        {
            if (!IsRpcConfigured)
            {
                _logger.LogInformation("RPC is not configured. Configuring...");
                ConfigureRpcClient();
                _logger.LogInformation("RPC configured successfully");
            }

            basicProperties ??= _channel.CreateBasicProperties();
            basicProperties.CorrelationId ??= CorrelationId;
            basicProperties.ReplyTo ??= ReplyQueueName;

            _logger.LogInformation("Processing remote procedure call");
            _channel.BasicPublish(
                _rabbitMqOptions.Exchange,
                routingKey,
                basicProperties,
                body
            );
            _logger.LogInformation("RPC processed successfully");

            return _respQueue.Take();
        }

        private void ConfigureRpcClient()
        {
            ReplyQueueName = _channel.QueueDeclare().QueueName;
            CorrelationId = Guid.NewGuid().ToString();

            var consumer = new EventingBasicConsumer(_channel);
            consumer.Received += (_, ea) =>
            {
                if (ea.BasicProperties.CorrelationId != CorrelationId)
                {
                    return;
                }

                var body = ea.Body.ToArray();
                _respQueue.Add(body);
            };

            _channel.BasicConsume(
                consumer: consumer,
                queue: ReplyQueueName,
                autoAck: true);

            IsRpcConfigured = true;
        }
```

```
}}
```