Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«Розробка системи для збору та обробки даних з використанням Apache Spark»**

Виконав: студент 4-го року навчання,

спеціальності

121 «Інженерія Програмного Забезпечення»

Пінкевич В.М.

Керівник

Старший викладач

Борозенний С.О.

_ травня 2023 р.

Київ – 2023

Календарний план виконання роботи

Тема: Розробка системи для збору та обробки даних з використанням Apache Spark

Календарний план виконання роботи:

№ п/п	Назва етапу кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання теми роботи	Жовтень 2022 р.	
2.	Отримання завдання на кваліфікаційну роботу	Листопад 2022 р.	
3.	Ознайомлення із літературою та джерелами за темою роботи	Листопад - грудень 2022 р.	
4.	Знайомство із засобами розробки системи	Грудень 2022 р. - січень 2023 р.	
5.	Розробка застосунку для збору даних	Січень – лютий 2023 р.	
6.	Розробка застосунку для обробки даних	Березень - квітень 2023 р.	
7.	Написання текстової частини роботи	Квітень - травень 2023 р.	
8.	Перегляд кваліфікаційної роботи науковим керівником	Квітень - травень 2023 р.	
9.	Внесення змін до роботи згідно з зауваженнями наукового керівника	Квітень - травень 2023 р.	
10.	Захист кваліфікаційної роботи	Червень 2023 р.	

Студент Пінкевич В.М.

Керівник Борозенний С.О.

Зміст

Зміст 1

Анотація...3

Використані скорочення 4

Вступ 6

Розділ 1. Огляд джерел отримання даних...8

1.1 Steam 8

1.2 Steam Spy 11

Розділ 2. Створення архітектури системи...13

2.1 Основні компоненти системи 13

2.2 Формат зберігання даних..15

2.3 Загальні принципи роботи системи...18

Розділ 3. Розробка застосунку для збору даних 20

3.1 Вибір технологій розробки 20

3.2 Steam API..21

3.3 Steam Spy API 23

3.4 Розробка застосунку..24

Розділ 4. Розробка застосунку для обробки даних...................................32

4.1 Вибір технологій розробки 32

4.2 Структура бази даних...34

4.3 Розробка застосунку..37

Розділ 5. Аналіз виконаної роботи 47

5.1 Можливості розвитку розробленої системи 47

5.2 Висновки 47

Список використаних джерел 49

Додатки 51

Додаток А. Клас WebClientConfiguration 51

Додаток Б. Клас SteamService..52

Додаток В. Клас SteamAppDetailsResponseDeserializer 53

Додаток Г. Клас SteamDataCsvFileWriter...54

Додаток Д. Клас SteamAppDetailsProcessor 57

Додаток Е. Файлова структура проекту застосунку для збору даних 58

Додаток Є. Файлова структура проекту застосунку для обробки даних 60

Анотація

У роботі описується процес розробки системи для отримання та обробки даних. Розглядаються можливі варіанти реалізації такої системи, їх переваги та недоліки, обрані інструменти розробки, їх особливості та основні можливості. Описуються деталі реалізації системи, можливості розширення та покращення розробленого рішення. Також проводиться аналіз основних проблем, які виникають при розробці таких систем, та основні способи їх вирішення.

Використані скорочення

API (Application Programming Interface) – засоби, які надає система і які можуть бути використані іншими системами для взаємодії із даною системою.

JSON (JavaScript Object Notation) – текстовий формат зберігання та передачі даних, який базується на синтаксисі та форматі об’єктів мови JavaScript.

CSV (Comma Separated Values) – текстовий формат зберігання та передачі даних, який зберігає інформацію у вигляді таблиць що складаються зі стовпців та рядків.

XML (eXtensible Markup Language) – текстовий формат зберігання та передачі даних, який базується на системі тегів.

СКБД (система керування базами даних) – програма або набір програм, які забезпечують контроль над доступом до бази даних і надають можливості для пошуку, додавання, оновлення та видалення даних із бази даних.

SQL (Structured Query Language) – декларативна мова програмування, яка надає можливості для керування базами даних а також інформацією, що зберігається у базах даних.

HTTP (HyperText Transfer Protocol) – мережевий протокол прикладного рівня, який забезпечує комунікацію та передачу даних між пристроями у мережі.

URL (Uniform Resource Locator) – стандартизована адреса унікального ресурсу в мережі.

JVM (Java Virtual Machine) – абстрактна машина, специфікація, яка описує, як має відбуватися виконання програм, написаних мовою Java, або програм, написаних іншими мовами програмування та скомпільованих у байт-код Java. Конкретні реалізації JVM забезпечують виконання байт-коду на різних платформах та кінцевих пристроях.

JDBC (Java Database Connectivity) – специфікація та інтерфейс (API), який надає можливість програмам, написаним мовою Java, взаємодіяти з різними СКБД.

UDF (User Defined Function) – функція, що створюється розробником і може бути зареєстрована та використана у фреймворку Apache Spark так, ніби вона є його частиною.

Вступ

Основою роботи будь-якої системи є дані. Будь-яка система у тому чи іншому вигляді працює із даними. Вона може їх створювати, обробляти, надавати іншим системам, тощо. Часто для повноцінної роботи системі потрібні не лише ті дані, які вона створює, а й сторонні, зібрані іншими системами. Ці дані можуть бути як публічно доступними і не вимагати жодних умов для використання, так і закритими, коли отримання доступу до них має свої особливості. Не менш важливою частиною багатьох систем є також обробка зібраних даних. Часто об’єм цих даних є настільки великим, що для виконання повної їх обробки необхідна велика кількість ресурсів.

Для вивчення особливостей роботи таких систем (а також кожного з етапів роботи з даними) було вирішено розробити власну систему, яка буде отримувати дані із відкритих джерел, проводити їх обробку та зберігати вже оброблені дані.

Об’єктом дослідження є вивчення принципів розробки та функціонування систем збору, обробки та зберігання великої кількості даних.

Предметом дослідження є розробка системи, яка буде взаємодіяти зі сторонніми сервісами через публічні веб API для отримання даних, виконувати очищення, обробку даних та зберігати оброблені дані для їх подальшого використання.

Метою роботи є створення системи, яка буде отримувати дані від сторонніх сервісів Steam та Steam Spy через публічні веб API, проводити їх очистку, трансформацію та зберігати оброблені дані у реляційній базі даних. Через взаємодію зі сторонніми публічними веб API буде досліджено особливості взаємодії із такими API. Використання фреймворку Apache Spark дозволить розглянути основні підходи до роботи із великою кількістю даних та дослідити можливу реалізацію цих підходів на прикладі обраного фреймворку.

Текстова частина роботи складається із п’яти розділів.

У першому розділі виконується огляд сервісів Steam та Steam Spy, API яких будуть використані для отримання даних.

Другий розділ фокусується на розробці архітектури системи. Проводиться огляд декількох можливих варіантів реалізації системи, їх переваг та недоліків, а також вибір фінального рішення.

Третій та четвертий розділи описують безпосередньо розробку обох частин системи: для збору та обробки даних відповідно. Для кожної частини обираються інструменти розробки, описуються їх переваги та недоліки, розглядаються деталі реалізації та проблеми, що виникали в ході розробки, і шляхи їх вирішення. Третій розділ також включає огляд особливостей при роботі зі сторонніми публічними веб API, а четвертий – особливості роботи з великими даними.

У п’ятому розділі підсумовуються результати розробки системи та аналізуються можливості покращення та подальшого розширення розробленої системи.

Розділ 1. Огляд джерел отримання даних

1.1 Steam

Steam, за визначенням своїх розробників – це “найбільше і найкраще місце для грання, обговорення та створення відеоігор”. [1]

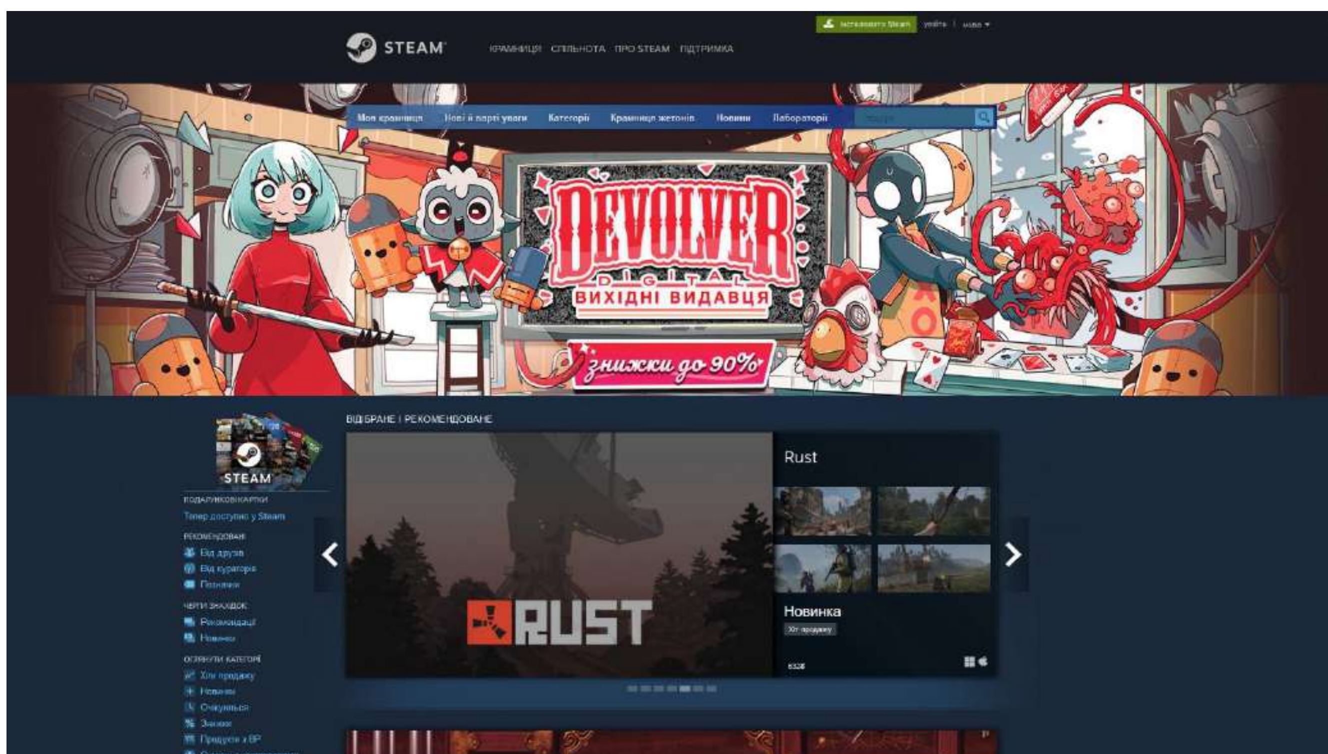


Рисунок 1.1.1. Головна сторінка вебсайту сервісу Steam

Якщо спочатку це був онлайн-сервіс для продажу відеоігор, то тепер це ціла окрема платформа, яка забезпечує велику кількість функцій як споживачам, так і розробникам відеоігор. Основні можливості, які Steam надає користувачам, включають:

- можливість покупки (чи безкоштовного отримання) відеігор, доповнень до них, саундтреків, програмного забезпеченням та їх зберігання у власній бібліотеці користувача;
- наявність детальної інформації про акаунт користувача, статистики користування сервісом (час, проведений у кожній грі, отримані досягнення, нещодавня активність), можливості його персоналізації та

розвитку (наявність системи карток, значків, способів оформлення власного профілю);

- розвинена система взаємодії між користувачами платформи. Вони можуть писати рецензії на ігри, рекомендувати чи не рекомендувати їх іншим користувачам, створювати власні обговорення на окремих форумах, створювати фото та відео за мотивами ігор, модифікації для них, писати інструкції та підказки, тощо. Тобто існують досить широкі можливості не лише для грання, а й для створення користувацького контенту, який стосується ігор, проте не є їх частиною, задуманою розробниками;
- наявність власної соціальної системи. Користувачі можуть додавати один одного до списку друзів, слідкувати за оновленнями та прогресом в іграх, грати в ігри разом, тощо;
- наявність внутрішнього ринку. Багато ігор надають можливість отримання, покупки та продажу внутрішньоігрових предметів. А Steam має власний ринок, який дозволяє користувачам виставляти предмети на продаж за бажаною ціною, купувати їх чи обмінюватись ними.

Розробникам Steam також надає велику кількість можливостей, серед них:

- повний контроль над власними продуктами у Steam. Розробник може виставляти власні ігри на продаж, влаштовувати акції та розпродажі, піднімати чи зменшувати ціну на гру, керувати додатковими матеріалами (фото, відео, опис, системні вимоги, тощо);
- наявність аналітичних даних щодо власних продуктів. Розробник має можливість спостерігати за кількістю проданих копій гри від часу її додавання, інформацію про кінцевих користувачів, які купили гру, поточну кількість гравців, які грають у гру;

- можливості для взаємодії із гравцями. Розробник може публікувати новини, що стуються гри, проводити прямі трансляції, створювати власний сторонній контент для гри, бачити рецензії та контент, розроблений гравцями. Це дозволяє розробникам контактувати зі своєю аудиторією, отримувати зворотній зв'язок та покращувати якість власних продуктів.

Саме завдяки наявності великої кількості можливостей не лише для покупки ігор, а й для їх обговорення, створення стороннього контенту та взаємодії кінцевих користувачів як між собою, так і напряму із розробниками, Steam є найпопулярнішим сервісом для покупки цифрових копій ігор на сьогоднішній день.

Steam має декілька різних клієнтів. На даний момент існує вебсайт, доступний через браузери, застосунки для операційних систем Windows, macOS та Linux, додатки для мобільних операційних систем Android та iOS, а також власна операційна система SteamOS (яка базується на операційній системі Debian сімейства Linux), яка уже при встановленні включає клієнт Steam та широко інтегрована з ним. Усі основні функції Steam доступні на будь-якому його клієнті, проте кожен має свої особливі функції. Наприклад, безпосередньо завантажити та грати в ігри зі своєї бібліотеки можна лише через настільні застосунки, а двофакторна аутентифікація доступна лише при використанні мобільних додатків.

Steam було обрано для отримання інформації про ігри саме через те, що він є найбільшим сервісом для продажу ігор, і, відповідно має інформацію про найбільшу кількість ігор, а також через наявність публічного API, який дозволяє отримувати дані без необхідності додаткових дій з боку розробника.

1.2 Steam Spy

Steam Spy – це статистичний сервіс для Steam, який базується на API, який надає компанія Valve (компанія, що розробляє Steam). Він автоматично збирає інформацію із користувацьких профілів Steam, аналізує її та дозволяє переглядати її у простому та зручному форматі. [2]

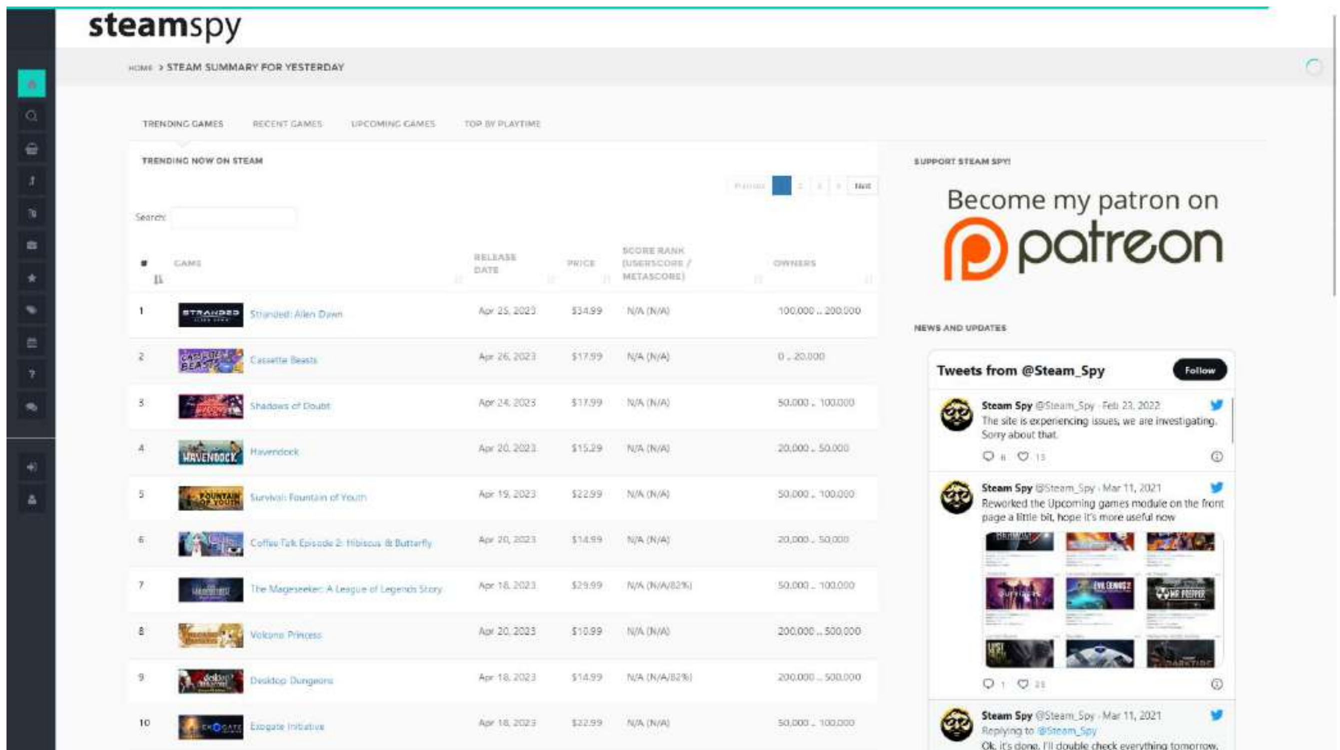


Рисунок 1.2.1. Головна сторінка вебсайту Steam Spy

Основною метою сервісу є отримання доступної інформації через API сервісу Steam та її аналіз, що дозволяє кінцевим користувачам сервісу отримувати найактуальнішу інформацію про наявні у Steam ігри. Крім перегляду звичайної інформації про ігри, яку з легкістю можна отримати у самому Steam, Steam Spy також надає дані про приблизну кількість проданих копій гри, кількість позитивних та негативних оцінок від користувачів, середній проведений час у грі, тощо. Також Steam Spy агрегує дані щодо розподілу ігор за різними критеріями: розробниками, видавцями, доступними мовами, жанрами, тегами, поточними знижками та розпродажами. Більше того, для кожного критерію сервіс надає різноманітну специфічну статистику. Наприклад, для кожного видавця доступна

інформація про загальну кількість його ігор у Steam, загальну кількість проданих копій усіх його ігор, середню ціну гри, середній проведений час у грі.

Як зазначають розробники Steam Spy, дані сервісу, особливо щодо кількості проданих копій ігор, не є точними і надають лише приблизну оцінку. Також у даних можуть виникати деякі аномальні значення, пов’язані із принципами роботи Steam. Наприклад, Steam деколи проводить “безкоштовні вихідні” – період часу, протягом якого деякі ігри можуть бути отримані кінцевими користувачами безкоштовно на цей період часу. Протягом цього періоду Steam позначає кожного кінцевого користувача, який хоча б один раз авторизувався в сервісі, власником цих безкоштовних ігор, а після закінчення цього періоду – повертається до попереднього стану. Дані щодо кількості власників гри Steam надає у публічному API, тому при отриманні даних щодо кількості власників гри у періоди “безкоштовних вихідних” значення може значно відрізнятись від того, яке було отримано поза цими періодами. [2]

Steam Spy було обрано додатковим джерелом інформації про ігри у Steam, оскільки цей сервіс дозволяє отримати дані, які важко знайти використовуючи лише API Steam, або дані, яких там взагалі немає і для отримання яких потрібно виконувати власний аналіз. Важливим фактором також є доступний публічний API, який дозволяє отримувати усю необхідну інформацію.

Розділ 2. Створення архітектури системи

2.1 Основні компоненти системи

Основні задачі, які має виконувати система – це отримання даних від описаних сервісів через публічні API, подальша їх обробка, очищення, трансформація та збереження. На перший погляд, для розробки такої системи достатньо розробити один застосунок, який буде виконувати усі необхідні задачі. Проте, при використанні такого підходу з'являється декілька проблем:

- сильна зв'язаність частин системи, які відповідають за отримання та обробку даних. Частина системи, яка відповідає за отримання даних, не має залежати від тієї частини, яка відповідає за їх обробку і навпаки. При внесенні змін в одну із частин, інша, з великою ймовірністю, також потребуватиме відповідних змін;
- декілька точок відмови. Наприклад, при отриманні даних зі сторонніх публічних API немає жодної впевненості в тому, чи будуть вони працювати в певний момент часу. Також якщо будуть виникати помилки при обробці чи збереженні даних, наслідком яких стане завершення роботи застосунку, частина системи, яка відповідає за отримання даних, також завершить свою роботу, хоча в її роботі не було жодних помилок;
- неефективний розподіл ресурсів. Якщо проаналізувати як має працювати система, то зрозуміло, що при обробці та збереженні даних найбільш потрібними ресурсами є процесор та оперативна пам'ять. Натомість, отримання даних від сторонніх API не потребує великої кількості ресурсів процесора чи оперативної пам'яті, оскільки жодних складних операцій не виконується, адже найбільша кількість часу витрачається на процеси введення-виведення, а саме – на мережеву комунікацію та очікування відповіді від стороннього сервера. Таким чином, при

реалізації усієї системи у вигляді єдиного застосунку досить важко буде реалізувати ефективний розподіл ресурсів між її частинами.

Проаналізувавши принципи роботи системи та описані проблеми, цілком логічним здається розподіл системи на дві частини: частину для отримання інформації від сторонніх API та частину для обробки отриманих даних та їх збереження, причому кожну частину можна реалізувати у вигляді окремого застосунку. Такий розподіл має декілька переваг, серед них:

- незалежні системи отримання та обробки даних. Їх розділення дозволяє вносити зміни до кожної системи окремо, не впливаючи на іншу;
- свобода у виборі технологій. Для кожного застосунку можна обрати набір технологій, який найкраще підійде для вирішення саме його задач;
- вища відмовостійкість. Через незалежність обох застосунків проблеми в роботі одного з них не впливають на роботу іншого, а при завершенні роботи одного з них інший продовжує працювати;
- можливості налаштування та керування ресурсами. Для кожного застосунку можна налаштовувати необхідну йому кількість ресурсів, забезпечуючи ефективне їх використання;
- зручність розробки та тестування. Кожен застосунок розгортається окремо, а тому для перевірки правильності роботи одного з них не потрібно розгортати інший, оскільки він не потрібний для тестування.

При розділенні системи на два окремих застосунки виникає потреба у забезпеченні процесу доставки даних від застосунку, який їх отримує, до застосунку, який займається їх обробкою. Для цього можна розглянути декілька варіантів:

- передача даних у вигляді зберігання проміжних результатів у файлах. Найбільш простий варіант, оскільки не вимагає використання жодних сторонніх систем, проте досить гнучкий, оскільки формат для зберігання можна обрати будь-який: JSON, CSV, XML, тощо;
- використання реляційної бази даних для зберігання проміжних результатів. Не дуже вдалий варіант, оскільки отримані дані можуть мати різну структуру, а реляційні бази даних не є найкращим вибором для роботи з такими даними;
- використання брокерів повідомлень для передачі проміжних даних. Такий варіант буде найкращим вибором при розробці системи, яка має отримувати, обробляти та зберігати дані в реальному часі, оскільки дозволяє обробити та зберегти нові дані до системи одразу після їх отримання через брокер повідомлень.

Оскільки система не повинна працювати із даними у реальному часі, то використання брокера повідомлень буде надлишковим, а використання бази даних для зберігання проміжних даних, які після обробки вже не будуть потрібними, є не дуже ефективним та потребує розгортання додаткового застосунку - СКБД. Таким чином, найкращим вибором буде зберігання усіх проміжних даних у файлах.

2.2 Формат зберігання даних

Для зберігання уже оброблених та трансформованих даних необхідно використати певне сховище. Оскільки кінцеві дані можуть бути використані в подальшому для виконання певного аналізу, підрахунку статистики, то найкращим рішенням буде зберігати дані у реляційній базі даних. Основними перевагами такого підходу є:

- зберігання не лише даних, а і зв'язків між ними. Основною особливістю реляційних баз даних є саме можливість будувати зв'язки між даними для подальшого їх використання;
- можливості контролю за вхідними даними. Більшість СКБД підтримують функціонал для створення певних перевірок вхідних даних на основі умов, заданих розробником, що дозволяє уникнути зберігання неправильних даних;
- можливості для редагування та видалення існуючих даних. При збереженні даних у файлах із цим можуть бути проблеми, оскільки зазвичай розробнику потрібно буде створювати власне рішення для правильного оновлення чи видалення даних із файлу. А СКБД надають ці можливості одразу, без необхідності додаткових дій з боку розробника;
- підтримка мови SQL. Керування базою даних через СКБД відбувається за допомогою мови SQL, яка є потужним інструментом для роботи із даними. Крім додавання, редагування та видалення даних, вона має широкі можливості для пошуку необхідних даних та їх аналізу.

Робота із базою даних зазвичай відбувається через СКБД. Саме СКБД є точкою входу в базу даних та відповідає за усі операції, які клієнт хоче виконати. СКБД відрізняються між собою стандартами мови SQL, які вони підтримують, наявністю специфічних можливостей та розширень SQL, таких як власні типи даних чи оператори, продуктивністю, тощо. Для зберігання даних у системі було обрано СКБД MySQL. MySQL – це найпопулярніша реляційна база даних із відкритим вихідним кодом, що розробляється, поширюється та підтримується компанією Oracle. Основними перевагами MySQL є: [3]

- повна підтримка багатьох стандартів мови SQL;
- наявність великої кількості типів даних як фіксованого, так і змінного розміру;

- висока продуктивність та ефективність завдяки можливості використання багатоядерних процесорів або декількох процесорів одночасно (якщо такі доступні на машині, на яку встановлено СКБД);
- підтримка найбільших операційних систем, таких як Windows, macOS та Linux;
- можливість використовувати двигуни як з підтримкою транзакцій, так і без неї;
- підтримка баз даних і таблиць із дуже великою кількістю даних;
- наявність драйверів та клієнтів для багатьох мов програмування;
- широкі можливості для налаштування та легкість розгортання.

Враховуючи усі вищезазначені переваги, використання СКБД MySQL для зберігання даних у системі є цілком доречним рішенням.

Для розгортання СКБД буде використано Docker. Docker – це інструмент для контейнеризації будь-яких застосунків. Він дозволяє розділити програми та інфраструктуру, яка потрібна для їх виконання, забезпечуючи ізоляцію програми від інфраструктури через використання контейнерів. [4] Використання Docker дозволяє уникнути необхідності розгортання СКБД та усіх її залежностей на локальній машині. Щоб уникнути необхідності встановлювати одні й ті ж налаштування для образу СКБД при кожному її запуску буде використано інструмент Docker Compose, який є частиною Docker. Встановлені налаштування образу СКБД через Docker Compose можна побачити на рисунку 2.2.1.

docker-compose.yml

C: > Users > Victor > IdeaProjects > diploma-work > docker-compose.yml

```yaml
version: '3'

services:

  mysql:
    container_name: mysql-diploma-work
    image: mysql:8.0.32
    restart: always
    ports:
      - "3306:3306"
    environment:
      MYSQL_DATABASE: diploma-work
      MYSQL_ROOT_PASSWORD: password
    volumes:
      - mysql-data:/var/lib/mysql

volumes:
  mysql-data:
    driver: local
    driver_opts:
      type: "none"
      o: "bind"
      device: "D:/University/Subjects/Рік 4/Дипломна робота/mysql/mysql-data"

```

Рисунок 2.2.1. Налаштування образу СКБД MySQL через Docker Compose

2.3 Загальні принципи роботи системи

Загальний процес роботи системи можна описати покроково:

1. Отримання даних від сторонніх API: Steam API та Steam Spy API.
2. Запис отриманих даних в окремі файли для їх подальшої обробки.
3. Зчитування даних із файлів для їх подальшої обробки.
4. Очищення та трансформація даних.
5. Збереження перетворених даних у базу даних.

Оскільки система складається із двох застосунків: для отримання даних та для їх обробки, то, відповідно, перший застосунок буде виконувати кроки 1 та 2, а

другий – кроки 3, 4 та 5. Структуру системи та взаємодію її частин можна побачити на рисунку 2.3.1:

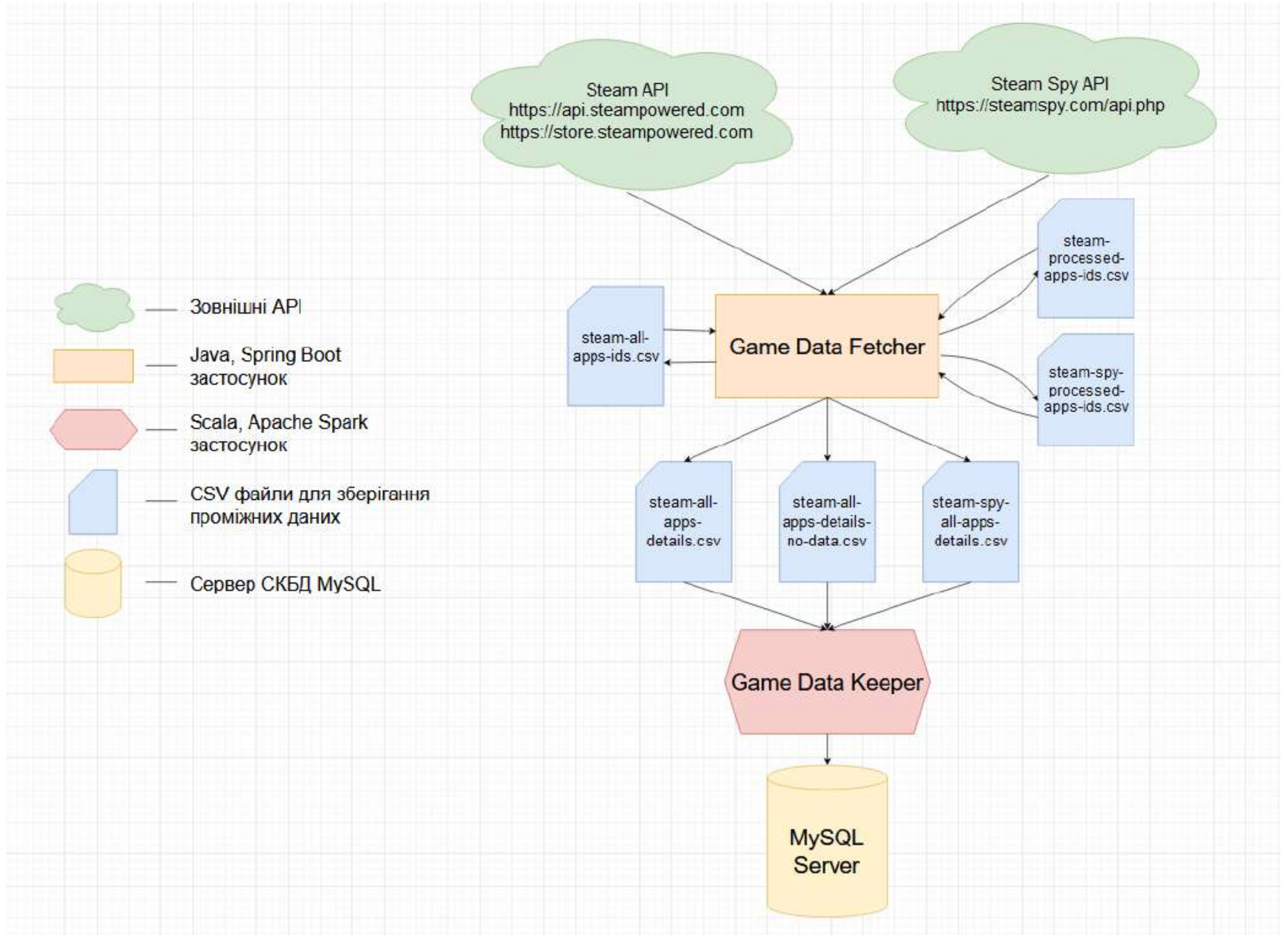


Рисунок 2.3.1. Архітектура системи

Розділ 3. Розробка застосунку для збору даних

3.1 Вибір технологій розробки

Основними завданнями застосунку є отримання даних від сторонніх API у форматі JSON, перетворення їх у формат CSV та збереження у файли. Для їх виконання найкраще буде використати:

- HTTP-клієнт, який дозволяє робити мережеві запити. Він забезпечить отримання даних від сторонніх API;
- засоби для роботи із даними у форматі JSON, оскільки сторонні API надають дані саме у форматі JSON;
- засоби для роботи із даними у форматі CSV - для збереження даних у проміжні CSV файли.

Для реалізації застосунку було обрано мову програмування Java, оскільки вона має велику кількість готових рішень для роботи із форматами JSON та CSV, а також фреймворк Spring та його модуль Spring WebFlux.

Зазвичай Spring та Spring WebFlux використовуються для розробки вебзастосунків, а точніше – реактивних вебзастосунків, оскільки модуль WebFlux реалізує підтримку парадигми реактивного програмування для Spring і базується на бібліотеці Project Reactor. [5] Реактивне програмування є функціональним та декларативним за своєю суттю. Замість опису конкретних кроків, які необхідно виконати для обробки даних послідовно, ця парадигма радше описує певний конвеєр або потік, через який дані проходять. Реактивний підхід не вимагає наявності усіх даних для їх обробки, натомість, він надає можливості для обробки даних в той момент, коли вони стають доступними. [6] Також Spring WebFlux надає зручний високорівневий HTTP-клієнт, який не вимагає ручного контролю за відкриттям та закриттям мережевих з’єднань, ручної обробки помилок, тощо. Водночас, цей клієнт має широкі можливості для налаштування, які дозволяють підібрати найкращу конфігурацію для комунікації із потрібним сервером.

3.2 Steam API

Steam надає досить широкий API, який має можливості, пов’язані із майже всіма функціями сервісу – від отримання інформації про сутність (гру, програму, відео, тощо) до перегляду поточної ситуації на внутрішньому ринку предметів та встановлення двофакторної аутентифікації для акаунту. Не всі можливості є публічними, оскільки деякі з них забезпечують виконання дій, специфічних для акаунту конкретного користувача, а тому вимагають попередньої авторизації. Майже всі доступні можливості Steam API можна знайти на вебресурсах [7] та [8]. Проте, через велику кількість можливостей, не усі вони описані у документації, а тому деякі з них були знайдені та досліджені сторонніми розробниками. Деякі їх знахідки описані на вебресурсі [9].

Для отримання даних про сутності буде використано два ендпоїнти. Вони є публічними, тому для їх використання не потрібно проходити авторизацію чи використовувати ключ для доступу до API.

Перший - GET http://api.steampowered.com/ISteamApps/GetAppList/v2. Він дозволяє отримати список усього контенту, який доступний у Steam – ігор, їх доповнень, програм, тощо. Він не має ніяких додаткових параметрів і повертає список сутностей з їх ідентифікаторами та назвою. Отримані ідентифікатори будуть використані для отримання інформації про конкретну сутність.

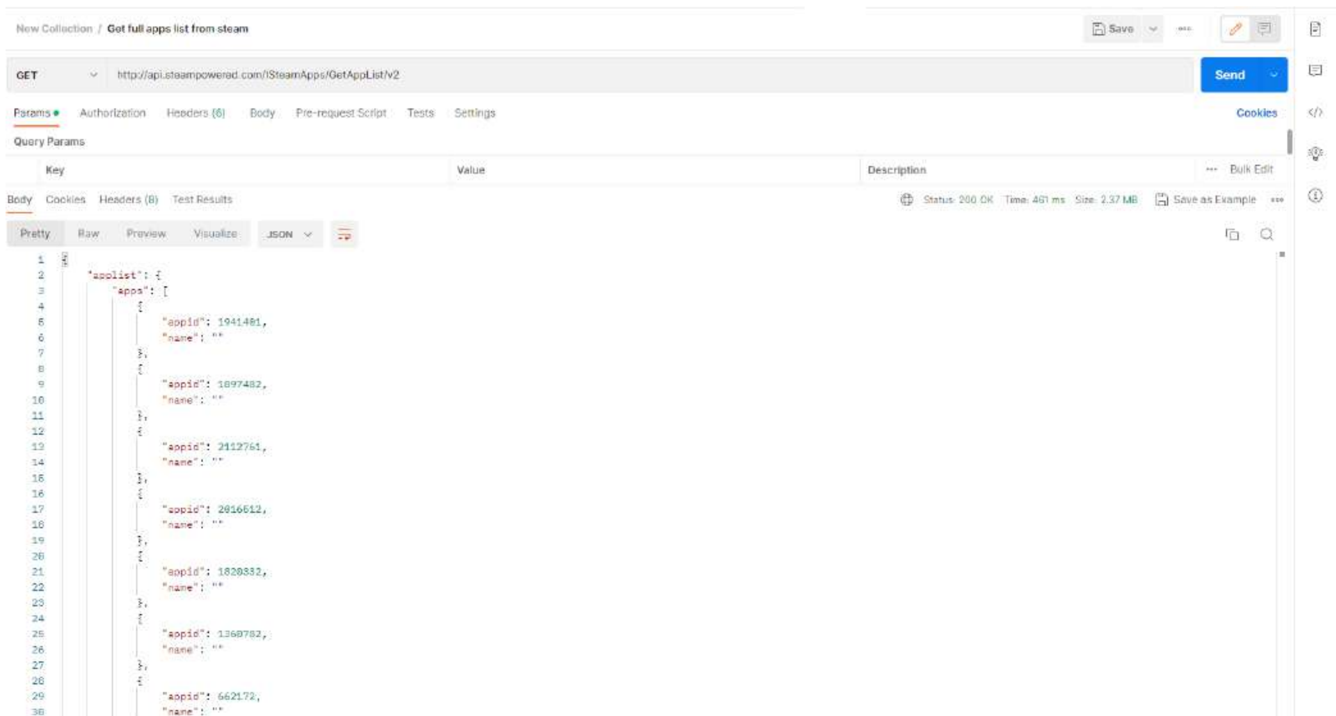


Рисунок 3.2.1. Частина відповіді на запит для отримання списку усього доступного контенту від Steam API

Другий – GET https://store.steampowered.com/api/appdetails. Він повертає детальну інформацію про сутність і включає її тип (гра, програма, відео, тощо) назву, опис, ціну, дату виходу, список розробників та видавців, список жанрів та багато іншої інформації. Також для різних типів сутностей дані, що повертаються, можуть мати різну структуру, наприклад, деякі поля будуть присутні або відсутні. Ендпоінт має один обов’язковий HTTP-параметр – “appids”. Йому потрібно надати ідентифікатор сутності, інформацію про яку потрібно отримати. Також є можливість вказати два додаткових параметри – “cc”, який дозволяє вказати код країни, у валюті якої потрібно рахувати ціни, та “l”, який дозволяє вказати назву мови, якою потрібно перекласти текстову інформацію у відповіді.

Рисунок 3.2.2. Частина відповіді на запит для отримання детальної інформації про сутність від Steam API

3.3 Steam Spy API

Steam Spy має значно простіший API, який лише надає інформацію про сутності за різними критеріями. Усі його можливості описані на вебресурсі [10].

Для виконання усіх запитів використовується основний URL - https://steamspy.com/api.php. Для усіх запитів обов’язковим HTTP-параметром є “request” – він визначає тип запиту та які ще параметри потрібно використати для отримання даних. Для отримання детальної інформації про сутність необхідно встановити значення “request” = “appdetails”, а також для параметру “appid” вказати ідентифікатор сутності. Оскільки Steam Spy повністю базується на API, який надає Steam, то ідентифікатори сутностей повністю співпадають із тими, які можна отримати від Steam API.

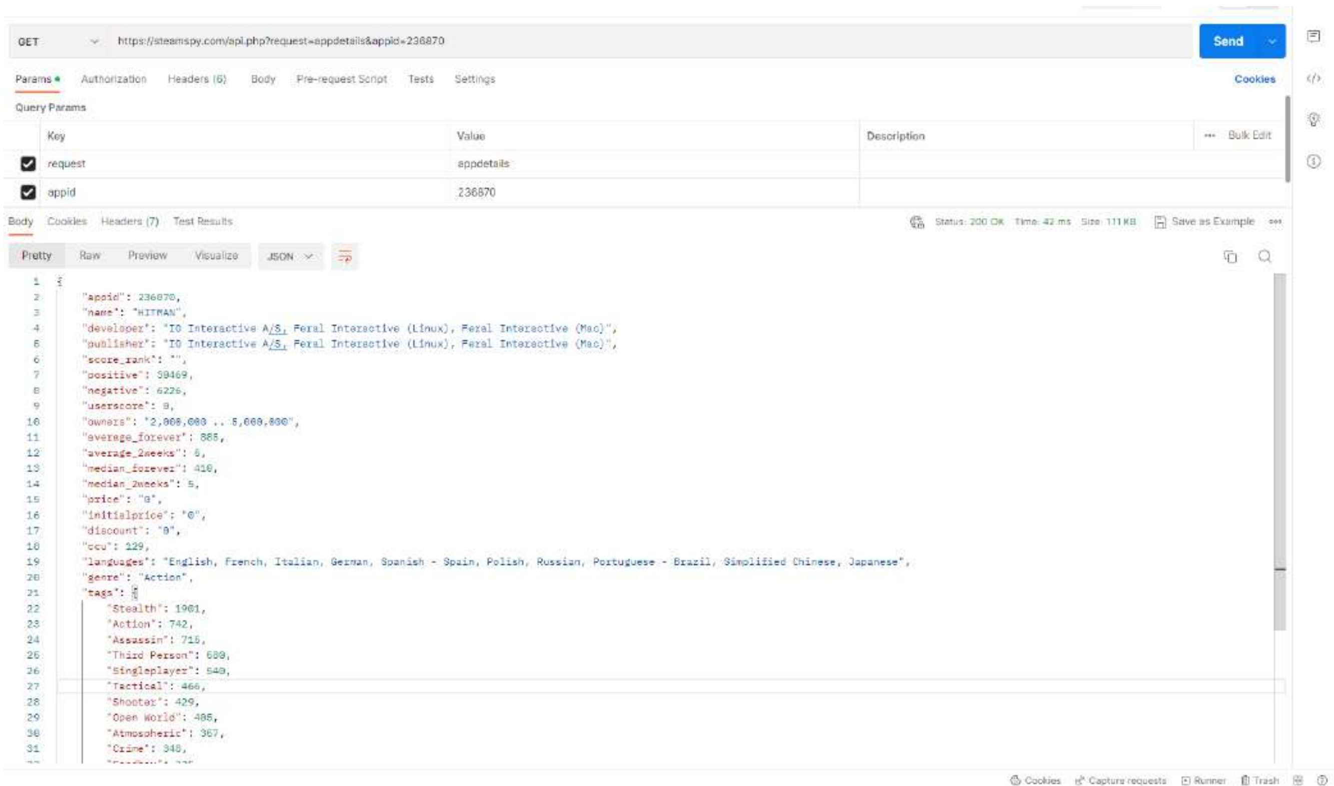


Рисунок 3.3.1. Частина відповіді на запит для отримання детальної інформації про сутність від Steam Spy API

3.4 Розробка застосунку

Принцип роботи застосунку, є досить простим. Його задачами є:

- Отримати список ідентифікаторів усіх сутностей від Steam API.
- Для кожної сутності із цього списку отримати інформацію про неї зі Steam API та Steam Spy API і записати цю інформацію у файли.

Кількість сутностей у Steam оновлюється не дуже часто, а тому немає сенсу кожен раз при необхідності отримати список усіх ідентифікаторів сутностей звертатися до Steam API. Достатньо буде отримати дані один раз і зберегти їх локально. Зберігання цих даних в оперативній пам’яті не є найкращим рішенням, оскільки після завершення роботи застосунку вони зникнуть і доведеться знову звертатися до Steam API. Тому для зберігання усіх ідентифікаторів було створено файл steam-all-apps-ids.csv. При старті застосунку для отримання усіх ідентифікаторів достатньо просто зчитати цей файл. Він має досить просту

структуру – лише один стовпчик із назвою “steamappid”, а в кожному рядку зберігається ідентифікатор сутності. На момент розробки у файлі було 157523 унікальних ідентифікатори.

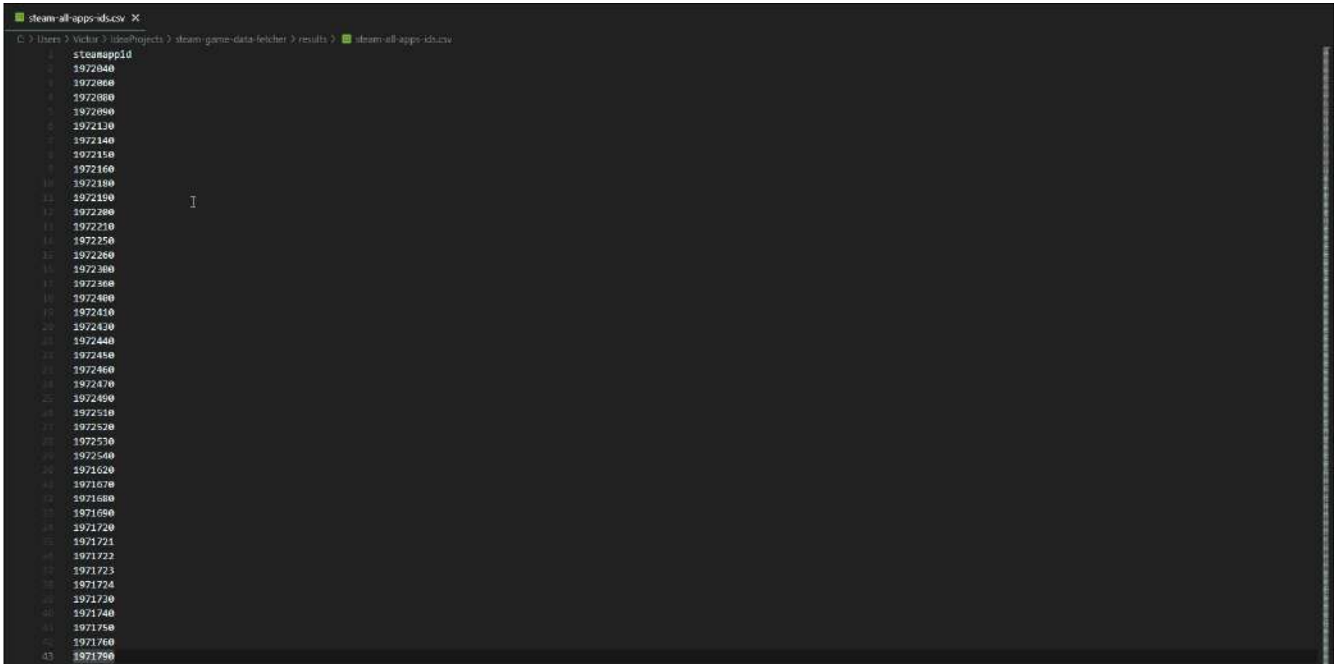


Рисунок 3.4.1. Частина файлу steam-all-apps-ids.csv

Після отримання усіх ідентифікаторів потрібно отримати дані про конкретну сутність на основі її ідентифікатора через Steam API та Steam Spy API. Цей процес дуже схожий для обох API, тому його можна описати на прикладі роботи зі Steam API.

Процес отримання даних про сутності зі Steam API складається із декількох кроків:

1. Отримати усі ідентифікатори сутностей, які ще не були оброблені.
2. Для кожного ідентифікатора зробити запит для отримання даних від Steam API.
3. Перетворити отриманий результат із формату JSON у формат CSV та записати у файл.

Розглянемо кожен крок детальніше.

Для отримання усіх ідентифікаторів сутностей, які ще не були оброблені, потрібно від усіх ідентифікаторів із файлу steam-all-apps-ids.csv відняти ті, які вже були оброблені. Вони зберігаються в окремому файлі, який має назву steam-processed-apps-ids.csv і має таку ж структуру, як і steam-all-apps-ids.csv. Цей файл формується і оновлюється після отримання даних про конкретну сутність від Steam API. Зберігання списку усіх оброблених ідентифікаторів дозволяє при завершенні та відновленні роботи застосунку не починати обробку усіх ідентифікаторів заново. Для роботи зі Steam Spy API існує аналогічний файл зі списком оброблених ідентифікаторів.

Наступним кроком є отримання даних від Steam API. Комунікація зі Steam API відбувається через окремий клас-сервіс SteamService і використовує HTTP-клієнт WebClient, який надає Spring WebFlux. При роботі зі стороннім публічним API ніколи не можна бути впевненим, що API буде стабільним, тому варто перебачити та врахувати сценарії, при яких процес отримання даних може піти не за планом. Основні можливі проблеми:

- тимчасова недоступність API. Наприклад, через велике навантаження API може дуже довго відповідати на клієнтські запити, або взагалі на них не відповідати. Іншим прикладом є проведення технічних робіт, під час яких API може бути повністю недоступним;
- наявність обмеження на кількість запитів від одного клієнта. Майже усі публічні API встановлюють певні обмеження на кількість запитів від одного клієнта щоб уникнути мережевих атак, пов’язаних із надсиланням великої кількості запитів для перевантаження сервера.

Ці проблеми були однією з причин використання окремих файлів для зберігання уже оброблених ідентифікаторів. Такі файли дозволяють відновити роботу застосунку при її завершенні через проблеми, спричинені сторонніми API. Також для зменшення впливу цих проблем були додані додаткові налаштування

для екземплярів класу WebClient, які можна побачити у ***Додатку А***. Було встановлено максимальний час підключення до сервера рівний 30 секундам, максимальний час отримання відповіді рівний 60 секундам та розмір буферу рівний 20 мегабайтам. Останнє налаштування потрібне для того, щоб отримати список усіх ідентифікаторів від Steam API, оскільки їх кількість досить велика (кінцева відповідь у форматі JSON займає декілька мегабайт).

У класі SteamService знаходиться основна логіка отримання даних від Steam API, його методи можна побачити у ***Додатку Б***. На прикладі цього класу можна побачити, що працювати із класом WebClient дуже зручно, оскільки для звернення до стороннього ресурсу достатньо обрати HTTP-метод, вказати URL ресурсу, якщо потрібно – додати HTTP-параметри. Також можна налаштувати логіку відправлення повторних запитів до сервера, якщо на попередній запит сервер повернув помилку. Відправлення повторних повідомлень використовується при отриманні даних про конкретну сутність за її ідентифікатором. Якщо сервер має повернути якісь дані, наприклад, документ у форматі JSON чи XML і його необхідно перетворити на екземпляр певного Java класу, то достатньо викликати спеціальний метод, у якому вказати назву класу. Оскільки Spring має інтеграцію із багатьма бібліотеками для серіалізації та десеріалізації даних у різні формати, він використовує їх для автоматичного перетворення отриманих даних із відповіді сервера в екземпляр бажаного класу. Цей спосіб використовується для перетворення відповіді від Steam API в екземпляр класу SteamAppDetailsResponse, який містить усю інформацію, яку можна отримати про сутність.

Для серіалізації та десеріалізації даних у формат JSON використовується бібліотека Jackson. Jackson – це набір інструментів для обробки даних для мови Java та платформи JVM, який надає можливості для парсингу та створення JSON-файлів а також перетворення Java об’єктів у формат JSON та створення їх із формату JSON. [11] Цю бібліотеку також використовує Spring як основну для

роботи із даними у форматі JSON. Класом, який відображає дані про сутності, отримані від Steam API є SteamAppDetailsResponse. Зазвичай, для перетворення даних із формату JSON в екземпляр Java класу достатньо визначити поля бажаного класу, надавши їм типи даних, які відповідають типам даних формату JSON, та налаштувати перетворення назв полів у JSON-документі в назви полів Java класу. Але при роботі зі Steam API з'ясувалось, що деякі поля JSON-документу, який повертає Steam API, мають нефіксований тип даних. Наприклад, деякі поля, якщо вони є JSON-об'єктами і мають вкладені поля, повертаються як JSON-об'єкти, а якщо не мають – як JSON-масиви. Таким чином, неможливо задати однозначний тип даних Java, який відповідає одночасно і об'єкту, і масиву, а тому потрібно створювати власний клас, який буде виконувати десеріалізацію JSON-документу в екземпляр класу SteamAppDetailsResponse. Для цього було створено клас SteamAppDetailsResponseDeserializer, який обробляє такі випадки і забезпечує правильну десеріалізацію. Реалізацію класу можна побачити у ***Додатку В***.

Для запису проміжних даних у файли формату CSV використовується бібліотека Apache Commons CSV. Вона була розроблена для створення єдиного та простого інтерфейсу для читання та запису даних у форматі CSV. [12] Для запису даних, отриманих зі Steam API використовується клас SteamDataCsvFileWriter, реалізацію якого наведено у ***Додатку Г***. Він наслідується від абстрактного класу AbstractCsvFileWriter, який містить логіку для налаштування формату CSV файлів та створення допоміжних екземплярів класів, що використовуються для запису у файли. Цю логіку було винесено в абстрактний клас, оскільки для збереження даних, отриманих від Steam Spy API, існує окремий клас SteamSpyDataCsvFileWriter, який використовує таку ж логіку для налаштування формату CSV. Використання абстрактного класу у цьому випадку дозволяє запобігти дублюванню коду. Основною задачею класу SteamDataCsvFileWriter є

обробка усіх можливих результатів отримання даних від Steam API. Якщо дані про сутність були успішно отримані від API, вони записуються у файл steam-all-apps-details.csv.

Рисунок 3.4.2. Частина файлу steam-all-apps-details.csv

Інколи Steam API може з певних причин не повертати дані про сутність. Для обробки цього випадку усі ідентифікатори сутностей, для яких даних немає, записуються в інший файл – steam-all-apps-details-no-data.csv. Як було зазначено раніше, відповідь від Steam API не завжди має однакову структуру, а тому можуть виникнути проблеми при десеріалізації відповіді в Java об’єкт. Для відстеження таких випадків SteamDataCsvFileWriter записує усі ідентифікатори сутностей, для яких десеріалізація була невдалою, в файл steam-all-apps-details-not-parsed.csv. Після обробки кожного випадку, як успішного, так і неуспішного, та запису результатів у файл потрібно зберегти інформацію, що сутність уже була оброблена. Для цього її ідентифікатор записується у файл steam-processed-apps-ids.csv. Він потрібний для того, щоб при пошуку уже оброблених ідентифікаторів

не збирати інформацію із трьох різних файлів, а використовувати лише один, який містить усю необхідну інформацію.

Повний цикл отримання даних від Steam API та їх збереження забезпечує клас SteamAppDetailsProcessor, реалізацію якого можна побачити у ***Додатку Д***. Він для кожного ідентифікатора отримує дані про сутність за допомогою SteamService, а потім записує отримані результати у файли за допомогою SteamDataCsvFileWriter. Важливою функцією класу SteamAppDetailsProcessor є забезпечення певної паузи між обробкою кожного ідентифікатора. Ця потреба виникає через наявність обмеження у Steam API – він дозволяє зробити максимум 200 запитів кожні 5 хвилин для одного клієнта. Таким чином, якщо робити усі запити послідовно, то для дотримання цього ліміту потрібно робити по одному запиту із мінімальним інтервалом між ними в 1.5 секунди. Одним зі способів обійти це обмеження є використання проксі-серверів. Це рішення дозволить видати запити, які робляться одним клієнтом (застосунком), за запити, відправлені декількома різними клієнтами, що дозволяє обробляти декілька ідентифікаторів одночасно. Проте, використання публічних проксі-серверів не є надійним, оскільки розробник не має ніякого контролю над ними, тому при роботі з ними доведеться витратити зайвий час на обробку можливих помилок, а для розгортання власних проксі-серверів потрібно витратити додатковий час та ресурси, що не буде дуже ефективним у даному випадку, оскільки кількість запитів, які необхідно зробити, не є дуже великою. Тому було обрано простіше рішення із встановленням затримки між обробкою ідентифікаторів у 1.6 секунди. Ця затримка трохи більша, ніж мінімально можлива, але вона дозволяє врахувати неточності при обрахунку часу між запитами, оскільки немає жодної інформації про те, з якою точністю рахує час Steam API.

Для усіх описаних вище класів існують аналогічні, вони забезпечують роботу зі Steam Spy API. Це SteamSpyService, SteamSpyAppDetailsResponse,

SteamSpyAppDetailsResponseDeserializer, SteamSpyDataCsvFileWriter та SteamAppDetailsProcessor. Завдяки схожому процесу отримання даних від обох API було використано можливість створення абстрактних класів для реалізації спільної логіки в одному місці та уникнення дублювання коду. Файлову структуру проекту застосунку можна побачити у ***Додатку Е***.

Розділ 4. Розробка застосунку для обробки даних

4.1 Вибір технологій розробки

Основними завданнями застосунку є отримання даних із файлів, що були підготовлені застосунком, який відповідає за збір даних, їх обробка, очищення, трансформація та збереження у базі даних. При виборі технологій також варто врахувати декілька аспектів:

- дані, із якими потрібно працювати, не проходили процес попередньої обробки, тому обрані технології мають забезпечувати роботу із неструктурованими та “сирими” даними;
- обрані технології мають підтримувати обробку великих об’ємів даних, оскільки файл із даними, отриманими від Steam API має досить значний об’єм – 1,02 гігабайта.

Враховуючи вищезазначені вимоги та деталі, найкращим вибором для виконання поставленого завдання є фреймворк Apache Spark.

Apache Spark – це двигун для роботи із великими даними. [13] Він має велику кількість можливостей, серед яких:

- робота як зі структрурованими, так і з неструктурованими даними;
- робота зі статичними даними та даними, що отримуються в реальному часі;
- наявність інструментів для виконання аналізу даних;
- наявність інструментів для машинного навчання;
- наявність клієнтів для декількох мов програмування: Scala, Java, Python, R;
- підтримка великої кількості джерел даних (різні файлові формати, реляційні та нереляційні бази даних, брокери повідомлень).

Spark базується на master-worker архітектурі. Основою цієї архітектури є поділ її компонентів на два види – головний (master) та підлеглий (worker). Зазвичай, використовується один головний та декілька підлеглих вузлів. Їх основна відмінність – це функції, які виконує кожен із них.

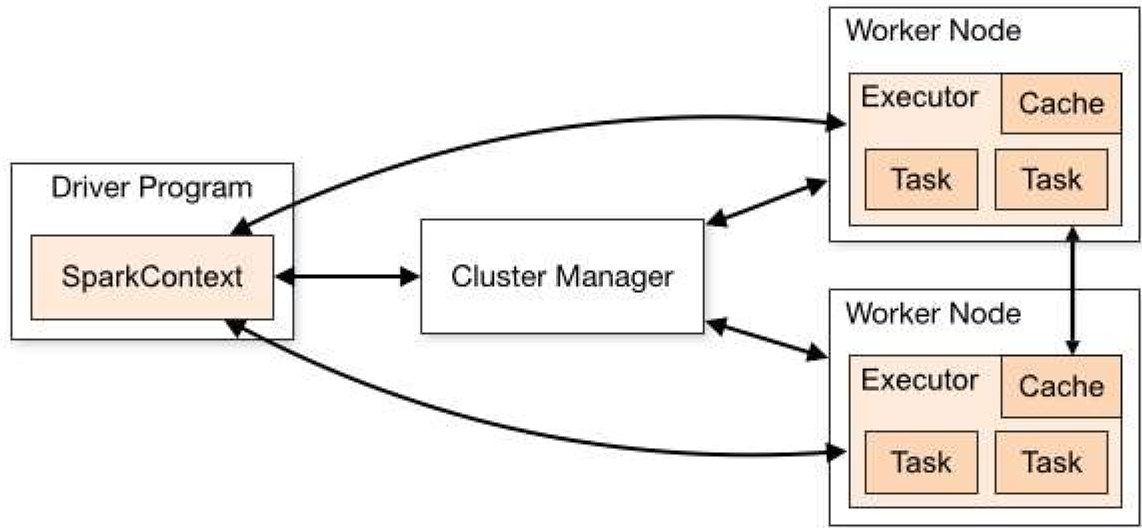


Рисунок 4.1.1. Master-worker архітектура Apache Spark

Головний компонент у Spark – це Driver. Він відповідає за поділ усіх задач, які має виконати застосунок, на етапи, побудову послідовності їх виконання та розподіл задач між підлеглими вузлами. Для представлення задач Spark використовує направлений ациклічний граф, який будується на основі усіх дій, які виконує застосунок. Цей граф також додатково оптимізується для забезпечення якнайбільшої продуктивності, а потім на його основі виокремлюються етапи, необхідні для виконання кожної задачі. Після цього виконується планування та розподіл усіх задач між доступними підлеглими компонентами, які мають назву Executor.

В свою чергу, кожен підлеглий компонент відповідає за безпосереднє виконання задач, отриманих від головного компонента. Саме підлеглі вузли виконують читання даних, операції для їх трансформації та їх запис. Підлеглі вузли також можуть обмінюватись даними один з одним, якщо виникає така

потреба. Якщо один із підлеглих вузлів не закінчив виконання своїх задач із певних причин (наприклад, фізична машина, на якій відбувалось виконання, стала недоступна), головний вузол виконує перерозподіл його задач між іншими доступними підлеглими вузлами.

Spark надає декілька можливостей для розгортання застосунків. Серед них:

- розгортання усього застосунку в одному процесі. Це найпростіший варіант, оскільки не потребує наявності декількох машин для роботи застосунку, а головний та підлеглі вузли знаходяться в одному процесі операційної системи;
- розгортання застосунку у вигляді декількох процесів на одній фізичній машині. Це один процес для головного вузла та один або більше процесів для підлеглих вузлів;
- розгортання застосунку у вигляді різних процесів на різних фізичних машинах. Для цього варіанту необхідно використати один із менеджерів кластерів, що підтримуються у Spark: власний менеджер кластеру, який надає Spark, Apache Mesos, Hadoop YARN чи Kubernetes.

Для виконання задач роботи було обрано варіант із розгортанням застосунку в одному процесі, оскільки об’єм даних не є настільки великим, що потрібно розгортати окремий кластер для роботи із ними.

У якості мови програмування було обрано Scala, оскільки Spark розроблений саме нею. Це дозволяє використовувати увесь доступний API, який надає Spark, а при необхідності навіть розширювати його можливості для своі потреб.

4.2 Структура бази даних

Для побудови структури бази та визначення необхідних таблиць необхідно розглянути доступні дані, обрати ключову інформацію, знайти зв’язки між

даними, а також визначити дані, які не є корисними і можуть бути проігноровані. Аналізуючи дані, отримані з обох джерел, можна помітити наступні деталі:

- кожна сутність має певний наперед визначений тип, тому можна прослідкувати зв’язок виду “один до багатьох” між типом та сутністю. Таким чином, тип може стати окремою моделлю у предметній області;
- кожна сутність має від 0 до декількох розробників, видавців, категорій та жанрів, тобто між сутністю та кожною із наведених моделей існує зв’язок виду “багато до багатьох”;
- для кожної сутності як Steam, так і Steam Spy надають деяку спільну інформацію (наприклад, назву, розробників та видавців), проте також присутня інформація, специфічна для кожного із сервісів. Тому доречним буде сформувати одну кореневу сутність, яка буде містити основну інформацію, а також допоміжні, у яких будуть зберігатись дані, специфічні для кожного із сервісів, причому допоміжні сутності будуть пов’язані із кореневою зв’язком виду “один до одного”. Ще одним аргументом на користь такого рішення є те, що про у кожному сервісі може не бути інформації про деякі сутності;
- деяка інформація, яку надають Steam та Steam Spy не є корисною в рамках системи, що розроблюється. Наприклад, Steam для кожної сутності повертає список скріншотів та трейлерів, які користувач бачить на сторінці цієї сутності на вебсайті або у настільному додатку. Така інформація може бути вилучена, оскільки ніяк не використовується.

На основі проведеного аналізу було розроблено схему бази даних, яку можна побачити на рисунку 4.2.1.

Рисунок 4.2.1. Схема бази даних

Кореневу сутність представляє таблиця “application”. Вона пов’язана зв’язком виду “багато до одного” з таблицею “application_type”, яка представляє тип сутності, а також зв’язками виду “багато до багатьох” із таблицями “developer”, “publisher”, “category” та “genre”, які представляють розробника, видавця, категорію та жанр відповідно. Для реалізації зв’язку виду “багато до багатьох” було використано проміжні таблиці. Також вона пов’язана зв’язками виду “один до одного” із таблицями “steam_application_details” та

“steam_spy_application_details”, які представляють дані про сутність, отримані від сервісів Steam та Steam Spy відповідно.

4.3 Розробка застосунку

Основними задачами застосунку є зчитування даних із попередньо підготовлених файлів формату CSV, їх обробка, трансформація та запис у базу даних. Усі ці завдання дозволяє виконати Spark. Для початку роботи зі Spark необхідно ініціалізувати екземпляр класу SparkSession, через який відбувається робота із Spark API.

Першим етапом роботи є зчитування даних із файлів формату CSV. Для цього було створено об’єкт CsvFileReader, який використовує SparkSession для зчитування даних. Цей об’єкт встановлює налаштування формату CSV, які потребує Spark для правильного читання, а також ім’я файлу, із якого необхідно зчитати дані. Результатом зчитування даних є датафрейм.

Рисунок 4.3.1. Реалізація об'єкту CsvFileReader

Датафрейм – це основна одиниця представлення даних, яку використовує Spark. Він складається зі стовпчиків, кожен із яких має назву і представляє певну властивість, та рядків, кожен з яких представляє одну сутність. Датафрейми дуже схожі на таблиці, які представляють дані у реляційних базах даних. Датафрейми є

незмінними і будь-яка операція, яка виконується над датафреймом, перетворює його на інший датафрейм, але не змінює початковий датафрейм. Кожен датафрейм має схему, яка визначає, які стовпчики наявні у датафреймі і тип даних для кожного з них.

За допомогою CsvFileReader необхідно зчитати 3 файли, дані з яких будуть використані для подальшої трансформації та збереження. Це steam-all-apps-details.csv, steam-all-apps-details-no-data.csv та steam-spy-all-apps-details.csv. Усі отримані після читання датафрейми будуть застосовані в подальшому для отримання необхідних даних. Проте, якщо один датафрейм необхідно використати декілька разів, то усі операції, потрібні для його отримання, будуть виконані заново. Таким чином, якщо один із отриманих вище датафреймів потрібно використати 5 разів, то він буде прочитаний із файлу 5 разів, що є дуже неефективним. Щоб уникнути перерахування датафрейму Spark дозволяє зберегти його у процесі виконання на певний час. Датафрейм може бути збережений в оперативній пам'яті машини, в постійній пам'яті, або використовувати обидва типи пам'яті. Після виконання усіх необхідних дій датафрейм може бути видалений із пам'яті. У застосунку усі датафрейми було збережено лише в оперативній пам'яті одразу після зчитування та видалено в кінці його роботи.

Для перетворення та збереження даних у кожну таблицю бази даних було розроблено єдину схему роботи із початковими датафреймами. Розглянемо її на прикладі збереження у базу даних інформації про відношення між сутностями та їх розробниками.

У датафреймах, отриманих із файлів, розробники сутності зберігаються у стовпчику CSV файлу, який представляє собою стрічку у форматі JSON, що представляє JSON-масив із назвами компаній розробників. У базі даних розробники зберігаються у вигляді таблиці, яка має 2 поля – ідентифікатор та назву компанії. Для зберігання відношень між сутністю та її розробниками у базі

даних існує таблиця, яка представляє це відношення у вигляді таблиці із 2 полями: ідентифікатором сутності та ідентифікатором розробника. Тому для зберігання відношень потрібно для кожної сутності із датафрейму додати ідентифікатори усіх її розробників. Для цього треба зчитати таблицю з розробниками в датафрейм. Для цього було створено об'єкт DatabaseReader, який дуже схожий на CsvFileReader. Він використовує SparkSession, встановлює необхідні налаштування для під'єднання до бази даних, назву таблиці, яку необхідно прочитати, та повертає датафрейм із даними з таблиці. Реалізацію об'єкту DatabaseReader наведено на рисунку 4.3.2.

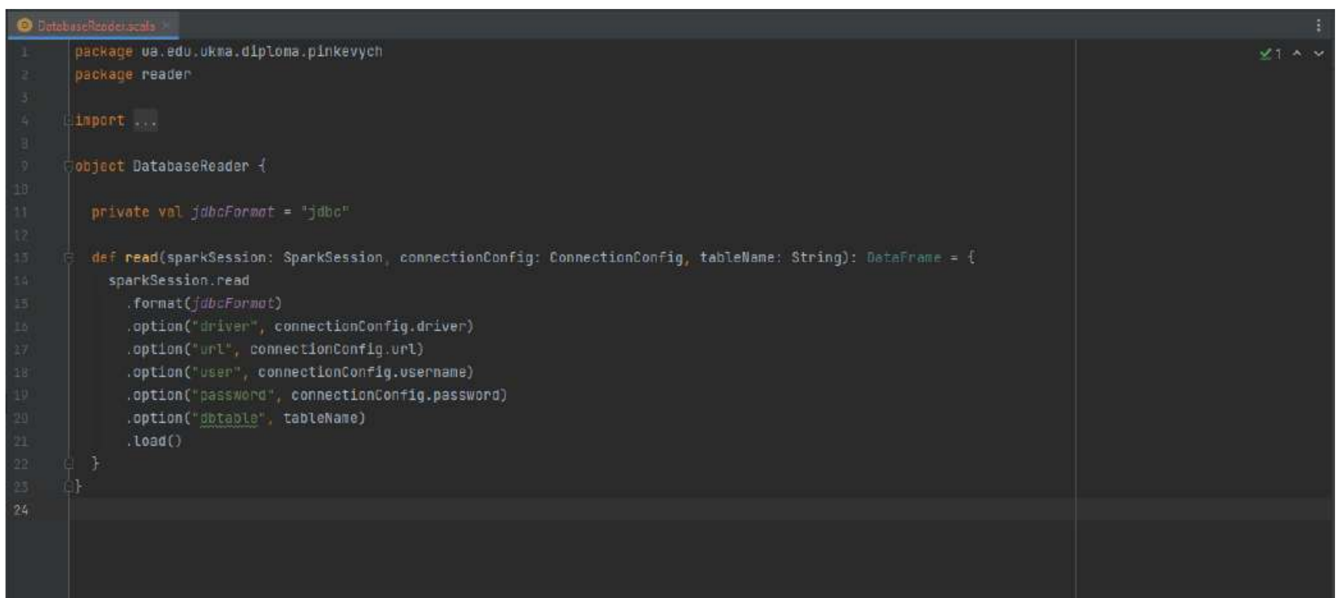


Рисунок 4.3.2. Реалізація об'єкту DatabaseReader

Після отримання усіх необхідних даних можна почати їх перетворення. За виконання перетворень над датафреймами сутностей та розробників відповідає об'єкт SteamApplicationDeveloperTransformer, реалізацію якого можна побачити на рисунку 4.3.3. Першим кроком є підготовка даних про сутність, а саме – отримання усіх імен її розробників у форматі масиву стрічок із JSON-масиву. Для цього використовується окрема визначена користувачем функція та вищезгадана бібліотека Jackson, оскільки вона також доступна і для мови Scala.

```scala
package ua.edu.ukma.diploma.pinkevych
package transformer.developer

import ...

object SteamApplicationDeveloperTransformer {

  def transform(
      steamAppsDetailsDF: DataFrame,
      developersDF: DataFrame,
      broadcast: Broadcast[JsonMapperProvider]
  ): DataFrame = {

    val developerArrayFromJson = udf((rawDeveloperNamesString: String) =>
      TransformationUtil.stringArrayFromJson(rawDeveloperNamesString, broadcast.value.getJsonMapper)
    )

    val steamApplicationDevelopersDF = steamAppsDetailsDF
      .select(
        col("steam_app_id"),
        explode(developerArrayFromJson(col("developers"))).as("developer_name")
      )

    steamApplicationDevelopersDF
      .as("steam_application_developers")
      .join(
        developersDF.as("developers"),
        steamApplicationDevelopersDF("developer_name") === developersDF("developer_name")
      )
      .select(
        col("steam_application_developers.steam_app_id").as("steam_application_id"),
        col("developers.id").as("developer_id")
      )
      .dropDuplicates()
  }
}
```

Рисунок 4.3.3. Реалізація об'єкту SteamApplicationDeveloperTransformer

Визначені користувачем функції (UDF) – це функції, яка дозволяють виконувати операції над стовпцями у датафреймах, але не є частиною фреймворку Spark, оскільки визначені сторонніми розробниками. Функції можуть приймати від 0 до 22 аргументів, але повертають лише одне значення, яке зберігається у стовпчику датафрейму. [14] Ця концепція є дуже потужною, оскільки дозволяє легко розширювати можливості фреймворку та додавати необхідний функціонал. Функції можуть бути написані мовою, якою виконується розробка, таким чином дозволяючи використовувати будь-які засоби мови, включно зі сторонніми бібліотеками та фреймворками. Ці функції можуть бути викликані як напряму, так і через Spark SQL. Для їх використання достатньо попередньо зареєструвати

бажані функції через SparkSession або використовуючи спеціальні функції, які Spark надає у клієнтах для кожної підтримуваної мови.

Також для перетворення JSON-масиву у масив стрічок необхідно використати об’єкт класу JsonMapper. Якщо створювати його напряму при виклику функції для перетворення, то новий об’єкт буде створюватись для кожного виклику, що є дуже неефективним. Найкращим рішенням буде передавати його параметром у функцію. Проте це не вийде зробити просто створивши об’єкт, оскільки усі дії над датафреймами виконуються окремими підлеглими вузлами, кожен із яких має свою пам’ять, кеш та є повністю незалежним від інших підлеглих вузлів. Проте Spark має механізм, який дозволяє передавати спільні дані від головного вузла усім підлеглим – трансльовані змінні.

Трансльовані змінні (Broadcast variables) - це змінні, які доступні усім вузлам кластера лише для читання і призначені для ефективного зберігання та кешування спільних даних кожним вузлом. Без використання трансльованих змінних Spark буде передавати необхідні для виконання задачі дані кожен раз перед виконанням нової задачі. Використання трансльованих змінних дозволяє передати дані лише один раз і зберігати їх у кеші вузла протягом усієї роботи. [15]

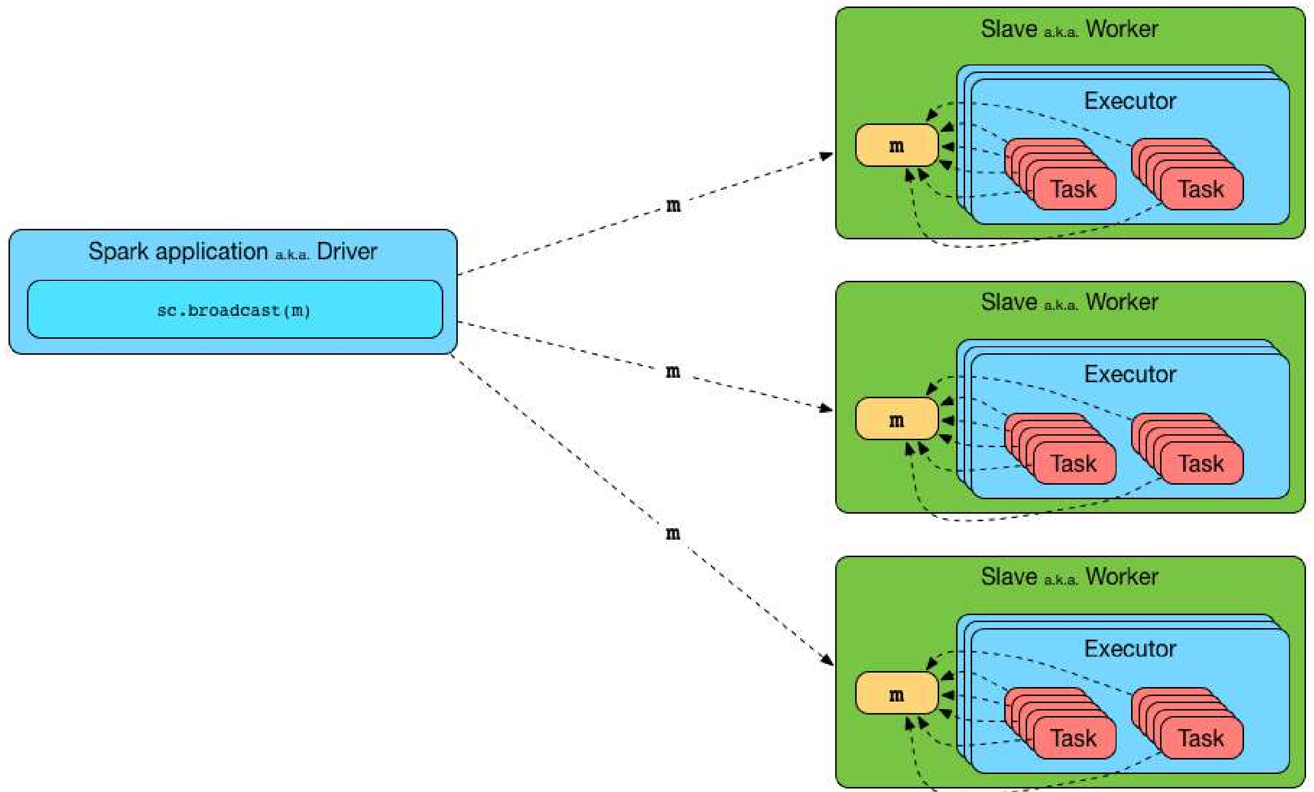


Рисунок 4.3.4. Принцип роботи трансльованих змінних в Spark

За допомогою трансльованих змінних у застосунку кожен підлеглий вузол отримує екземпляр класу JsonMapperProvider, який створює екземпляр класу JsonMapper із необхідними налаштуваннями. Таким чином, кожен підлеглий вузол має власний екземпляр класу JsonMapper.

Після підготовки даних про сутності потрібно пов’язати їх із даними про розробників. Для поєднання даних між датафреймами Spark надає можливість виконувати з’єднання датафреймів, які за своєю суттю є аналогічними до з’єднань таблиць у реляційних базах даних. Spark підтримує усі ті ж види з’єднань, що і реляційні бази даних: внутрішні, ліві, праві та повні. Для з’єднання датафреймів потрібно обрати два датафрейми, які потрібно з’єднати, та задати умову, при виконанні якої рядки датафреймів будуть об’єднані. Результатом об’єднання є новий датафрейм, який містить усі стовпчики із обох датафреймів, а також нові рядки, значення в яких залежать від типу з’єднання.

Для того щоб пов’язати ідентифікатори сутностей та ідентифікатори їх розробників достатньо виконати внутрішнє з’єднання на основі рівності імен розробників. Результатом внутрішнього з’єднання є датафрейм, який містить лише ті рядки, для яких виконується умова з’єднання, тобто ті, для яких було знайдено ідентифікатори розробників за їх іменами.

Після формування кінцевого датафрейму його потрібно зберегти у таблицю бази даних. Spark надає зручний API для запису датафреймів у будь-яке сховище: файл, базу даних, тощо. Для збереження датафрейму достатньо вказати формат та відповідні налаштування. Оскільки в застосунку усі датафрейми зберігаються у базу даних, було створено об’єкт DatabaseWriter, який для заданого датафрейму встановлює відповідні налаштування для запису у базу даних та зберігає його. Реалізацію цього об’єкту можна побачити на рисунку 4.3.5.

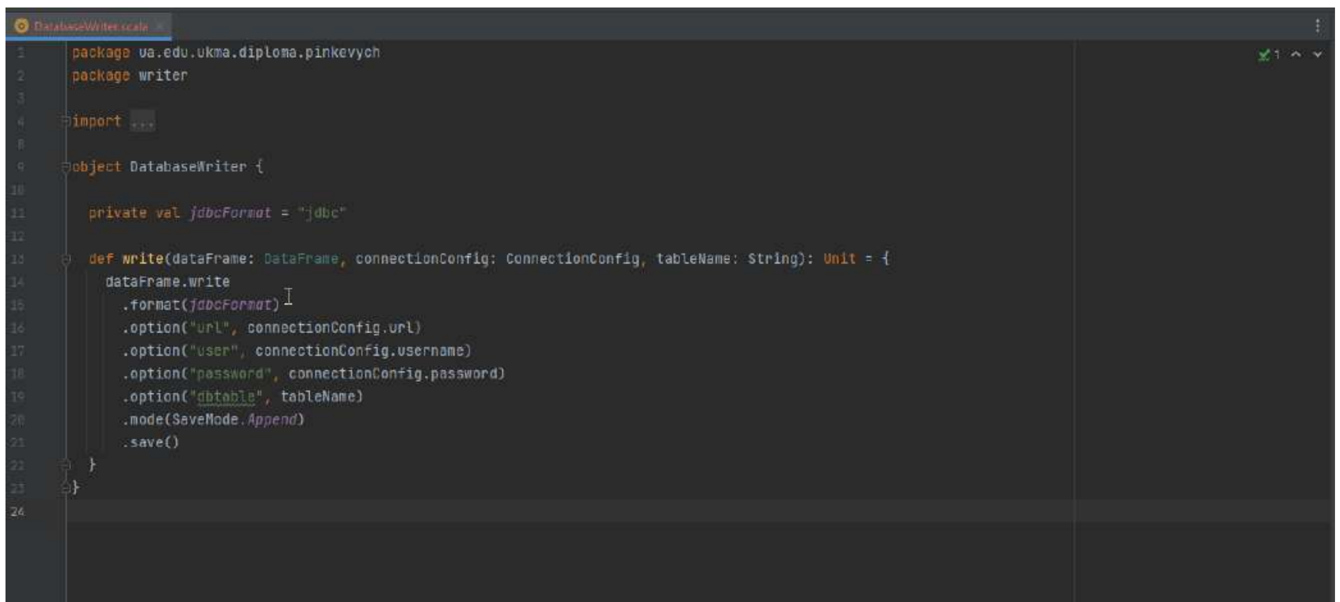


Рисунок 4.3.5. Реалізація об'єкту DatabaseWriter

Spark дозволяє як зберігати датафрейми в існуючі таблиці бази даних, так і створювати нові на основі схеми датафрейму. Проте підхід зі створенням таблиць на основі схеми датафрейму не підійде, якщо потрібно створити таблицю із певними обмеженнями, чітко вказати тип даних стовпчика чи задати первинні та зовнішні ключі, створити індекси. В таких випадках найкращим рішенням є

створення таблиць заздалегідь і налаштування Spark для запису лише в уже існуючі таблиці. Щоб уникнути необхідності створювати таблиці у базі даних вручну перед кожним запуском застосунку було створено об’єкт SchemaInitializer, який на початку роботи застосунку створює необхідні таблиці у базі даних, якщо вони відсутні. Таблиці створюються з використанням SQL-скриптів, що зберігаються у зовнішніх файлах, а взаємодія із базою даних відбувається через JDBC – API для мови Java та платформи JVM, який забезпечує доступ до баз даних через використання відповідних драйверів. Оскільки Spark працює на платформі JVM, то він також використовує JDBC для роботи із базами даних. Реалізацію об’єкту SchemaInitializer можна побачити на рисунку 4.3.6.

```scala
object SchemaInitializer {

  def initializeSchema(schemaConfig: SchemaConfig, connectionConfig: ConnectionConfig): Unit = {
    val files: Seq[String] = Seq(
      schemaConfig.applicationTypeFile,
      schemaConfig.applicationFile,
      schemaConfig.steamApplicationDetailsFile,
      schemaConfig.steamSpyApplicationDetailsFile,
      schemaConfig.categoryFile,
      schemaConfig.applicationCategoryFile,
      schemaConfig.genreFile,
      schemaConfig.applicationGenreFile,
      schemaConfig.developerFile,
      schemaConfig.applicationDeveloperFile,
      schemaConfig.publisherFile,
      schemaConfig.applicationPublisherFile
    )
    val queries: Seq[String] = files.map(readQueryFromFile)
    queries.foreach(println)

    val connection: Connection =
      DriverManager.getConnection(connectionConfig.url, connectionConfig.username, connectionConfig.password)
    val statement: Statement = connection.createStatement()
    queries.foreach(statement.execute)
    statement.close()
    connection.close()
  }

  private def readQueryFromFile(fileName: String): String = {
    val filePath = getClass.getClassLoader.getResource(fileName).getPath
    val source   = Source.fromFile(filePath)
    val query    = source.getLines().mkString
    source.close()
    query
  }
}
```

Рисунок 4.3.6. Реалзіація об'єкту SchemaInitializer

Для об'єднання процесів отримання, обробки та збереження даних про сутності та їх розробників було створено об'єкт ApplicationDeveloperKeeper, реалізацію якого наведено на рисунку 4.3.7. Він отримує початкові датафрейми і на їх основі, використовуючи усі описані вище об'єкти, виконує трансформацію та збереження даних. У реалізації об'єкту можна побачити використання датафрейму steamSpyAppsDetailsWithNoDataInSteamDF. Він містить дані про усі сутності, про які немає даних у сервісі Steam, але є у сервісі Steam Spy. Його використання дозволяє отримати ключові дані про сутність незалежно від сервісу, з якого вони були отримані. Цей датафрейм також використовується для отримання даних про видавців та жанри сутностей.

```
package ua.edu.ukma.diploma.pinkevych
package keeper

import ...

object ApplicationDeveloperKeeper {

  def keep(
      sparkSession: SparkSession,
      steamAppsDetailsDF: DataFrame,
      steamSpyAppsDetailsWithNoDataInSteamDF: DataFrame,
      applicationConfig: ApplicationConfig,
      broadcast: Broadcast[JsonMapperProvider]
  ): Unit = {
    val developersDF =
      DatabaseReader.read(
        sparkSession,
        applicationConfig.databaseConfig.connectionConfig,
        applicationConfig.databaseConfig.tables.developer
      )
    developersDF.persist(StorageLevel.MEMORY_ONLY)

    val transformedSteamApplicationDevelopersDF =
      SteamApplicationDeveloperTransformer.transform(steamAppsDetailsDF, developersDF, broadcast)
    val transformedSteamSpyApplicationDevelopersDF =
      SteamSpyApplicationDeveloperTransformer.transform(steamSpyAppsDetailsWithNoDataInSteamDF, developersDF)
    val transformedDevelopersDF =
      transformedSteamApplicationDevelopersDF.unionByName(transformedSteamSpyApplicationDevelopersDF)

    DatabaseWriter.write(
      transformedDevelopersDF,
      applicationConfig.databaseConfig.connectionConfig,
      applicationConfig.databaseConfig.tables.applicationDeveloper
    )

    developersDF.unpersist()
  }

}
```

Рисунок 4.3.7. Реалізація об'єкту ApplicationDeveloperKeeper

Описаний вище процес отримання даних для їх збереження в одній з таблиць бази даних є спільним для усіх таблиць. Для кожної з них існують об’єкти із суфіксом Keeper та Transformer, а також використовуються об’єкти DatabaseReader та DatabaseWriter для читання та запису даних у базу даних відповідно. Такий підхід забезпечує легке розуміння зв’язків між об’єктами в застосунку і зручність роботи із ним, оскільки завжди зрозуміло, де потрібно зробити зміни, якщо виникає така потреба. Файлову структуру проекту застосунку можна побачити у ***Додатку Є***.

Розділ 5. Аналіз виконаної роботи

5.1 Можливості розвитку розробленої системи

Розроблена система уже складається із двох незалежних частин, а тому може стати прототипом для більш функціональної системи, що буде отримувати, обробляти та надавати дані, отримані від сервісів Steam та Steam Spy, у реальному часі. Це вимагатиме досить частих звертань до сторонніх API для перевірки оновлення даних, більшої потужності для обробки більшої кількості даних, а також кращого рішення для комунікації між частинами системи, ніж проміжні файли. Для виконання цих вимог можна застосувати:

- Проксі-сервери, які дозволять звертатися до сторонніх сервісів паралельно та уникнути обмежень на кількість запитів. Це значно пришвидшить процес отримання даних від сторонніх API.
- Брокери повідомлень, що забезпечать комунікацію між частинами системи в режимі реального часу. Із одним файлом одночасно може взаємодіяти лише один процес операційної системи, тому на їх основі не вийде побудувати систему для комунікації між процесами в реальному часі.
- Створення повноцінного Spark-кластеру, що дозволить задіяти більше ресурсів для обробки більшої кількості даних.

Впровадження наведених вище рішень дозволить розробити систему, яка дозволить отримувати найбільш актуальні дані від сторонніх API у реальному часі.

5.2 Висновки

В ході виконання роботи було розроблено систему, яка виконує повний процес отримання, обробки та зберігання даних, отриманих зі сторонніх джерел. Було розроблено реляційну модель предметної області на основі доступних даних.

Розроблена реляційна модель надає основну інформацію для опису сутностей предметної області і може бути розширена при наявності нових джерел інформації.

При розробці застосунку, який відповідає за отримання даних, були розглянуті основні проблеми, які можуть виникнути при взаємодії зі сторонніми публічними API та основні способи їх вирішення. Також було описано особливості роботи із неструктурованими даними, отриманими зі сторонніх джерел. Важливою частиною розробки було використання сторонніх інструментів для комунікації зі сторонніми API та перетворення даних між різними форматами представлення. Використання таких інструментів дозволяє уникнути необхідності власноруч створювати рішення для стандартних проблем і зосередитись на виконанні задач, що стоять перед застосунком.

При розробці застосунку для обробки та збереження даних було розглянуто основні можливості фреймворку Apache Spark. Вони були застосовані для вирішення задач обробки великої кількості неструктурованих даних, отриманих із різних джерел. Використання Apache Spark забезпечило зручність розробки, велику кількість інструментів для виконання обробки даних, можливості для гнучкого розширення стандартного функціоналу Spark а також високу ефективність роботи кінцевого застосунку.

Було розглянуто можливості подальшого розвитку та розширення розробленої системи, які дозволять забезпечити її роботу із даними, отриманими у реальному часі.

Дані, отримані в результаті роботи системи, можуть бути використані для проведення аналізу стану ринку відеоігор на момент їх збору.

Список використаних джерел

1. Офіційний сайт сервісу Steam [Електронний ресурс] – Режим доступу до ресурсу: https://store.steampowered.com/about.
2. Офіційний сайт сервісу Steam Spy [Електронний ресурс] – Режим доступу до ресурсу: https://steamspy.com/about.
3. Документація СКБД MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://dev.mysql.com/doc/refman/8.0/en/.
4. Документація Docker [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/.
5. Документація Spring WebFlux [Електронний ресурс] – Режим доступу до ресурсу: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html.
6. Walls C. Spring in Action / Crain Walls., 2018. – 520 с.
7. Документація Steamworks API [Електронний ресурс] – Режим доступу до ресурсу: https://partner.steamgames.com/doc/webapi.
8. Документація Steam API [Електронний ресурс] – Режим доступу до ресурсу: https://steamapi.xpaw.me/.
9. Official Team Fortress Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.teamfortress.com/wiki/User:RJackson/StorefrontAPI.
10. Документація Steam Spy API [Електронний ресурс] – Режим доступу до ресурсу: http://steamspy.com/api.php.
11. Домашня сторінка бібліотеки Jackson на GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/FasterXML/Jackson.
12. Документація Apache Commons CSV [Електронний ресурс] – Режим доступу до ресурсу: https://commons.apache.org/proper/commons-csv/index.html.
13. Документація Apache Spark [Електронний ресурс] – Режим доступу до ресурсу: https://spark.apache.org/docs/latest/.

14. Perrin J. Spark in Action, Second Edition / Jean-Georges Perrin., 2020. – 576 с.
15. Spark by Examples - Spark Broadcast Variables [Електронний ресурс] – Режим доступу до ресурсу: https://sparkbyexamples.com/spark/spark-broadcast-variables/.

Додатки

Додаток А. Клас WebClientConfiguration

```java
package ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.configuration;

import io.netty.channel.ChannelOption;
import io.netty.handler.timeout.ReadTimeoutHandler;
import java.util.concurrent.TimeUnit;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.reactive.ReactorClientHttpConnector;
import org.springframework.web.reactive.function.client.ExchangeStrategies;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.netty.http.client.HttpClient;

@Configuration
@RequiredArgsConstructor
public class WebClientConfiguration {

  private final WebClientProperties webClientProperties;

  @Bean
  public WebClient steamWebClient() {
    return WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(createHttpClient()))
        .exchangeStrategies(
            ExchangeStrategies.builder()
                .codecs(
                    codecs ->
                        codecs
                            .defaultCodecs()
                            .maxInMemorySize(webClientProperties.getClient().getMaxBufferSize()))
                .build())
        .build();
  }

  @Bean
  public WebClient steamSpyWebClient() {
    return WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(createHttpClient()))
        .build();
  }

  private HttpClient createHttpClient() {
    return HttpClient.create()
        .option(
            ChannelOption.CONNECT_TIMEOUT_MILLIS,
            Long.valueOf(webClientProperties.getClient().getConnectTimeout().toMillis()).intValue())
        .doOnConnected(
            connection ->
                connection.addHandlerFirst(
                    new ReadTimeoutHandler(
                        webClientProperties.getClient().getReadTimeout().toMillis(),
                        TimeUnit.MILLISECONDS)));
  }
}
```

Додаток Б. Клас SteamService

```java
package ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import reactor.util.retry.Retry;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.configuration.WebClientProperties;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.response.SteamAllAppsResponse;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.response.SteamAppDetailsResponse;

@Service
@RequiredArgsConstructor
public class SteamService {

  private final SteamProperties steamProperties;
  private final WebClientProperties webClientProperties;
  private final WebClient steamWebClient;

  private static final String STEAM_APP_DETAILS_ID_PARAM = "appids";
  private static final String STEAM_APP_DETAILS_LANGUAGE_PARAM = "l";
  private static final String STEAM_APP_DETAILS_LANGUAGE_ENGLISH = "english";
  private static final String STEAM_APP_DETAILS_CURRENCY_PARAM = "cc";
  private static final String STEAM_APP_DETAILS_CURRENCY_DOLLAR = "US";

  public Mono<SteamAllAppsResponse> getAllSteamApps() {
    return steamWebClient
        .get()
        .uri(steamProperties.getAllAppsUrl())
        .retrieve()
        .bodyToMono(SteamAllAppsResponse.class);
  }

  public Mono<SteamAppDetailsResponse> getSteamAppDetails(Long steamAppId) {
    return steamWebClient
        .get()
        .uri(
            steamProperties.getAppDetailsUrl(),
            uriBuilder ->
                uriBuilder
                    .queryParam(STEAM_APP_DETAILS_ID_PARAM, steamAppId)
                    .queryParam(
                        STEAM_APP_DETAILS_LANGUAGE_PARAM, STEAM_APP_DETAILS_LANGUAGE_ENGLISH)
                    .queryParam(STEAM_APP_DETAILS_CURRENCY_PARAM, STEAM_APP_DETAILS_CURRENCY_DOLLAR)
                    .build())
        .retrieve()
        .bodyToMono(SteamAppDetailsResponse.class)
        .retryWhen(
            Retry.fixedDelay(
                webClientProperties.getRetry().getAmount(),
                webClientProperties.getRetry().getDelay()));
  }
}
```

Додаток В. Клас SteamAppDetailsResponseDeserializer

SteamAppDetailsResponseDeserializer.java

```java
package ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.response;

import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.deser.std.StdDeserializer;
import com.fasterxml.jackson.databind.node.JsonNodeType;
import com.fasterxml.jackson.databind.node.ObjectNode;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class SteamAppDetailsResponseDeserializer extends StdDeserializer<SteamAppDetailsResponse> {

  private static final Logger log = LogManager.getLogger(SteamAppDetailsResponseDeserializer.class);

  public SteamAppDetailsResponseDeserializer() { super(SteamAppDetailsResponse.class); }

  @Override
  public SteamAppDetailsResponse deserialize(JsonParser jsonParser, DeserializationContext ctx)
      throws IOException {
    JsonNode jsonNode = ctx.readTree(jsonParser);
    List<JsonNode> singleNode = new ArrayList<>(1);
    jsonNode.fields().forEachRemaining(entry -> singleNode.add(entry.getValue()));

    final JsonNode responseDataNode = singleNode.get(0);
    final boolean isSuccessful = responseDataNode.get("success").asBoolean();
    if (isSuccessful) {
      final ObjectNode mutableDataNode = ((ObjectNode) responseDataNode.get("data"));
      setNullObjectIfNodeIsArray(mutableDataNode, "pc_requirements");
      setNullObjectIfNodeIsArray(mutableDataNode, "mac_requirements");
      setNullObjectIfNodeIsArray(mutableDataNode, "linux_requirements");
      return convertToSteamAppDetailsResponse(ctx, responseDataNode);
    } else {
      return SteamAppDetailsResponse.builder()
          .responseData(
              SteamAppDetailsResponse.ResponseData.builder().success(Boolean.FALSE).build())
          .build();
    }
  }

  private void setNullObjectIfNodeIsArray(ObjectNode responseDataNode, String fieldName) {
    Optional.ofNullable(responseDataNode.get(fieldName))
        .map(JsonNode::getNodeType)
        .filter(nodeType -> nodeType.equals(JsonNodeType.ARRAY))
        .ifPresent(ignored -> responseDataNode.putNull(fieldName));
  }

  private SteamAppDetailsResponse convertToSteamAppDetailsResponse(
      DeserializationContext ctx, JsonNode node) {
    try {
      SteamAppDetailsResponse.ResponseData responseData =
          ctx.readTreeAsValue(node, SteamAppDetailsResponse.ResponseData.class);
      return SteamAppDetailsResponse.builder().responseData(responseData).build();
    } catch (Exception ex) {
      log.error("Error occurred while deserializing SteamAppDetailsResponse", ex);
      return SteamAppDetailsResponse.builder()
          .responseData(
              SteamAppDetailsResponse.ResponseData.builder()
                  .success(Boolean.FALSE)
                  .errorData(
                      SteamAppDetailsResponse.ErrorData.builder().message(ex.getMessage()).build())
                  .build())
          .build();
    }
  }
}
```

Додаток Г. Клас SteamDataCsvFileWriter

```java
package ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.csv.writer;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.*;
import java.util.Objects;
import java.util.Optional;
import org.apache.commons.csv.CSVPrinter;
import org.apache.commons.lang3.tuple.Pair;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.csv.SteamDataProperties;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.response.SteamAppDetailsResponse;

public class SteamDataCsvFileWriter extends AbstractCsvFileWriter<SteamAppDetailsResponse> {

  private final CSVPrinter steamAppDetailsCsvPrinter;
  private final CSVPrinter steamAppDetailsNotParsedCsvPrinter;
  private final CSVPrinter steamAppDetailsNoDataCsvPrinter;
  private final CSVPrinter steamProcessedAppsIdsCsvPrinter;
  private final ObjectMapper objectMapper;

  public SteamDataCsvFileWriter(final SteamDataProperties steamDataProperties) {
    this.steamAppDetailsCsvPrinter = createCSVPrinter(steamDataProperties.getSteamAllAppsDetails());
    this.steamAppDetailsNotParsedCsvPrinter =
        createCSVPrinter(steamDataProperties.getSteamAllAppsDetailsNotParsed());
    this.steamAppDetailsNoDataCsvPrinter =
        createCSVPrinter(steamDataProperties.getSteamAllAppsDetailsNoData());
    this.steamProcessedAppsIdsCsvPrinter =
        createCSVPrinter(steamDataProperties.getSteamProcessedAppsIds());

    this.objectMapper = new ObjectMapper();
  }

  @Override
  public void write(Pair<Long, SteamAppDetailsResponse> pair) {
    if (Objects.isNull(pair.getRight().getResponseData().getErrorData())) {
      if (pair.getRight().getResponseData().getSuccess()) {
        writeSuccessfulSteamAppDetails(pair);
      } else {
        writeNoDataSteamAppDetails(pair);
      }
    } else {
      writeNotParsedSteamAppDetails(pair);
    }
    writeProcessedSteamAppId(pair);
  }

  @Override
  public void close() throws IOException {
    steamAppDetailsCsvPrinter.close();
    steamAppDetailsNotParsedCsvPrinter.close();
    steamAppDetailsNoDataCsvPrinter.close();
    steamProcessedAppsIdsCsvPrinter.close();
  }

  private void writeSuccessfulSteamAppDetails(Pair<Long, SteamAppDetailsResponse> pair) {
    System.out.printf("Saving successful steamAppDetails for steamAppId=[%s]...\n", pair.getLeft());
    try {
      final SteamAppDetailsResponse.AppDetails appDetails =
          pair.getRight().getResponseData().getData();
      steamAppDetailsCsvPrinter.printRecord(
          appDetails.getType(),
          appDetails.getName(),
          appDetails.getSteamAppId(),
          appDetails.getRequiredAge(),
          appDetails.getIsFree(),
          appDetails.getDetailedDescription(),
          appDetails.getAboutTheGame(),
          appDetails.getShortDescription(),
          appDetails.getSupportedLanguages(),
          appDetails.getHeaderImage(),
          appDetails.getWebsite(),
```

```java
Optional.ofNullable(appDetails.getPcRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getMinimum)
    .orElse( null),
Optional.ofNullable(appDetails.getPcRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getRecommended)
    .orElse( null),
Optional.ofNullable(appDetails.getMacRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getMinimum)
    .orElse( null),
Optional.ofNullable(appDetails.getMacRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getRecommended)
    .orElse( null),
Optional.ofNullable(appDetails.getLinuxRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getMinimum)
    .orElse( null),
Optional.ofNullable(appDetails.getLinuxRequirements())
    .map(SteamAppDetailsResponse.PcRequirements::getRecommended)
    .orElse( null),
toJsonArray(appDetails.getDevelopers()),
toJsonArray(appDetails.getPublishers()),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getCurrency)
    .orElse( null),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getInitialPrice)
    .orElse( null),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getFinalPrice)
    .orElse( null),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getDiscountPercent)
    .orElse( null),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getInitialFormatted)
    .orElse( null),
Optional.ofNullable(appDetails.getPriceOverview())
    .map(SteamAppDetailsResponse.PriceOverview::getFinalFormatted)
    .orElse( null),
toJsonArray(appDetails.getPackages()), // array
toJsonArray(appDetails.getPackageGroups()), // array
```

```java
Optional.ofNullable(appDetails.getPlatforms())
    .map(SteamAppDetailsResponse.Platforms::getWindows)
    .orElse( null),
Optional.ofNullable(appDetails.getPlatforms())
    .map(SteamAppDetailsResponse.Platforms::getMac)
    .orElse( null),
Optional.ofNullable(appDetails.getPlatforms())
    .map(SteamAppDetailsResponse.Platforms::getLinux)
    .orElse( null),
Optional.ofNullable(appDetails.getMetacritic())
    .map(SteamAppDetailsResponse.Metacritic::getScore)
    .orElse( null),
Optional.ofNullable(appDetails.getMetacritic())
    .map(SteamAppDetailsResponse.Metacritic::getUrl)
    .orElse( null),
toJsonArray(appDetails.getCategories()), // array
toJsonArray(appDetails.getGenres()), // array
toJsonArray(appDetails.getScreenshots()), // array
Optional.ofNullable(appDetails.getRecommendations())
    .map(SteamAppDetailsResponse.Recommendations::getTotal)
    .orElse( null),
Optional.ofNullable(appDetails.getReleaseDate())
    .map(SteamAppDetailsResponse.ReleaseDate::getComingSoon)
    .orElse( null),
Optional.ofNullable(appDetails.getReleaseDate())
    .map(SteamAppDetailsResponse.ReleaseDate::getDate)
    .orElse( null),
Optional.ofNullable(appDetails.getSupportInfo())
    .map(SteamAppDetailsResponse.SupportInfo::getUrl)
    .orElse( null),
Optional.ofNullable(appDetails.getSupportInfo())
    .map(SteamAppDetailsResponse.SupportInfo::getEmail)
    .orElse( null),
appDetails.getBackground(),
appDetails.getBackgroundRaw(),
toJsonArray(
    Optional.ofNullable(appDetails.getContentDescriptors())
        .map(SteamAppDetailsResponse.ContentDescriptors::getIds)
        .orElse( null)), // array
Optional.ofNullable(appDetails.getContentDescriptors())
    .map(SteamAppDetailsResponse.ContentDescriptors::getNotes)
```

```
153                .orElse( other: null));
154        } catch (IOException e) {
155          throw new RuntimeException(e);
156        }
157      }
158
         8 usages
159      private String toJsonArray(Object[] array) {
160        return Optional.ofNullable(array).map(this::writeArrayAsJsonString).orElse( other: null);
161      }
162
         1 usage
163      private String writeArrayAsJsonString(Object[] array) {
164        try {
165          return objectMapper.writeValueAsString(array);
166        } catch (JsonProcessingException e) {
167          throw new RuntimeException(e);
168        }
169      }
170
         1 usage
171      private void writeNoDataSteamAppDetails(Pair<Long, SteamAppDetailsResponse> pair) {
172        System.out.printf(
173            "Saving steamAppDetails with no data for steamAppId=[%s]...\n", pair.getLeft());
174        try {
175          steamAppDetailsNoDataCsvPrinter.printRecord(pair.getLeft());
176        } catch (IOException e) {
177          throw new RuntimeException(e);
178        }
179      }
180
```

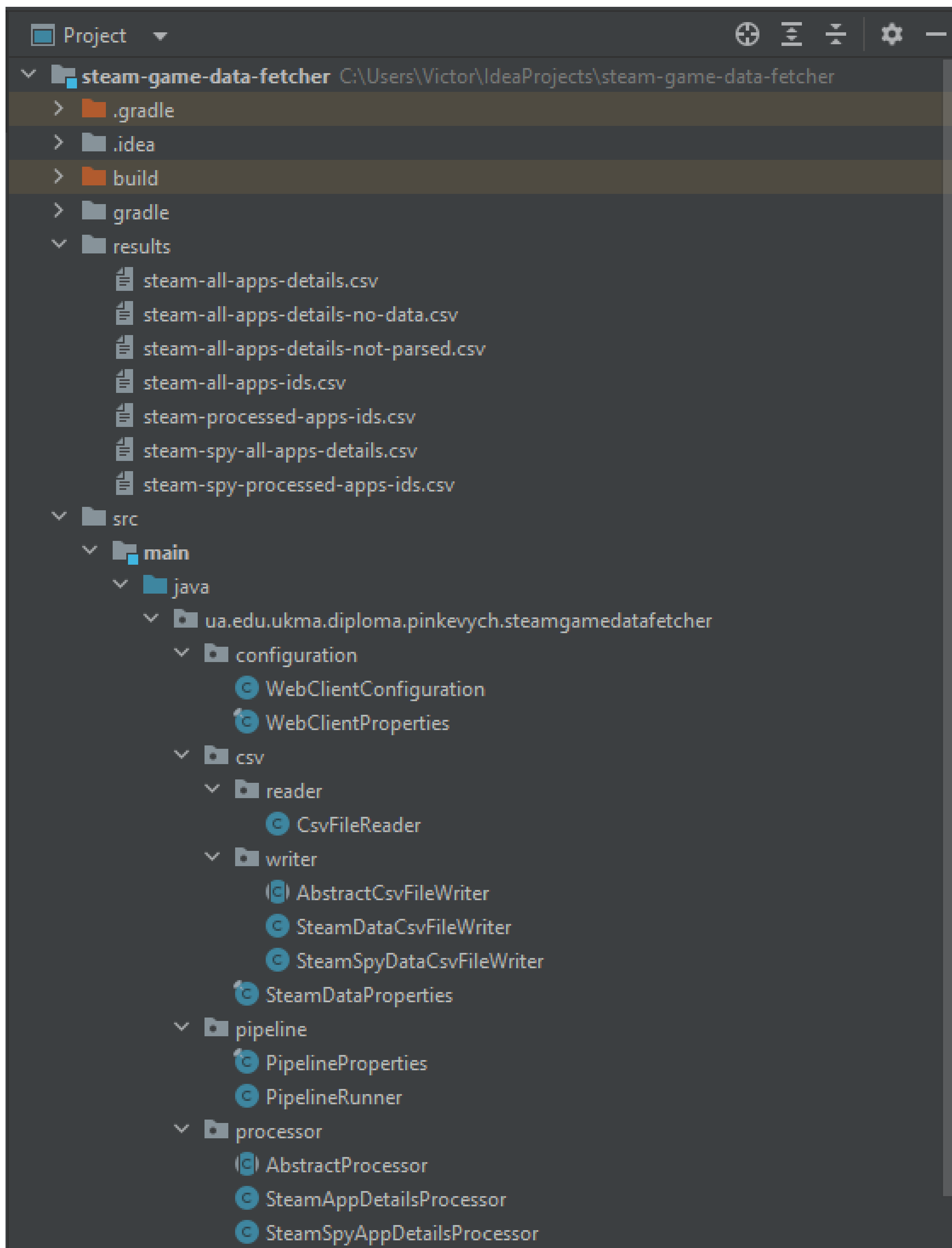
```
         1 usage
181      private void writeNotParsedSteamAppDetails(Pair<Long, SteamAppDetailsResponse> pair) {
182        System.out.printf("Saving not parsed steamAppDetails for steamAppId=[%s]...\n", pair.getLeft());
183        try {
184          steamAppDetailsNotParsedCsvPrinter.printRecord(
185              pair.getLeft(), pair.getRight().getResponseData().getErrorData().getMessage());
186        } catch (IOException e) {
187          throw new RuntimeException(e);
188        }
189      }
190
         1 usage
191      private void writeProcessedSteamAppId(Pair<Long, SteamAppDetailsResponse> pair) {
192        System.out.printf("Saving processed steamAppId=[%s]...\n", pair.getLeft());
193        try {
194          steamProcessedAppsIdsCsvPrinter.printRecord(pair.getLeft());
195        } catch (IOException e) {
196          throw new RuntimeException(e);
197        }
198      }
199    }
200
```

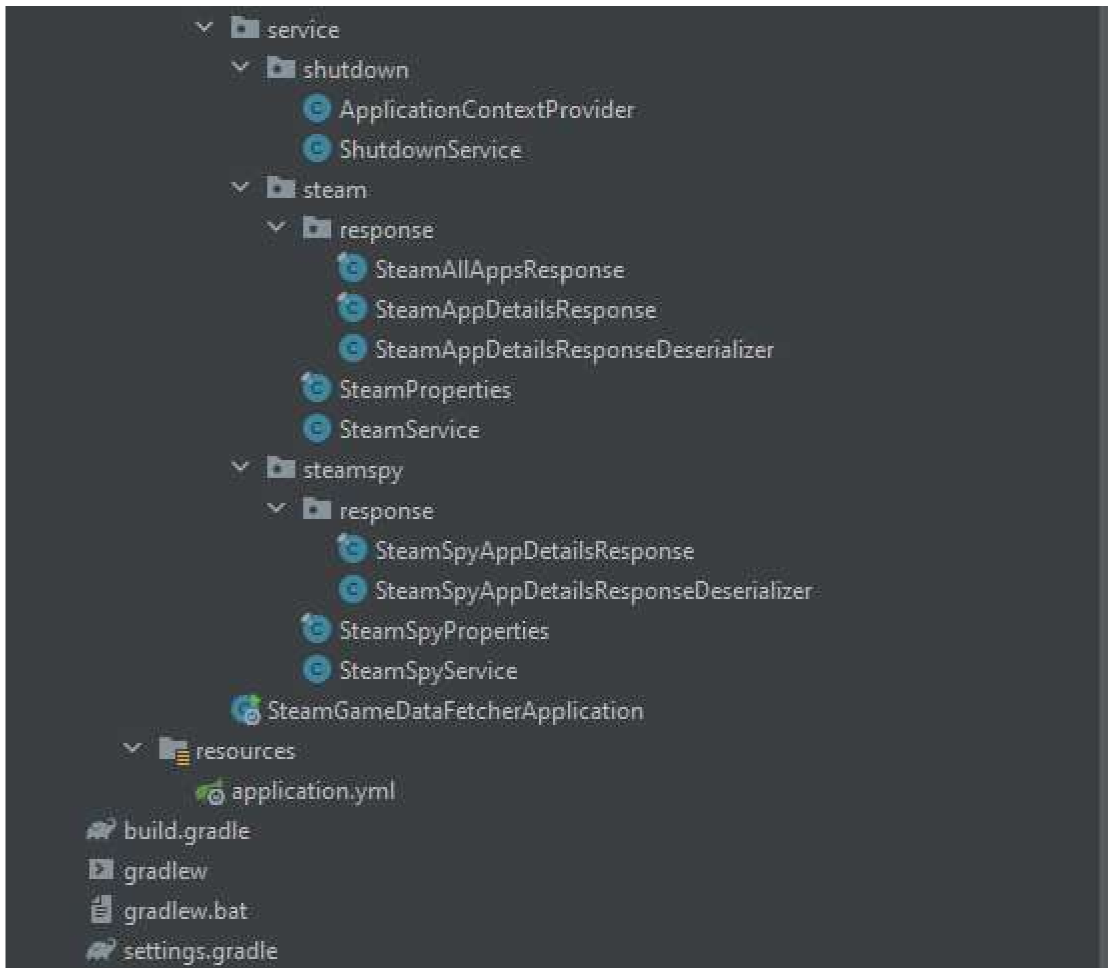
Додаток Д. Клас SteamAppDetailsProcessor

SteamAppDetailsProcessor.java

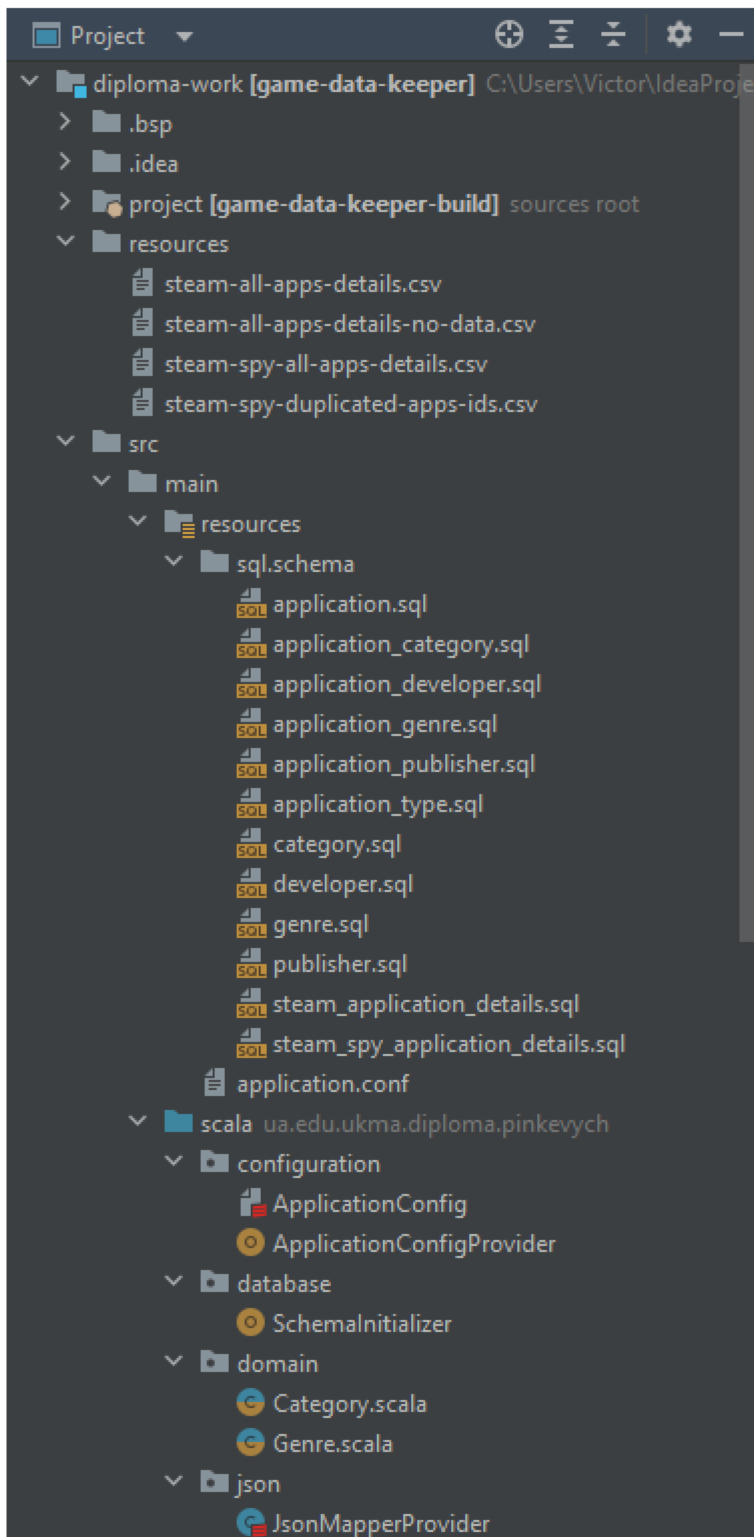
```java
package ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.processor;

import java.time.Duration;
import org.apache.commons.lang3.tuple.Pair;
import org.reactivestreams.Publisher;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.csv.SteamDataProperties;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.csv.writer.AbstractCsvFileWriter;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.csv.writer.SteamDataCsvFileWriter;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.SteamService;
import ua.edu.ukma.diploma.pinkevych.steamgamedatafetcher.service.steam.response.SteamAppDetailsResponse;

@Component
public class SteamAppDetailsProcessor extends AbstractProcessor<SteamAppDetailsResponse> {

  private static final Long STEAM_API_REQUEST_DELAY_MILLS = 1600L;

  private final SteamService steamService;
  private final AbstractCsvFileWriter<SteamAppDetailsResponse> csvFileWriter;

  public SteamAppDetailsProcessor(
      SteamService steamService, SteamDataProperties steamDataProperties) {
    this.steamService = steamService;
    this.csvFileWriter = new SteamDataCsvFileWriter(steamDataProperties);
  }

  @Override
  protected Publisher<Pair<Long, SteamAppDetailsResponse>> doProcess(
      Flux<Long> appIds, AbstractCsvFileWriter<SteamAppDetailsResponse> csvFileWriter) {
    return appIds
        .delayElements(Duration.ofMillis(STEAM_API_REQUEST_DELAY_MILLS))
        .flatMap(
            appId ->
                steamService
                    .getSteamAppDetails(appId)
                    .map(steamAppDetailsResponse -> Pair.of(appId, steamAppDetailsResponse)))
        .doOnNext(csvFileWriter::write);
  }

  @Override
  protected AbstractCsvFileWriter<SteamAppDetailsResponse> getCsvFileWriter() { return csvFileWriter; }
}
```

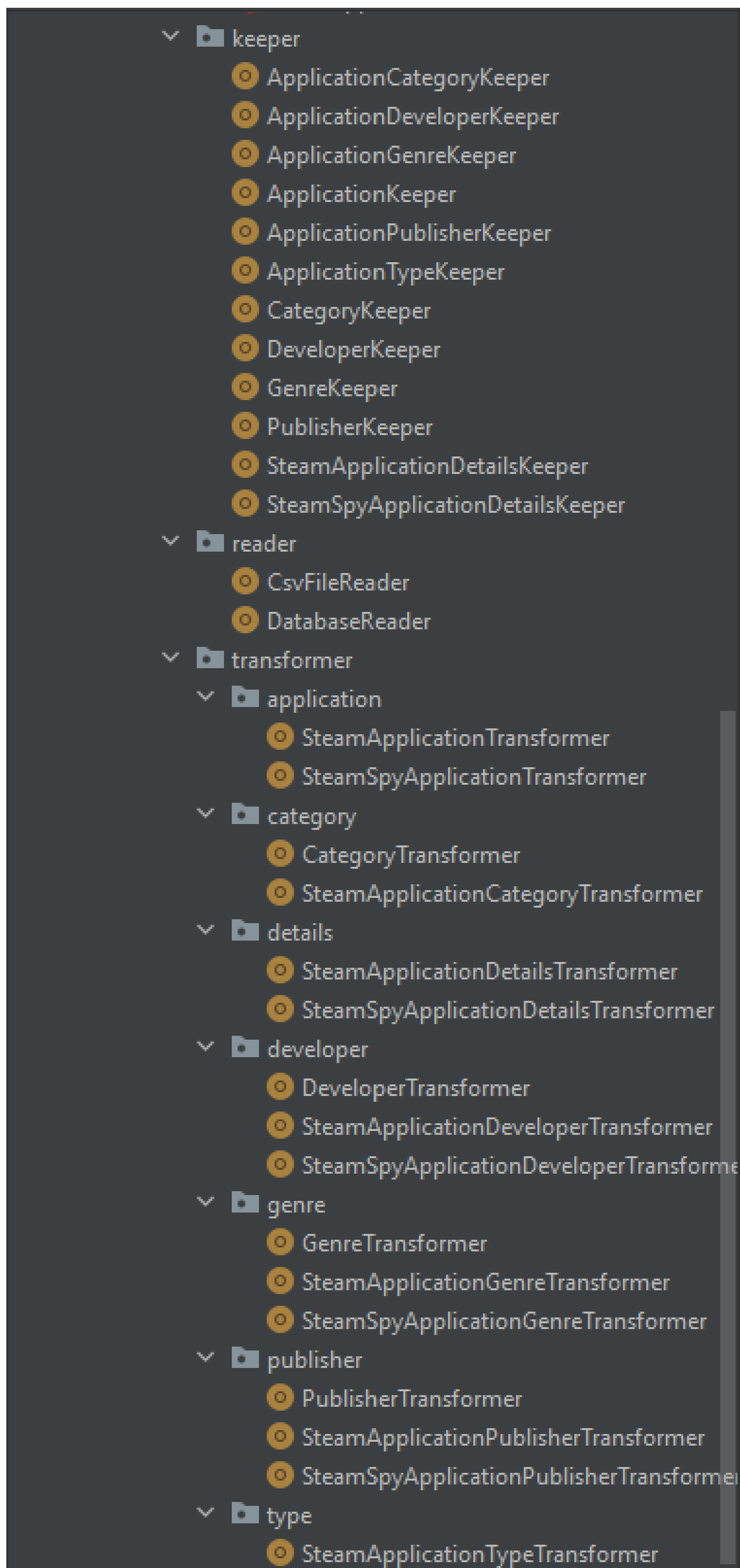
Додаток Е. Файлова структура проекту застосунку для збору даних

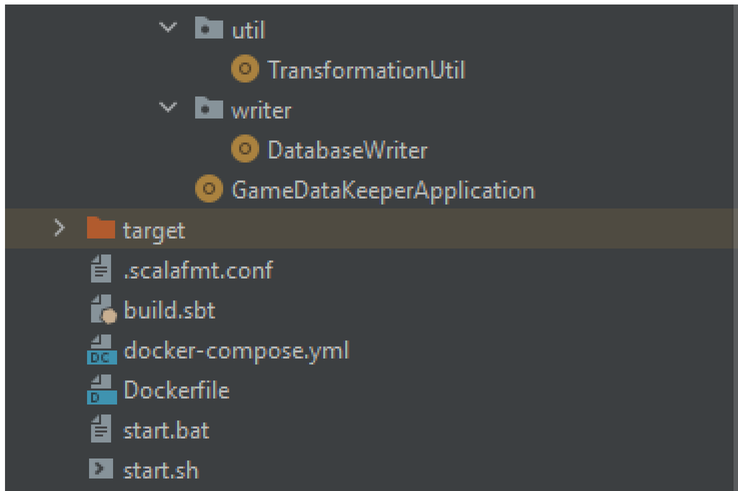



service
shutdown
ApplicationContextProvider
ShutdownService
steam
response
SteamAllAppsResponse
SteamAppDetailsResponse
SteamAppDetailsResponseDeserializer
SteamProperties
SteamService
steamspy
response
SteamSpyAppDetailsResponse
SteamSpyAppDetailsResponseDeserializer
SteamSpyProperties
SteamSpyService
SteamGameDataFetcherApplication
resources
application.yml
build.gradle
gradlew
gradlew.bat
settings.gradle

Додаток Є. Файлова структура проекту застосунку для обробки даних




keeper
ApplicationCategoryKeeper
ApplicationDeveloperKeeper
ApplicationGenreKeeper
ApplicationKeeper
ApplicationPublisherKeeper
ApplicationTypeKeeper
CategoryKeeper
DeveloperKeeper
GenreKeeper
PublisherKeeper
SteamApplicationDetailsKeeper
SteamSpyApplicationDetailsKeeper
reader
CsvFileReader
DatabaseReader
transformer
application
SteamApplicationTransformer
SteamSpyApplicationTransformer
category
CategoryTransformer
SteamApplicationCategoryTransformer
details
SteamApplicationDetailsTransformer
SteamSpyApplicationDetailsTransformer
developer
DeveloperTransformer
SteamApplicationDeveloperTransformer
SteamSpyApplicationDeveloperTransforme
genre
GenreTransformer
SteamApplicationGenreTransformer
SteamSpyApplicationGenreTransformer
publisher
PublisherTransformer
SteamApplicationPublisherTransformer
SteamSpyApplicationPublisherTransformer
type
SteamApplicationTypeTransformer


util
TransformationUtil
writer
DatabaseWriter
GameDataKeeperApplication
target
.scalafmt.conf
build.sbt
docker-compose.yml
Dockerfile
start.bat
start.sh