Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів програмної реалізації Cosmos DB API на платформі .NET
(тема)

Виконав:
Випускник 2 курсу, групи ІПЗм-21-1
Андрущенко М.О.
(прізвище, ініціали)

Спеціальність 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми Освітньо-наукова
(код і повна назва спеціальності)

Керівник доц. Мазурова О.О.
(посада, прізвище)

Допускається до захисту
Зав. кафедри _____ З.В. Дудар
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Програмної інженерії
(повна назва)
Рівень вищої освіти другий (магістерський)
(повна назва)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«___»_____ 2023 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента *Андрущенко Миколі Олександровичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження методів програмної реалізації Cosmos DB API на платформі .NET*

затверджена наказом університету від 29.03.2023 № 302ст

2. Термін подання студентом роботи до екзаменаційної комісії "_18_"_05_2023 р.

3. Вихідні данні до роботи *опис досліджуваних Cosmos DB API, вимоги до розробки схеми бази даних для проведення досліджень за обраною предметною областю, мови програмування C#, технології .NET 6.0, СУБД Cosmos DB, середовища розробки Visual Studio 2022.*

4. Перелік питань, що потрібно опрацювати в роботі *аналіз та порівняння існуючих Cosmos DB API, вибір підходящих API для дослідження, проектування логічної моделі даних для проведення експериментальних досліджень, написання програмних рішень, проведення експериментів та аналіз отриманих результатів.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз проблемної області та постановка задачі	23.01 – 14.02.23	Виконано
2	Аналіз та вибір API для дослідження	15.02 – 24.02.23	Виконано
3	Аналіз та моделювання предметної області	17.02 – 28.02.23	Виконано
4	Планування експериментів	25.02 – 28.02.23	Виконано
5	Програмна реалізація кожного з обраних для дослідження API	25.02 – 01.04.23	Виконано
6	Експериментальні дослідження	02.04 – 20.04.23	Виконано
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	20.04 – 23.04.23	Виконано
8	Написання та оформлення статті та тез доповіді	17.04 – 23.04.23	Виконано
9	Підготовка пояснювальної записки	01.04 – 26.04.23	Виконано
10	Підготовка презентації та доповіді	26.04 – 2.05.23	Виконано
11	Нормоконтроль	3.05 – 08.05.23	Виконано
12	Рецензування	08.05 – 14.05.23	Виконано
13	Занесення диплома в електронний архів	15.05.2023	Виконано
14	Попередній захист	15.05.2023	Виконано
15	Допуск до захисту у зав. кафедри	18.05.2023	Виконано

Дата видачі завдання «23» січня 2023р.

Студент _____ Андрущенко М.О.
(підпис) (прізвище, ініціали)

Керівник роботи _____ доц. Мазурова О.О.
(підпис) (прізвище, ініціали)

РЕФЕРАТ

Кваліфікаційна робота містить: 104 с., 37 рис., 14 табл., 26 джерел.

NOSQL БАЗА ДАНИХ, ХМАРНА БАЗА ДАНИХ, AZURE CLOUD, AZURE COSMOS DB, COSMOS DB API FOR NOSQL, COSMOS DB API FOR MONGODB, COSMOSCLIENT, ENTITY FRAMEWORK CORE COSMOS, MONGO CLIENT, .NET, C#, VISUAL STUDIO 2022.

Об’єктом дослідження є хмарні бази даних та їх програмні інтерфейси.

Метою роботи є проведення дослідження продуктивності методів програмної реалізації Cosmos DB API на платформі .NET.

Методами розробки та проектування є аналіз проблемної області дослідження, вибір Cosmos DB API для проведення дослідження шляхом вирішення багатокритеріальної задачі прийняття рішень. Проектування логічної схеми даних для обраної предметної області, планування проведення експериментальних досліджень, написання програм та проведення досліджень. Аналіз результатів проведених досліджень та формування рекомендацій щодо використання того чи іншого програмного підходу.

У результаті кваліфікаційної роботи було розроблено три програми, по одній для кожного з досліджуваних Cosmos DB API: Cosmos DB API for NoSQL (Cosmos Client), Cosmos DB API for NoSQL (Entity Framework Core Cosmos) та Cosmos DB for MongoDB (Mongo Client).

NON-RELATIONAL DATABASE, CLOUD DATABASE, AZURE CLOUD, AZURE COSMOS DB, COSMOS DB API FOR NOSQL, COSMOS DB API FOR MONGODB, COSMOSCLIENT, ENTITY FRAMEWORK CORE COSMOS, MONGO CLIENT, .NET, C#, VISUAL STUDIO 2022.

The object of research is cloud databases and their software interfaces.

The purpose of the work is to conduct a study of the productivity of methods of software implementation of the Cosmos DB API on the .NET platform.

The development and design methods are the analysis of the problem area of the study, the selection of the Cosmos DB API for conducting the study by solving a multi-criteria decision-making problem. Designing a logical data scheme for the selected subject area, planning experimental research, writing programs and conducting research. Analysis of the results of research and the formation of recommendations for the use of one or another software approach.

As a result of the qualification work, three programs were developed, one for each of the studied Cosmos DB APIs: Cosmos DB API for NoSQL (Cosmos Client), Cosmos DB API for NoSQL (Entity Framework Core Cosmos) and Cosmos DB for MongoDB (Mongo Client).

Умови публікації пояснювальної записки

Я, Андрущенко Микола Олександрович, студент гр. ІПЗм-21-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів програмної реалізації Cosmos DB API на платформі .NET», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Сучасний підхід до розробки програмних систем спрямований на те, щоб спростити програму, зробити код простішим та дешевшим у підтримці в майбутньому. Саме тому зараз дуже популярні хмарні розподілені системи, які складаються з багатьох маленьких програм, які відповідають лише за одну функцію. Такі програми називаються сервісами, а архітектура таких систем називається мікросервісною. Така архітектура легка у підтримці та розробці, за кожен сервіс частіше всього відповідає окрема команда. Це досягається шляхом розподілення задач, які повинен виконувати окремий сервіс. Саме тому більшість нових систем розробляється саме з такою архітектурою.

Такі гнучкі архітектури легко масштабувати, саме тому для них потрібні легко масштабовані бази даних. Саме такою хмарна Cosmos DB API, яка має шість API, високу доступніть з різних регіонів, геореплікацію та просте масштабування.

Дослідження методів програмної реалізації Cosmos DB API на платформі .NET є важливим завданням з декількох причин.

По-перше, .NET є однією з найбільш популярних платформ для програмування на мові C#, яку часто використовують для розробки різних додатків від веб-сайтів до мобільних додатків. Cosmos DB API має підтримку для різних мов програмування, включаючи C#, тому програмісти можуть використовувати .NET для розробки додатків, які працюють з Cosmos DB.

По-друге, дослідження методів програмної реалізації Cosmos DB API на платформі .NET може допомогти програмістам виявити і виправити помилки та вдосконалити процес розробки. Наприклад, виявлення проблем в реалізації API може допомогти у покращенні продуктивності і зниженні витрат ресурсів на сервері.

По-третє, дослідження методів програмної реалізації Cosmos DB API на платформі .NET може допомогти програмістам зрозуміти, як використовувати

різні функції та можливості, що надає Cosmos DB API, та як краще застосовувати їх у конкретних випадках.

Таким чином, дослідження методів програмної реалізації Cosmos DB API на платформі .NET може допомогти покращити якість програмного забезпечення, зменшити витрати ресурсів та зробити процес розробки більш ефективним. Під час виконання кваліфікаційної роботи був проведений аналіз проблемної області хмарних баз даних на основі публікацій світових та вітчизняних фахівців в проблемній області (див. додаток А). Розглянуті та проаналізовані усі Cosmos DB API, а також розглянуті методи програмної реалізації обраних для дослідження Cosmos DB API.

Також, було спроектовано процес проведення експерименту дослідження, що в свою чергу включає аналіз предметної області БД, проектування логічної моделі даних, розробка об’єктно-орієнтованої моделі для роботи з даними, описані досліджувані запити до БД. Сформовані метрики для порівняння швидкості виконання запитів, їх ефективності та вартості.

Робота пройшла успішну перевірку на академічну доброчесність (див. додаток Б).

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток В). За результатами роботи були створені тези доповіді на двадцять сьомий міжнародний молодіжний форум «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ В XXI ст.» , а також, підготовлено до подачі наукову статтю «Дослідження методів програмної реалізації Cosmos DB API на платформі .NET» у науковому журналі «Сучасний стан наукових досліджень і технологій в промисловості» (див. додаток Г).

Також, робота перевірена на відповідність вимогам оформлення (див. додаток Д).

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області дослідження

Платформа .NET є однією з найпопулярніших технологій для розробки програмного забезпечення на сьогоднішній день. Вона надає розробникам широкий спектр інструментів для створення різноманітних додатків - від веб-сайтів до мобільних додатків та ігор.

Актуальність дослідження для платформи .NET полягає в тому, що ця технологія постійно розвивається та вдосконалюється. Це означає, що розробникам потрібно постійно оновлювати свої знання та навички, щоб бути в курсі останніх тенденцій та можливостей платформи.

Крім того, .NET є платформою з відкритим кодом, що дає можливість розробникам з усього світу співпрацювати та вносити свій внесок у розвиток цієї технології. Таким чином, дослідження у галузі .NET може допомогти збільшити співпрацю та знання в цій галузі, що в свою чергу сприятиме подальшому розвитку технології та створенню нових інноваційних продуктів на її базі.

Бази даних NoSQL були створені у відповідь на обмеження традиційної технології реляційних баз даних. Порівняно з реляційними базами даних, бази даних NoSQL часто більш масштабовані та забезпечують чудову продуктивність. Крім того, гнучкість і простота використання їхніх моделей даних можуть прискорити розробку порівняно з реляційною моделлю, особливо в середовищі хмарних обчислень.

Кожен окремий тип бази даних NoSQL має різні переваги. Розглянемо спільні фундаментальні характеристики, які дозволяють їм:

– обробляти великі обсяги даних на високій швидкості за допомогою масштабованої архітектури;

– зберігати неструктуровані, напівструктуровані або структуровані дані;

– легко оновлювати структуру файлів, що зберігаються;

– бути простими у використанні для розробників;

– використовувати всі перевагами хмари, щоб забезпечити нульовий час простою.

Ці можливості надають користувачам багато переваг порівняно з реляційними базами даних.

Бази даних SQL найчастіше реалізуються в масштабованій архітектурі, яка базується на використанні все більших комп'ютерів з більшою кількістю ЦП і пам'яті для підвищення продуктивності [1].

Бази даних NoSQL були створені в епоху Інтернету та хмарних обчислень, що дозволило легше реалізувати архітектуру масштабування. У масштабованій архітектурі масштабованість досягається шляхом розподілу сховища даних і роботи з обробки даних на великий кластер комп'ютерів. Щоб збільшити потужність, до кластера додається більше комп'ютерів [2].

Ще одна велика відмінність між SQL і NoSQL полягає в їх масштабованості. Більшість баз даних SQL є вертикально масштабованими, що означає, що ви можете збільшити навантаження на один сервер, збільшивши такі компоненти, як оперативна пам'ять, SSD або процесор.

А от бази даних NoSQL є горизонтально масштабованими, що означає, що вони можуть обробляти збільшений трафік, просто додавши більше серверів до бази даних. Бази даних NoSQL мають можливість ставати більшими та потужнішими, що робить їх кращим вибором для великих або постійно змінюваних наборів даних.

Ця масштабована архітектура особливо безболісна для реалізації в середовищах хмарних обчислень, де нові комп'ютери та сховище можна легко додати до кластера.

Розширена архітектура систем NoSQL забезпечує чіткий шлях до масштабованості, коли обсяг даних або трафік зростає. Досягнення того самого типу масштабованості з базами даних SQL може бути дорогим, вимагати великої кількості інженерних робіт або може бути неможливим.

Реляційні бази даних зберігають дані в структурованих таблицях, які мають заздалегідь визначену схему. Щоб використовувати реляційні бази даних,

необхідно розробити модель даних, а потім дані трансформувати та завантажувати в базу даних.

Коли дані використовуються в програмах, дані повинні бути отримані за допомогою SQL і адаптовані до форми, що використовується в програмі. Потім, коли дані записуються назад, їх потрібно знову перетворити назад у реляційні таблиці.

Бази даних NoSQL виявилися популярними, оскільки вони дозволяють зберігати дані у спосіб, який легше зрозуміти або ближче до способу використання даних програмами. Потрібно менше перетворень, коли дані зберігаються або витягуються для використання. Багато різних типів даних, структурованих, неструктурованих або напівструктурованих, можна зберігати та витягувати легше [3].

Крім того, структури багатьох NoSQL баз даних гнучкі та знаходяться під контролем розробників, що полегшує адаптацію бази даних до нових форм даних. Це усуває вузькі місця в процесі розробки, пов’язані з проханням до адміністратора бази даних перепроектувати базу даних SQL.

Бази даних NoSQL підтримують широко використовувані формати даних:

– великі дані всіх видів — текстові дані, а також дані часових рядів;
– файли JSON, які є вкладеними, зрозумілими для людини файлами, що складаються з пар імен і значень. Цей формат може фіксувати дуже складні ієрархічні структури «батько-нащадок», які можна ефективно зберігати в базах даних документів;
– прості двійкові значення, списки, карти та рядки можна обробляти з високою швидкістю в сховищах ключ-значення;
– розріджені дані можна ефективно зберігати в базах даних зі стовпцями, де нульові значення взагалі не займають місця. Вони також ефективні для інформації, яка не змінюється часто (енергонезалежні дані);
– мережі взаємопов’язаної інформації можуть зберігатися в графових базах даних.

Прийняття NoSQL баз даних насамперед було зумовлене захопленням розробників, яким легше створювати різні типи програм порівняно з використанням реляційних баз даних.

Бази даних документів використовують JSON як спосіб перетворити дані на щось більше схоже на код. Це дозволяє розробнику контролювати структуру даних.

Крім того, бази даних NoSQL зберігають дані у формах, близьких до типу об’єктів даних, що використовуються в програмах, тому потрібно менше перетворень під час переміщення даних у бази даних і з них.

Бази даних NoSQL можуть зберігати дані у рідних форматах, що означає, що розробникам не потрібно адаптувати дані до сховища. Зберігання даних «як є» означає відсутність інтерфейсної системи ETL для переведення напівструктурованих даних у формати рядків і стовпців, а також менше додатків для розробки або покупки для запуску нової бази даних.

Навколо більшості баз даних NoSQL є сильна спільнота розробників. Це означає, що є екосистема доступних інструментів і спільнота інших розробників, з якими можна спілкуватися.

Архітектура масштабування, яку використовують більшість баз даних NoSQL, не лише забезпечує чіткий шлях до масштабування для розміщення величезних наборів даних і великого обсягу трафіку. Надання бази даних за допомогою кластера комп’ютерів також дозволяє автоматично розширювати та скорочувати ємність бази даних.

Крім того, багато баз даних NoSQL можна оновити, що дозволяє змінювати структуру бази даних без простою.

На сьогодні існує багато популярних NoSQL баз даних. Нам необхідно обрати найкращу для подальшого дослідження, перетворимо цей процес на вирішення багатокритеріальної задачі прийняття рішень. Так, в якості альтернатив для такої задачі розглянемо:

– Azure Cosmos DB;
– AWS Dynamo DB;

– MongoDB Atlas;
– GCP Bigtable;
– GCP Cloud Datastore.

Сформуємо множину критеріїв вибору:

– популярність серед розробників – для отримання рейтингу популярності серед розробників було використано сайт https://db-engines.com [4];
– вартість 1 млн запитів на читання даних – параметри вартості відображені на сторінках з цінами кожної з баз даних:
    1) Azure Cosmos DB pricing [5];
    2) Amazon DynamoDB pricing [6];
    3) MongoDB Pricing [7];
    4) Cloud Bigtable pricing [8];
    5) Firestore in Datastore mode pricing [9];
    6) NoSQL databases comparison: Cosmos DB vs DynamoDB vs Cloud Datastore and Bigtable [10];
– кількість API та інших методів доступу – для отримання даних було використано сайт https://db-engines.com;
– кількість мов програмування які підтримує – для отримання даних було використано сайт https://db-engines.com;
– кількість концепцій узгодженості які підтримуються – для отримання даних було використано сайт https://db-engines.com.

Після проведення збору необхідних даних отримаємо наступні результати:

Azure Cosmos DB має рейтинг: 37,95, вартість: 0.25$/1 млн запитів на читання даних, кількість API та інших методів доступу: 6 (Azure Cosmos DB for NoSQL, Azure Cosmos DB for MongoDB, Azure Cosmos DB for Apache Cassandra, Azure Cosmos DB for Table, Azure Cosmos DB for Apache Gremlin, Azure Cosmos DB for PostgreSQL), кількість мов програмування: 30 (JavaScript (Node.js), та всі мови, що підтримує MongoDB), кількість концепцій узгодженості які підтримуються: 5 (Bounded Staleness, Consistent Prefix, Eventual Consistency, Immediate Consistency, Session Consistency).

AWS Dynamo DB має рейтинг: 83,85, вартість: 0.25$/1 млн запитів на читання даних, кількість API та інших методів доступу: 1 (RESTful HTTP API [11]), кількість мов програмування: 10 (.Net, ColdFusion, Erlang, Groovy, Java, JavaScript, Perl, PHP, Python, Ruby), кількість концепцій узгодженості які підтримуються: 2 (Eventual Consistency, Immediate Consistency).

MongoDB Atlas має рейтинг: 469,33, вартість: 0.10$/1 млн запитів на читання даних, кількість API та інших методів доступу: 1 (proprietary protocol using JSON), кількість мов програмування: 29 (JavaScript (Node.js), Actionscript, C, C#, C++, Clojure, ColdFusion, D, Dart, Delphi, Erlang, Go, Groovy, Haskell, Java, JavaScript, Lisp, Lua, MatLab, Perl, PHP, PowerShell, Prolog, Python, R, Ruby, Rust, Scala, Smalltalk, Swift), кількість концепцій узгодженості які підтримуються: 2 (Eventual Consistency, Immediate Consistency).

GCP Bigtable має рейтинг: 5,01, вартість: 0.65$/1 млн запитів на читання даних, кількість API та інших методів доступу: 3 (gRPC (using protocol buffers) API, HappyBase (Python library), HBase compatible API (Java)), кількість мов програмування: 6 (C#, C++, Go, Java, JavaScript, (Node.js), Python), кількість концепцій узгодженості які підтримуються: 2 (Immediate consistency (for a single cluster), Eventual consistency (for two or more replicated clusters)).

GCP Cloud Datastore має рейтинг: 6,36, вартість: 0.30$/1 млн запитів на читання даних, кількість API та інших методів доступу: 2 (gRPC (using protocol buffers) API, RESTful HTTP/JSON API), кількість мов програмування: 7 (.Net,, Go, Java, JavaScript (Node.js), PHP, Python, Ruby), кількість концепцій узгодженості які підтримуються: 2 (Immediate Consistency or Eventual Consistency depending on type of query and configuration).

Сформулюємо шкали оцінок за критеріями:

– вартість $/1 млн запитів на читання даних:

1) 0.10$ – 0.10;
2) 0.25$ – 0.25;
3) 0.30$ – 0.30;
4) 0.65$ – 0.65.

– кількість концепцій узгодженості які підтримуються:

1) 1-2 концепцій узгодженості – 1;
2) 3-4 концепцій узгодженості – 5;
3) 5-6 концепцій узгодженості – 10.

Сформулюємо таблицю для моделювання задачі прийняття рішень (див. табл. 1).

Таблиця 1 – Моделювання задачі прийняття рішень щодо вибору NoSQL БД

	Популярність серед розробників	Вартість $/1 млн запитів на читання даних	Кількість API та інших методів доступу	Кількість мов програмування які підтримує	Кількість концепцій узгодженості які підтримуються
Azure Cosmos DB	37.95	0.25	6	30	5
AWS Dynamo DB	83.85	0.25	1	10	2
MongoDB Atlas	469.33	0.10	1	29	2
GCP Bigtable	5.01	0.65	3	6	2
GCP Cloud Datastore	6.36	0.30	2	7	2

Тепер потрібно зробити нормування по minmax та еталону (див. табл. 2).

Minmax буде використовуватись для розрахунків популярності серед розробників:

$$f = \frac{f\text{значення} - f\min}{f\max - f\min}.$$

Звичайне нормування по еталону буде робитись для кількості API та інших методів доступу, кількості мов програмування які підтримує, кількість концепцій узгодженості які підтримуються:

$$f = \frac{f\text{значення}}{f\text{еталон}}.$$

Чим нижча ціна, тим краще, тому для вартості $/1 млн запитів на читання даних ми будемо використовувати обернене еталонне нормування (див. табл. 3):

$$f = \frac{f\text{еталон}}{f\text{значення}}.$$

Таблиця 2 – Проведення нормування по minmax та еталону таблиці 1

Нормування по minmax та еталону					
	Популярність серед розробників	Вартість $/1 млн запитів на читання даних	Кількість API та інших методів доступу	Кількість мов програмування які підтримує	Кількість концепцій узгодженості які підтримуютьс я
Azure Cosmos DB	0.070942453	0.4	1	1	1
AWS Dynamo DB	0.169796692	0.4	0	0.166666667	0.1
MongoDB Atlas	1	1	0	0.958333333	0.1
GCP Bigtable	0	0.15384	0.4	0	0.1
GCP Cloud Datastore	0.002907478	0.33333	0.2	0.041666667	0.1

Таблиця 3 – Показники обмеження та еталонні значення за параметрами

Популярність (minmax)	
min	max
5.01	469.33
Кількість API та інших методів доступу (minmax)	
min	max
1	6
Кількість мов програмування які підтримує (minmax)	
min	max
6	30
Кількість концепцій узгодженості які підтримуються (еталон)	
еталон	10
Вартість $/1 млн запитів на читання даних (обернений еталон)	
еталон	0.1

За принципом Парето можна зменшити кількість варіантів, що розглядаються: “Варіант а краще варіанту b згідно з відношенням Парето, якщо а хоча б за одним критерієм краще ніж b, а по іншим критеріям не гірше, ніж b” (див. рис. 1.1).

Парето					
	Популярність серед розробників	Вартість $/1 млн запитів на читання даних	Кількість API та інших методів доступу	Кількість мов програмування які підтримує	Кількість концепцій узгодженості які підтримуються
Azure Cosmos DB	0.070942453	0.4	1	1	1
AWS Dynamo DB	0.169796692	0.4	0	0.166666667	0.1
MongoDB Atlas	1	1	0	0.958333333	0.1
GCP Bigtable	0	0.153846154	0.4	0	0.1
GCP Cloud Datastore	0.002907478	0.333333333	0.2	0.041666667	0.1

Рисунок 1.1 – Спрощення за принципом Парето.

За принципом Парето бачимо, що MongoDB Atlas за всіма критеріями краща або така само, тож можемо викреслити AWS Dynamo DB.

Отримаємо: $ОП_A$ = {Azure Cosmos DB, Mongo DB Atlas, GCP Bigtable, GCP Cloud Datastore}.

Для того щоб побачити, які бази даних найбільш задовольняють нашим вимогам, необхідно розрахувати лінійну адитивну згортку з ваговими коефіцієнтами (див. табл. 4).

Таблиця 4 – Лінійна адитивна згортка з ваговими коефіцієнтами

Лінійна адитивна згортка з ваговими коефіцієнтами						
	Популярність серед розробників	Вартість $/1 млн запитів на читання даних	Кількість API та інших методів доступу	Кількість мов програмування які підтримує	Кількість концепцій узгодженості які підтримуються	Z*
Azure Cosmos DB	0.070942453	0.4	1	1	1	0.4626887
Mongo DB Atlas	1	1	0	0.958333333	0.1	0.2302261
GCP Bigtable	0	0.153846	0.4	0	0.1	0.1602978
GCP Cloud Datastore	0.002907478	0.333333	0.2	0.041666667	0.1	0.0945753
Вагові коеф	0.2	0.1	0.6	0.05	0.05	

Для початку потрібно розставити вагові коефіцієнти згідно з метою дослідження. Основним критерієм, який планується досліджувати у роботі, є порівняння програмних реалізацій декількох API або методів доступу до бази даних, тому цей стовпчик (Кількість API та інших методів доступу) отримую

найвищий коефіцієнт 0.6. Важливо досліджувати ту базу даних, якою користуються більша кількість людей, так це дослідження буде мати більший сенс та принесе користі для більшої кількості розробників, тому цей критерій (Популярність серед розробників) отримує коефіцієнт 0.2. На третьому місці по важливості буде (Вартість $/1 млн запитів на читання даних) з коефіцієнтом 0.1. Для тестування доведеться розробити базу даних у хмарі, чим дешевше це буде коштувати тим краще. Дві категорії які залишились (Кількість мов програмування які підтримує, Кількість концепцій узгодженості які підтримуються) мають найменший вплив на дослідження, тому отримують однакові коефіцієнти 0.05.

Проведемо розрахунки за формулою:

$$Z* = max \sum_{j=1}^{n} \alpha_j \beta_j a_j ,$$

де α_j – нормуючі множники, β_j– вагові коефіцієнти, що відображають відносний внесок окремих критеріїв до загального критерію.

Зробивши розрахунки за лінійною адитивною згорткою з ваговими коефіцієнтами, ми бачимо, що Azure Cosmos DB має перевагу по критерію з найвагомішим коефіцієнтом і має найбільше значення (0.4626887), саме тому є кращим вибором для дослідження Azure Cosmos DB API.

1.2 Постановка задачі

Після того як ми провели аналіз проблемної області дослідження та обрали Cosmos DB для дослідження, можна сформувати список задач для проведення дослідження. Потрібно обрати програмні інтерфейси, які будуть досліджуватися, слід враховувати, що дослідження буде проводитися на платформі .NET і обирати програмні інтерфейси, які мають можливість реалізації на C#.

Для достовірності дослідження слід обрати предметну область, в якій часто використовуються бази даних, щоб досліджувана модель була максимально

наближена до реальних задач. Описати предметну область, розробити схему бази даних.

Після цього необхідно розробити план експериментального дослідження, слід враховувати, що проводити дослідження різних програмних інтерфейсів необхідно на однаковій схемі бази даних для достовірності дослідження.

Для досліду необхідно буде створити декілька баз даних Cosmos DB у Azure з різними API, провести їх налаштування та підключитися до них через розроблений програмний код.

Отже, для проведення дослідження необхідно вирішити наступні задачі:

- провести аналіз та обрати підходящі для дослідження програмні інтерфейси Cosmos DB;
- розробити план експериментального дослідження обраних методів (обрати предметну область, критерії порівняння програмних інтерфейсів, обмеження тощо);
- спроектувати БД для обраної предметної області та запити для дослідження;
- створити бази даних у хмарі, спроектувати та розробити програмне забезпечення для проведення дослідження;
- провести експериментальне дослідження обраних API та розробити рекомендації щодо використання того чи іншого програмного інтерфейсу.

2. ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Аналіз прикладів використання Cosmos DB

Azure Cosmos DB широко використовується у власних платформах електронної комерції Microsoft, на яких працюють Windows Store і XBox Live. Він також використовується в галузі роздрібної торгівлі для зберігання даних каталогу та пошуку подій у конвеєрах обробки замовлень [12].

Сценарії використання даних каталогу включають зберігання та запит набору атрибутів для таких сутностей, як люди, місця та продукти. Деякими прикладами даних каталогу є облікові записи користувачів, каталоги продуктів, реєстри пристроїв Інтернету речей і системи опису матеріалів. Атрибути для цих даних можуть відрізнятися та змінюватися з часом відповідно до вимог програми.

Розглянемо приклад каталогу продукції для постачальника автомобільних запчастин. Кожна частина може мати власні атрибути на додаток до загальних атрибутів, які мають усі частини. Крім того, атрибути для певної частини можуть змінитися наступного року, коли буде випущена нова модель. Azure Cosmos DB підтримує гнучкі схеми та ієрархічні дані, тому добре підходить для зберігання даних каталогу продуктів (див. рис. 2.1).

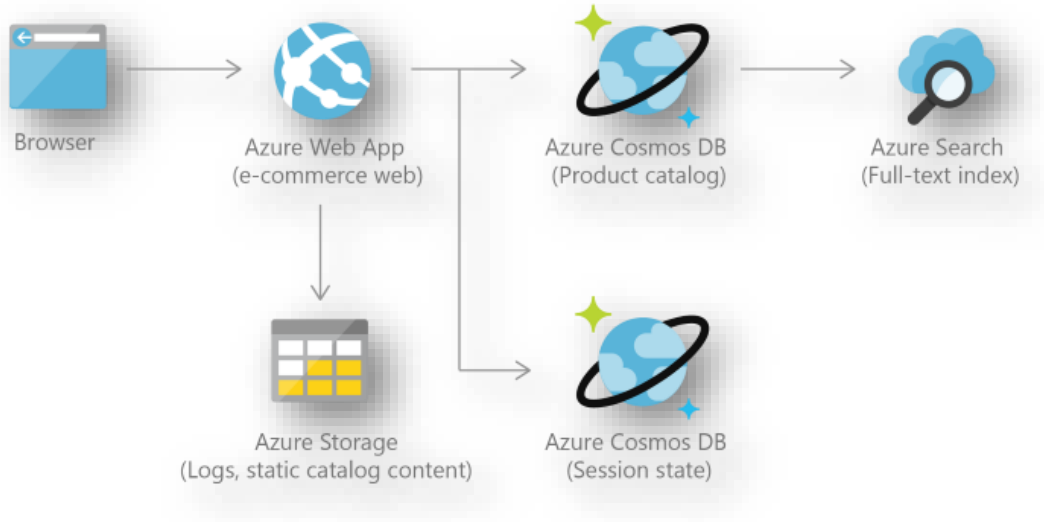

Рисунок 2.1 – Приклад використання Cosmos DB для системи електронної комерції

Azure Cosmos DB часто використовується для джерела подій для забезпечення керованих подіями архітектур за допомогою функції каналу змін. Канал змін надає подальшим мікросервісам можливість надійно та поступово зчитувати вставки та оновлення (наприклад, події замовлення), внесені до бази даних Azure Cosmos [13]. Цю функціональність можна використовувати для забезпечення постійного сховища подій як брокера повідомлень для подій, що змінюють стан, і керувати процесом обробки порядку між багатьма мікросервісами (які можна реалізувати як безсерверні функції Azure) (див. рис. 2.2).

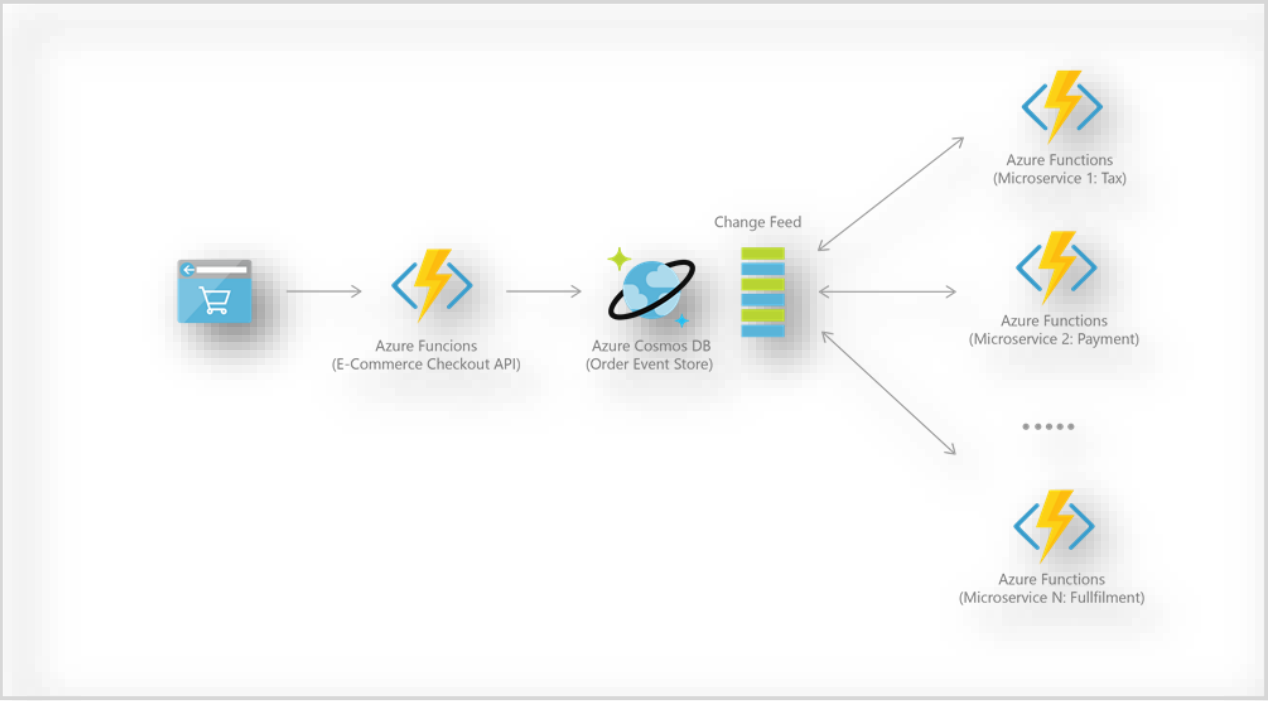

Рисунок 2.2 – Приклад використання Cosmos DB як брокера повідомлень в системі з мікросервісною архітектурою.

Для проведення дослідження та експериментів було вирішено в якості предметної області обрати електронну комерцію. Цей сценарій використання дозволить аналізувати отримані результати на максимально наближеній моделі даних до популярного сценарію використання Cosmos DB.

2.2 Аналіз та вибір методів програмної реалізації COSMOS DB API

Azure Cosmos DB пропонує кілька API бази даних, зокрема NoSQL, MongoDB, PostgreSQL Cassandra, Gremlin і Table (див. рис. 2.3). Використовуючи ці API, ви можете моделювати дані, використовуючи моделі документів, ключ-значення, графіки та сімейство стовпців. Ці API дозволяють вашим програмам обробляти Azure Cosmos DB так, ніби це різні інші технології баз даних, без накладних витрат на керування та підходи до масштабування. Azure Cosmos DB допомагає вам використовувати екосистеми, інструменти та навички, які ви вже маєте, для моделювання даних і запитів за допомогою різноманітних API [14].

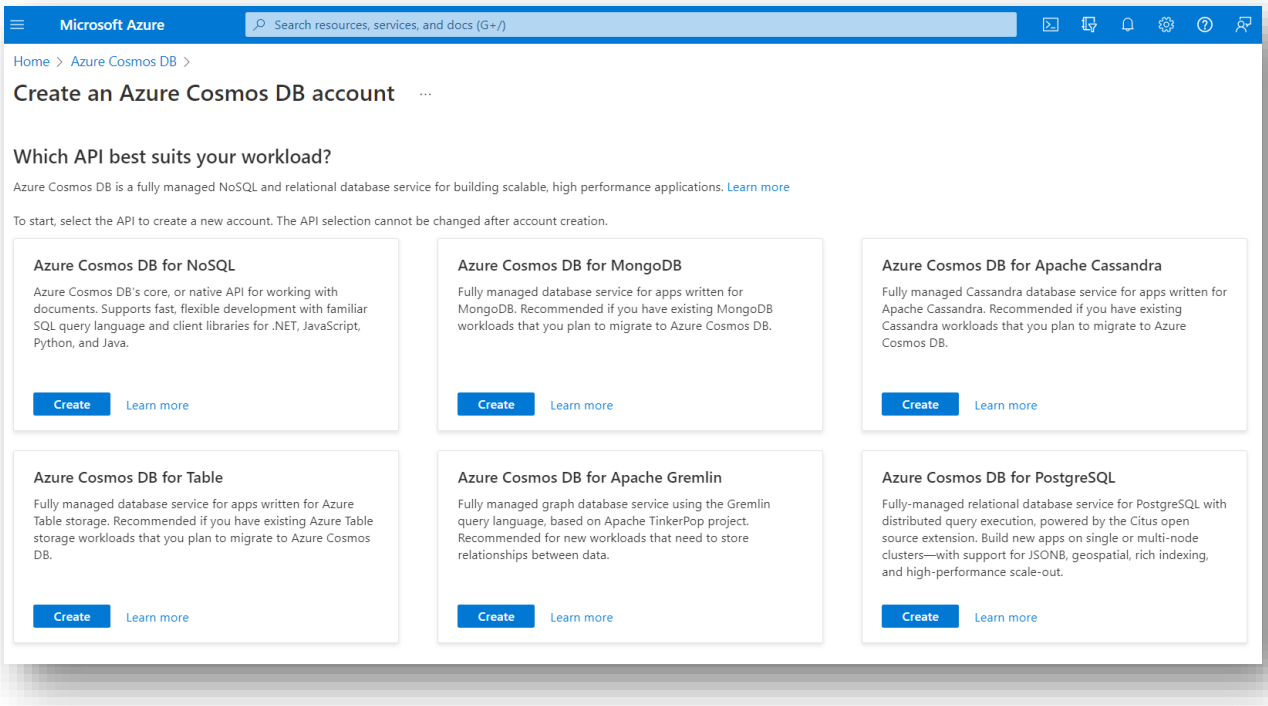

Рисунок 2.3 – Microsoft Azure сторінка створення Cosmos DB

API Azure Cosmos DB для NoSQL зберігає дані у форматі документа. Він пропонує найкращий наскрізний досвід, оскільки ми маємо повний контроль над інтерфейсом, сервісом і клієнтськими бібліотеками SDK. Будь-яка нова функція, яка розгортається в Azure Cosmos DB, спочатку доступна в API для облікових

записів NoSQL. Облікові записи NoSQL забезпечують підтримку для запиту елементів за допомогою синтаксису мови структурованих запитів (SQL), однієї з найвідоміших і найпопулярніших мов запитів для запитів об’єктів JSON.

Якщо ви переходите з інших баз даних, таких як Oracle, DynamoDB, HBase тощо, і якщо ви хочете використовувати модернізовані технології для створення своїх програм, API для NoSQL є рекомендованим варіантом. API для NoSQL підтримує аналітику та забезпечує ізоляцію продуктивності між операційними та аналітичними навантаженнями.

API Azure Cosmos DB для MongoDB зберігає дані у структурі документа через формат BSON. Він сумісний із дротовим протоколом MongoDB; однак він не використовує жодного рідного коду, пов’язаного з MongoDB. API для MongoDB — чудовий вибір, якщо ви хочете використовувати ширшу екосистему MongoDB і навички без шкоди для використання функцій Azure Cosmos DB.

Функції, надані Azure Cosmos DB, включають:

– масштабування;
– висока доступність;
– геореплікація;
– кілька місць для запису;
– автоматичне та прозоре керування сегментами;
– прозора реплікація між операційними та аналітичними сховищами.

Ви можете використовувати існуючі програми MongoDB з API для MongoDB, просто змінивши рядок підключення. Ви можете перемістити будь-які наявні дані за допомогою власних інструментів MongoDB, таких як mongodump і mongorestore, або за допомогою інструменту міграції бази даних Azure. Інструменти, такі як оболонка MongoDB, MongoDB Compass і Robo3T, можуть виконувати запити та працювати з даними так само, як і з нативною MongoDB.

Azure Cosmos DB для PostgreSQL — це керована служба для запуску PostgreSQL у будь-якому масштабі з суперпотужністю розподілених таблиць із відкритим кодом Citus. Він зберігає дані або на одному вузлі, або в конфігурації з кількома вузлами.

Azure Cosmos DB для PostgreSQL побудовано на основі власного PostgreSQL, а не на форку PostgreSQL, і дозволяє вибрати будь-яку основну версію бази даних, яку підтримує спільнота PostgreSQL. Він ідеально підходить для запуску одновузлової бази даних із багатим індексуванням, геопросторовими можливостями та підтримкою JSONB. Пізніше, якщо знадобиться більша продуктивність, ви можете додати вузли до кластера без простою.

API Azure Cosmos DB для Cassandra зберігає дані в схемі, орієнтованій на стовпці. Apache Cassandra пропонує високорозподілений, горизонтально масштабований підхід до зберігання великих обсягів даних, одночасно пропонуючи гнучкий підхід до схеми, орієнтованої на стовпці. API для Cassandra в Azure Cosmos DB відповідає цій філософії наближення до розподілених баз даних NoSQL. Цей API для Cassandra є дротовим протоколом, сумісним із рідним Apache Cassandra. Варто розглянути API для Cassandra, якщо ви хочете отримати переваги від гнучкості та повністю керованого характеру Azure Cosmos DB і при цьому використовувати більшість власних функцій, інструментів та екосистеми Apache Cassandra. Ця повністю керована природа означає, що в API для Cassandra вам не потрібно керувати ОС, Java VM, збирачем сміття, продуктивністю читання/запису, вузлами, кластерами тощо.

Ви можете використовувати клієнтські драйвери Apache Cassandra для підключення до API для Cassandra. API для Cassandra дає змогу взаємодіяти з даними за допомогою мови запитів Cassandra (CQL) і таких інструментів, як оболонка CQL, клієнтські драйвери Cassandra, з якими ви вже знайомі. API для Cassandra наразі підтримує лише сценарії OLTP. Використовуючи API для Cassandra, ви також можете використовувати унікальні функції Azure Cosmos DB, такі як канал змін.

API Azure Cosmos DB для Gremlin дозволяє користувачам створювати запити на графіки та зберігає дані як ребра та вершини.

Використовуйте API для Gremlin для сценаріїв:

– залучення динамічних даних;

– залучення даних зі складними зв’язками;

– залучення даних, надто складних для моделювання за допомогою реляційних баз даних;
– якщо ви хочете використовувати наявну екосистему та навички Gremlin.

API для Gremlin поєднує в собі потужність алгоритмів графічної бази даних із високомасштабованою керованою інфраструктурою. Цей API надає унікальне та гнучке рішення типових проблем із даними, пов'язаних із відсутністю гнучкості чи реляційних підходів. API для Gremlin наразі підтримує лише сценарії OLTP.

API для Gremlin базується на структурі графових обчислень Apache TinkerPop. API для Gremlin використовує ту саму мову запитів Graph для прийому та запиту даних. Він використовує стратегію розділу Azure Cosmos DB для виконання операцій читання/запису з механізму бази даних Graph. API для Gremlin підтримує протокол зв'язку з Gremlin з відкритим вихідним кодом, тому ви можете використовувати SDK Gremlin з відкритим кодом для створення своєї програми. API для Gremlin також працює з Apache Spark і GraphFrames для сценаріїв складних аналітичних графіків.

API Azure Cosmos DB for Table зберігає дані у форматі ключ/значення. Якщо ви використовуєте сховище Azure Table, ви можете помітити певні обмеження щодо затримки, масштабування, пропускної здатності, глобального розподілу, керування індексами, низької продуктивності запитів. API для таблиць долає ці обмеження, тому рекомендується перенести свою програму, якщо ви хочете скористатися перевагами Azure Cosmos DB. API для таблиць підтримує лише сценарії OLTP.

Програми, написані для сховища таблиць Azure, можна перенести на API для таблиць з невеликими змінами коду та скористатися перевагами преміум-можливостей.

Після проведення опису та порівняння існуючих Cosmos DB API нам необхідно обрати ті програмні інтерфейси, які найбільш підходять для нашого дослідження (див. табл. 5). Для початку сформуємо множину альтернатив:

– API for NoSQL;

– API for MongoDB;
– API for PostgreSQL;
– API for Apache Cassandra;
– API for Apache Gremlin;
– API for Table.

Таблиця 5 – Моделювання задачі прийняття рішень щодо вибору Cosmos DB API для проведення дослідження

	Підтримка платформи .NET	Частоста оновлення API	Доступність функцій БД через код	Формат зберігання даних
API for NoSQL	Так	Найновіші функції	Доступні всі функції	Документ
API for MongoDB	Так	Часто	Обмеження від залежного API	Документ
API for PostgreSQL	Так	Часто	Обмеження від залежного API	Вузли
API for Apache Cassandra	Так	Часто	Обмеження від залежного API	Схема орієнтована на стовпці
API for Apache Gremlin	Так	Часто	Обмеження від залежного API	Як ребра та вершини
API for Table	Так	Рідко	Доступна більшість функцій	Документ

Тепер необхідно сформувати множину критеріїв вибору API:

– підтримка платформи .NET — мета дослідження це порівняти програмні реалізації різних API саме на платформі .NET, тому важливо, щоб досліджуваний програмний інтерфейс мав підтримку даної платформи;
– частота оновлення API — цей критерій показує наскільки швидко нові функції Cosmos DB будуть доступні на тому чи іншому API;
– доступність функцій БД через код — цей критерій показує можливість створювати базу даних та контейнери з коду, перевіряти чи існує контейнер та інші налаштування;

– формат зберігання даних — за цим критерієм буде порівняно в якому форматі зберігаються дані.

Для того, щоб було простіше порівнювати обрані критерії, необхідно сформулювати шкали оцінок за цими критеріями.

– підтримка платформи .NET:
    1) так – 1;
    2) ні – 0.
– частоста оновлення арі:
    1) рідко – 1;
    2) часто – 2;
    3) найновіщі функції – 3.
– доступність функцій БД через код:
    1) доступні всі функції – 3;
    2) доступна більшість функцій – 2;
    3) обмеження від залежного арі – 1.
– формат зберігання даних:
    1) документ – 4;
    2) вузли – 3;
    3) схема орієнтована на стовпці – 2;
    4) як ребра та вершини – 1.

Після формулювання шкал оцінок, оновимо дані в таблиці (див. табл. 6).

Бачимо, що всі програмні інтерфейси підтримують платформу .NET тому цей стовпець ми можемо відкинути. Застосуємо формулу minmax щоб привести всі коефіцієнти до значень від 0 до 1, які буде легко порівнювати (див. табл. 7).

Таблиця 6 – Таблиця нь щодо вибору Cosmos DB API для проведення дослідження після формулювання шкал оцінок критеріїв

	Підтримка платформи .NET	Частоста оновлення API	Доступність функцій БД через код	Формат зберігання даних
API for NoSQL	1	3	3	4
API for MongoDB	1	2	1	4
API for PostgreSQL	1	2	1	3
API for Apache Cassandra	1	2	1	2
API for Apache Gremlin	1	2	1	1
API for Table	1	1	2	4

Таблиця 7 – Таблиця після застосування правила minmax

Нормування по minmax та еталону			
	Частоста оновлення API	Доступність функцій БД через код	Формат зберігання даних
API for NoSQL	1	1	1
API for MongoDB	0.666	0.333	1
API for PostgreSQL	0.666	0.333	0.75
API for Apache Cassandra	0.666	0.333	0.5
API for Apache Gremlin	0.666	0.333	0.25
API for Table	0.333	0.666	1

Розрахуємо лінійну адитивну згортку з ваговими коефіцієнтами, щоб наглядно побачити які програмні інтерфейси найбільше задовольняють нашим критеріям (див. табл. 8). Для початку потрібно розставити вагові коефіцієнти згідно нашими пріоритетами дослідження. Найбільш впливовим критерієм буде доступність функцій бази даних через код з коефіцієнтом 0.6, це самий важливий параметр, бо він визначає яку кількість можливостей того чи іншого програмного інтерфейсу можна реалізувати саме через код. Наступний важливий критерій це частота оновлення API з коефіцієнтом 0.2, важливо використовувати нові функції обраної бази даних, тому те API в якому швидше доступні нові функції платформи буде оцінюватися більше. Останній критерій формат зберігання даних має коефіцієнт 0.2.

Таблиця 8 – Результати використання лінійної адитивної згортки

Лінійна адитивна згортка з ваговими коефіцієнтами				
	Частоста оновлення API	Доступність функцій БД через код	Формат зберігання даних	Z*
API for NoSQL	1	1	1	0.36901789
API for MongoDB	0.666	0.333	1	0.179174099
API for PostgreSQL	0.666	0.333	0.75	0.164888385
API for Apache Cassandra	0.666	0.333	0.5	0.15060267
API for Apache Gremlin	0.666	0.333	0.25	0.136316956
API for Table	0.333	0.666	1	0.246674773
Вагові коефіцієнти	0.2	0.6	0.2	

З таблиці 8 ми бачимо, що API for NoSQL має найвищий показник 0.36901789, це значить що цей програмний інтерфейс найбільше нам підходить. Також врахуємо те, що за цим інтерфейсом ми можемо різними програмними способами реалізувати роботу з базою даних, а саме це Entity Framework Core

Cosmos та CosmosClient. На другому місці ми бачимо API for Table, але цей інтерфейс специфічний, бо він більше всього підходить не для вирішення загальних задач, а для, наприклад переносу архітектури у Cosmos DB з Azure Table, тому ми його не будемо досліджувати. І друге API, яке ми будемо досліджувати це API for MongoDB з коефіцієнтом 0.179174099. MongoDB — це база даних документів, яка використовується для створення високодоступних і масштабованих інтернет-додатків. Завдяки гнучкому підходу до схеми він популярний серед команд розробників, які використовують гнучкі методології. API for MongoDB це гарна альтернатива для порівняння з API for NoSQL.

2.3 Аналіз та моделювання предметної області

Електронна комерція (e-commerce) - це процес купівлі та продажу товарів та послуг через інтернет. У цій предметній області можна назвати кілька ключових аспектів.

– технічні аспекти: один із головних факторів, що визначають успіх електронної комерції, – це технічна складова. Важливо мати добре розроблену та налагоджену платформу, яка забезпечує зручну навігацію, швидке завантаження сторінок, безпечну роботу з даними та можливість проведення онлайн-платежів;

– маркетингові аспекти: для успішної електронної комерції необхідно вміти просувати свій бізнес в онлайн-просторі. Ключові маркетингові інструменти в цій галузі – пошукова оптимізація (SEO), контекстна реклама, соціальні мережі, email-маркетинг та інші;

– юридичні аспекти: в електронній комерції необхідно дотримуватись законодавства, що регулює цю область. Це стосується таких питань як захист персональних даних, права споживачів, оподаткування, захист інтелектуальної власності та інші;

– логістичні аспекти: для ефективної електронної комерції важливо мати розроблену систему логістики, що дозволяє швидко та ефективно доставляти товари споживачам. Ключові питання у цій галузі – це складські процеси, управління запасами, логістика доставки, відстеження вантажів та інші;

– фінансові аспекти: для ефективної електронної комерції необхідно вміти управляти фінансами бізнесу. Ключові питання у цій галузі – це облік доходів та витрат, податкова звітність, фінансовий моніторинг, управління фінансовим потоком та інші.

Загалом електронна комерція — це складний і багатогранний процес, що потребує комплексного підходу та високої кваліфікації фахівців у різних галузях. Моделювання предметної області електронної комерції може включати такі моделі:

– модель бізнес-процесів: ця модель описує всі процеси, що відбуваються в електронній комерції – від розміщення товару на сайті до оплати та доставки. Така модель може бути корисною для оптимізації бізнес-процесів та поліпшення ефективності роботи;

– модель бази даних: ця модель відображає структуру бази даних, яка використовується для зберігання інформації про товари, замовлення, споживачів та інших даних. Така модель може допомогти розробити оптимальну структуру бази даних та покращити швидкість доступу до даних.

– модель взаємодії з користувачем: ця модель описує, як користувач взаємодіє із сайтом електронної комерції, зокрема, як він робить замовлення, реєструється на сайті, шукає товари та інші дії. Така модель може допомогти зрозуміти поведінку користувачів на сайті та вдосконалити його інтерфейс;

– модель безпеки: ця модель відображає всі аспекти безпеки в електронній комерції - захист даних користувачів, захист від шахрайства та

інших злочинів, захист від зловмисників. Така модель допоможе розробити ефективну систему безпеки для сайту електронної комерції;

– модель логістики: Ця модель описує процес доставки товарів, включаючи керування запасами, розміщення замовлень, відправку вантажів та інші аспекти. Така модель може допомогти покращити управління логістикою та доставкою товарів, скоротити час доставки та збільшити задоволеність клієнтів.

Для проведення дослідження нам не потрібно буде розробляти діаграму варіантів використання, діаграму послідовності, діаграму компонентів чи інтерфейс сайту бо ми орієнтуємося саме на роботу коду з базою даних.

Моделювання бази даних для електронної комерції є важливим етапом в розробці будь-якого інтернет-магазину або іншої платформи електронної комерції. Для початку треба розробити концептуальну модель ключових сутностей предметної області. В базі даних будуть зберігатися основні сутності: Товар, Замовлення, Товар в замовленні та Платіж у відповідних концептах: Product, Order, OrderItem та Payment. Розробимо загальну діаграму класів предметної області (див.рис. 2.4).

2.4 Проектування бази даних

Перейдемо до розробки ER діаграми бази даних, для цього нам потрібно описати кожну сутність та властивості, які вона має. ER-діаграма (Entity-Relationship diagram) - це графічний інструмент моделювання даних, який дозволяє показати структуру бази даних та відношення між різними її сутностями. ER-діаграми є важливим інструментом проектування баз даних, оскільки вони допомагають розуміти структуру даних та відношення між ними перед створенням самої бази даних.

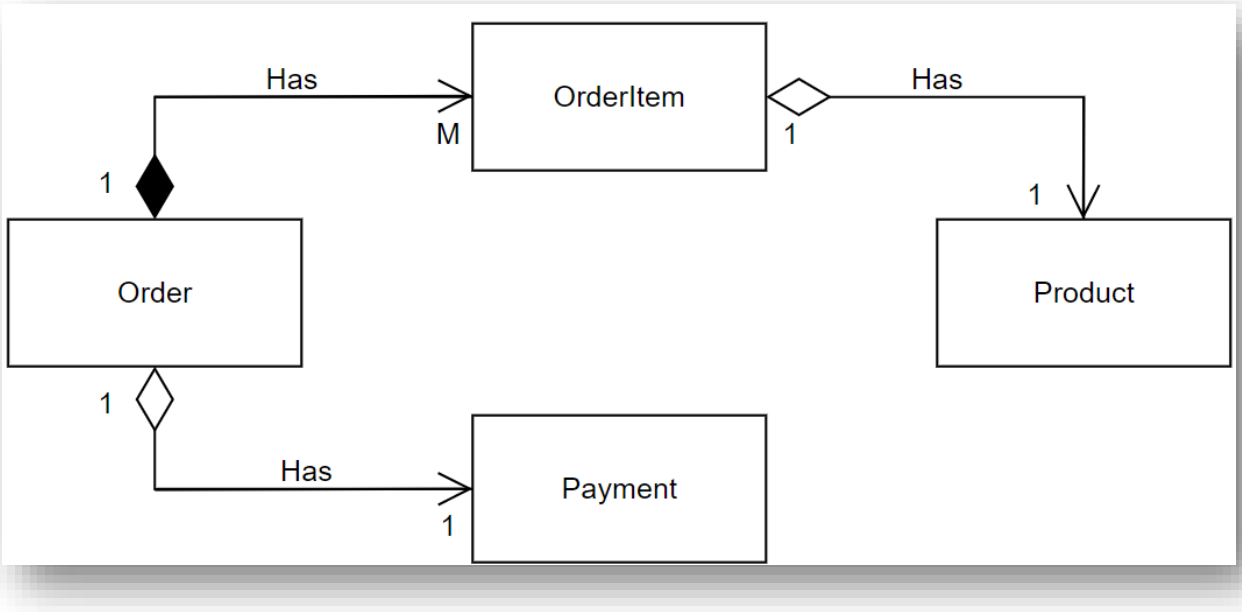

Рисунок 2.4 – Загальна діаграма класів предметної області

Product має наступні поля: id, name, manufacturer, description. У цій сутності будуть зберігатися дані про усі товари, кожен товар має свою назву, свого виробника та опис товару, а також унікальний ідентифікатор id товару.

Order має наступні поля: id, orderDate, orderPrice. У цій сутності будуть зберігатися дані про замовлення, кожне замовлення має унікальний ідентифікатор id, дату замовлення та вартість замовлення.

OrderItem має наступні поля: price, quantity. У цій сутності будуть зберігатися дані про позиції кожного замовлення, кількість товару та ціну.

Payment має наступні поля: paymentDate, price. У цій сутності будуть зберігатися дані про оплату замовлення, ця сутності має поле для зберігання вартості замовлення та дати оплати.

Після визначення властивостей основних сутностей була розроблена ER діаграма бази даних (див. рис. 2.5).

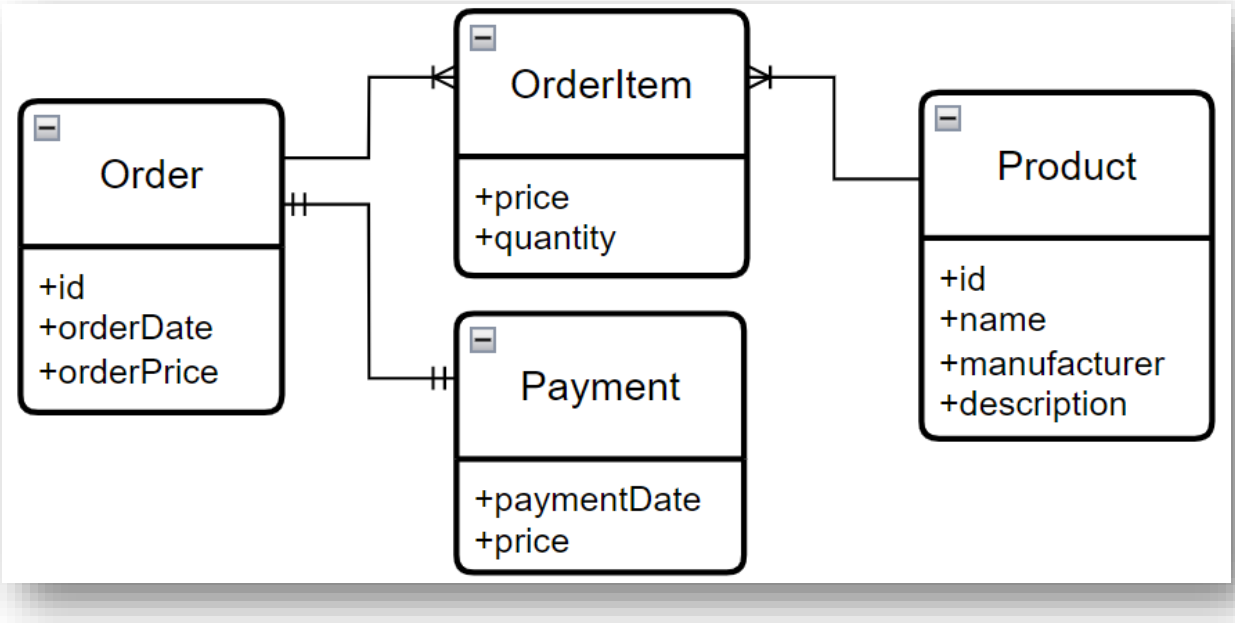

Рисунок 2.5 – ER діаграма бази даних

Після розробки ER діаграми ми можемо її використати для опису логічної структури даних предметної області у MongoDB (див. рис. 2.6).

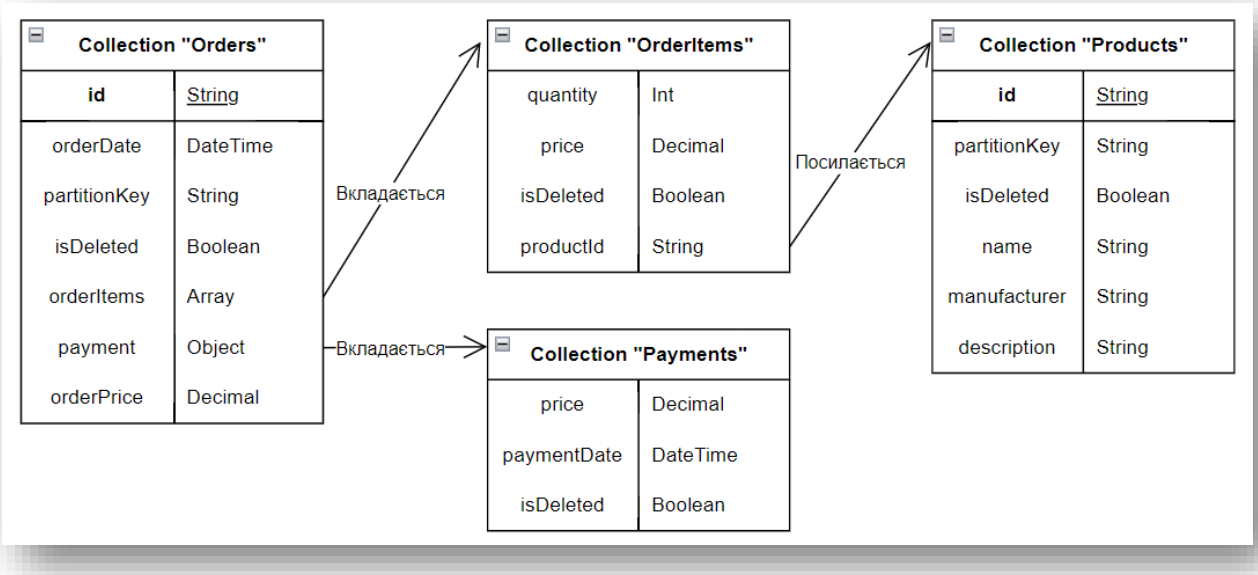

Рисунок 2.6 – Логічна структура даних для MongoDB

У MongoDB дані зберігаються у колекціях (collections), які містять документи (documents). Документи - це об'єкти у форматі BSON, які містять

ключі та значення. Ключі відповідають полям документа, а значення - їх значенням. BSON (Binary JSON) – це бінарний формат серіалізації JSON, який забезпечує більш компактне зберігання даних та дозволяє більш ефективно використовувати ресурси.

Отже для моделювання логічної структури даних в MongoDB були використані поля з ER-діаграми та додаткові поля необхідні для програмної реалізації роботи з CosmosDB колекціями: partitionKey та isDeleted. PartitionKey необхідний CosmosDB для оптимізації запитів, а властивість isDeleted для реалізації м’якого видалення.

2.5 Планування експериментальної частини дослідження

В ході порівняння та вибору Cosmos DB API для дослідження було обрано API for NoSQL та API for MongoDB. На платформі .NET програмно реалізувати API for NoSQL можна двома способами: використовуючи Entity Framework Core Cosmos та використовуючи Cosmos Client системний клас, тому ці дві реалізації будуть досліджуватися і порівнюватися як окремі.

Для будь-якої бази даних важливими параметрами є швидкість виконання CRUD операцій, які використовуються найчастіше. Отже, під час дослідження було розглянуто наступні запити до бази даних:

– «INSERT-запит» - запит на створення нового замовлення: «INSERT INTO Orders ("id", "isDeleted", "orderDate", "orderPrice", "partitionKey", "orderItems", "payment") VALUES ("b0842587-42e7-4b5f-ab17-c09ca1c9629f", "false", "2023-04-27T20:16:34.4468614Z", "100", "100", [{ "price": 50, "productId": "d489b6b9-5157-408f-a249-fcfc33493c08", "quantity": 1 }, { "price": 25, "productId": "f88a4167-b622-4b7c-9770-a39d4aa2de85", "quantity": 2 }], { "paymentDate": "2023-04-27T20:16:12.0488881Z", "price":

100 });». У цьому INSERT запиті пропонується дослідити швидкість створення нового замовлення.

– «SELECT-запит №1» - запит на отримання замовлення за певним partitionKey: «SELECT * FROM Orders WHERE Orders.partitionKey = "100"». У цьому SELECT запиті пропонується дослідити швидкість повернення усіх замовлень, що мають спільний ключ розділу Cosmos DB.
– «SELECT-запит № 2» - запит на отримання замовлення за певним id: «SELECT * FROM Orders WHERE Orders.id = "b0842587-42e7-4b5f-ab17-c09ca1c9629f"». У цьому SELECT запити пропонується дослідити швидкість отримання замовлення по id без оптимізації partition key. Обгрунтуйте тут, навіщо 2 майже однакових запити, що Ви хотіли побачити тут особливого!
– «UPDATE-запит» - апит на оновлення полів сутності замовлення: «UPDATE Orders SET "orderDate" = "2023-04-28T11:15:49.9163777Z" WHERE Orders.id == "b0842587-42e7-4b5f-ab17-c09ca1c9629f";». У цьому UPDATE запиті пропонується дослідити швидкість оновлення сутності замовлення.
– «DELETE-запит» - запит на видалення сутності замовлення з бази даних: «DELETE FROM Orders WHERE Orders.id == "b0842587-42e7-4b5f-ab17-c09ca1c9629f";». У цьому DELETE запити пропонується дослідити швидкість видалення запису з БД.
– «SOFT DELETE-запит» - запит на безпечне видалення замовлення з бази даних: «UPDATE Orders SET "isDeleted" = "true" WHERE Orders.id == "b0842587-42e7-4b5f-ab17-c09ca1c9629f";». У цьому SOFT DELETE запиті пропонується дослідити швидкість м’якого видалення запису у БД та порівняти її з DEELTE запитом.

Метрикою порівняння цих операцій буде швидкість виконання у мілісекундах, кількість витрачених сервером на це байтів та кількість витрачених ресурсів RU на виконання операції в базі даних. Database throughput RU (Request

Units) – це одиниця виміру продуктивності, що використовується в Azure Cosmos DB. Кожна дія, що виконується у Cosmos DB, потребує певної кількості RU.

Окремо слід дослідити чи можливо використовуючи той чи інший програмний підхід створювати базу даних або контейнер, перевіряти чи база даних або контейнер вже створені. Метрикою для цього порівняння буде флаг TRUE чи FALSE, можливо чи ні створити з коду базу даних чи контейнер.

Також слід дослідити складність реалізації для кожного з підходів, проаналізувати наскільки багато коду потрібно написати для його роботи та зробити висновки для вирішення яких поставлених задач, який підхід краще використовувати та чому. Метрикою для цього порівняння буде кількість рядків коду для кожної з реалізацій.

2.6 Розробка об’єктно-орієнтованої моделі для роботи з базою даних

Для розробки коду реалізації програмних інтерфейсів CosmosDB буде використовуватися підхід об’єктно-орієнтованого програмування. Для цього нам необхідно виділити ключові об’єкти предметної області у класи та описати які дані можуть зберігати ці об’єкти та яку поведінку вони мають. Тепер ми повинні використати розроблену логічну модель БД електронної комерції. Це товар, замовлення, товар в замовленні та оплата.

Основним об’єктом нашої логічної моделі є товар (class Product), який має назву (string Name), має виробника (string Manufacturer) та опис (string Description).

Для відображення замовлення будемо використовувати клас замовлення (class Order), який має поле дата замовлення (DateTime OrderDate), список товарів доданих до замовлення (List<OrderItem> OrderItems), загальну вартість замовлення (decimal OrderPrice) та дані про платіж (Payment Payment).

Додатковим класом для відображення кожної позиції в замовленні буде (class OrderItem), який має поле для зберігання посилання на товар (string

ProductId), поле для вартості товару (decimal Price), а також поле для зберігання кількості товарів у замовленні (int Quantity).

Кожне замовлення повинно бути оплаченим, для цього нам необхідно додати клас оплата (class Payment), який має поле для зберігання дати платежу (DateTime PaymentDate) та поле для зберігання загальної ціни замовлення (decimal Price).

Після визначення базових об’єктів обраної предметної області була створена загальна діаграма базових класів електронної комерції (див. рис. 2.7).

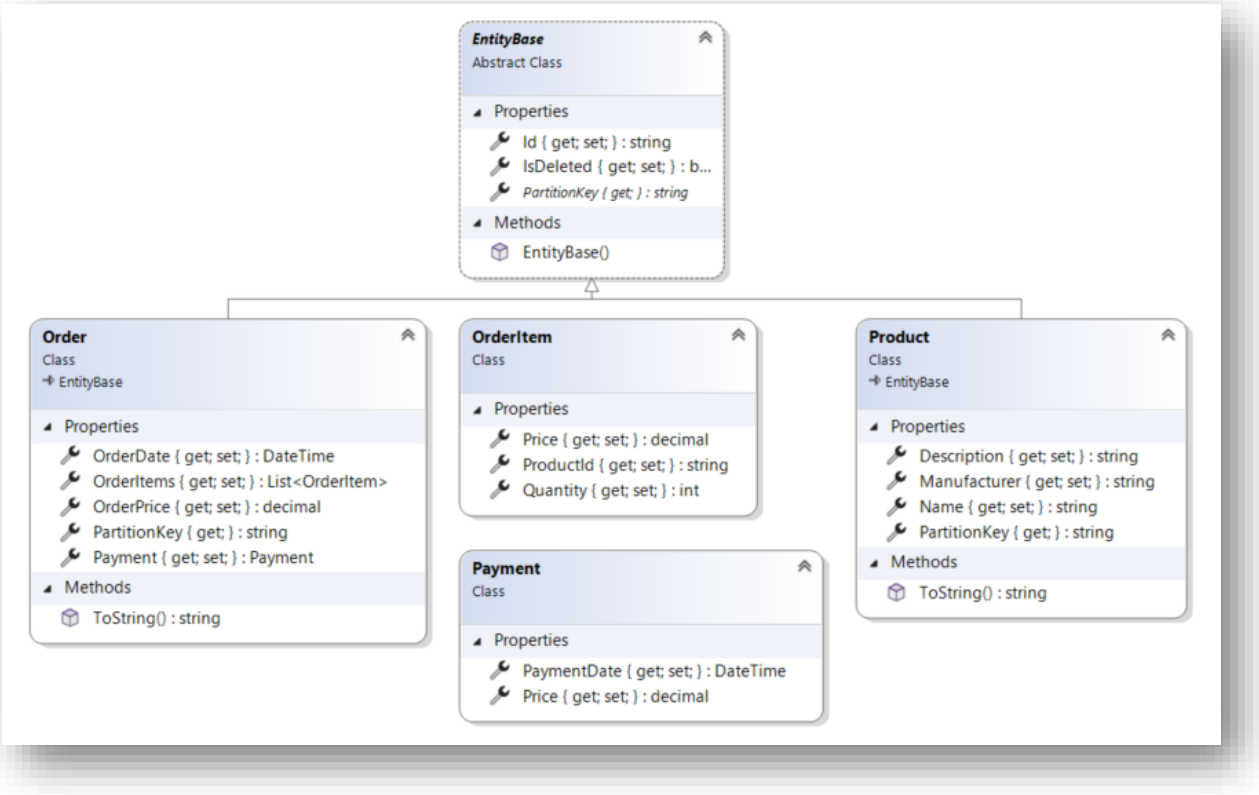

Рисунок 2.7 – Діаграма базових класів для обраної предметної області

Усі класи які будуть зберігатися в базі даних в окремих колекціях, це колекції Orders та Products, повинні мати ряд схожих властивостей, які ми можемо винести в базовий абстрактний клас (class EntityBase), а саме це унікальний ключ кожного запису (string Id), булеве поле для відображення чи видалений запис (bool IsDeleted), це необхідно для реалізації підходу безпечного видалення (soft

delete), м'яке видалення може покращити продуктивність і може дозволити відновити видалені дані, а ключ розділу (string PartitionKey) необхідний для оптимізації запитів. З точки зору написання ефективних запитів, Cosmos DB дозволяє групувати набір елементів або даних у вашій колекції за подібною властивістю, визначеною ключем розділу. Ключі розділу є ключовим елементом для ефективного розподілу ваших даних у різних логічних і фізичних наборах, щоб запити, що виконуються до бази даних, завершувалися якомога швидше. Важливо вибрати ключ розділу на етапі проектування програм, оскільки ви не можете змінити ключ розділу після створення контейнера. Зробивши властивість PartitionKey абстрактною в базовому класі, ми зможемо перевизначити її по-різному в класах спадкоємців.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі буде розглянуто налаштування Azure для роботи з Cosmos DB та створення трьох програмних рішень, а саме, використовуючи Cosmos Client, використовуючи Entity Framework Core Cosmos для Azure Cosmos DB for NoSQL API та використовуючи Mongo Client для Azure Cosmos DB for MongoDB API.

3.1 Налаштування Azure для роботи з Cosmos DB

Для того, щоб провести дослідження нам потрібно створити три окремі проекти для реалізації кожного з обраних підходів. Для розробки буде використовуватися Azure Cloud та Visual Studio 2022. Було вирішено створювати програми з веб API, щоб було зручніше бачити результати виконання запитів, а саме використовуючи Swagger OpenAPI, а результати експериментів писати у вигляді логів в консоль.

Swagger OpenAPI - це інструментарій, що використовується для опису та документування веб-сервісів RESTful API. Він дозволяє описати структуру веб-сервісу, методи API та параметри, які вони приймають та повертають, а також формати даних, що використовуються при передачі інформації через API [15].

Було вирішено проводити розробку на мові C# та використовувати платформу .Net Core. Це дуже популярна технологія на сьогоднішній день з великим ком’юніті. .NET Core - це переносна, високопродуктивна та відкрита платформа для створення сучасних програм, яка може працювати на різних операційних системах, таких як Windows, Linux та macOS. Вона включає основні бібліотеки та інструменти для розробки додатків на платформі .NET.

C# .NET Core поєднує можливості мови програмування C# і платформи .NET Core. Він дозволяє створювати високопродуктивні програми з використанням сучасних технологій, таких як мікросервісна архітектура,

контейнеризація, хмарні сервіси і т.д. .NET Core має широкий спектр функціональності та бібліотек класів, які допомагають розробникам створювати масштабовані та надійні програми. Крім того, він має інтегроване середовище розробки Visual Studio, яке надає зручний інтерфейс для створення, налагодження та тестування програм на C#.

Було створено програмне рішення CosmosDbResearch, в якому пізніше створені три проекти на .NET 6.0 по одному для кожного з досліджуваних програмних інтерфейсів CosmosDB (див. рис. 3.1):

– CosmosDbResearch.Cc.Api для Cosmos Client;
– CosmosDbResearch.Ef.Api для Entity Framework Core Cosmos;
– CosmosDbResearch.Md.Api для MongoDB.

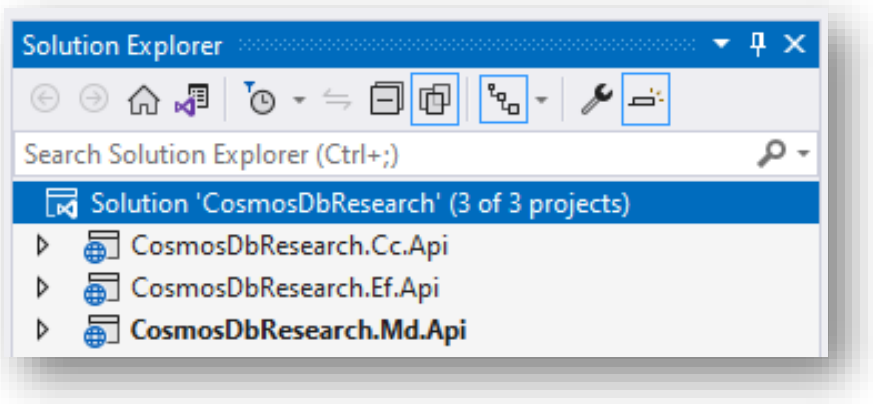

Рисунок 3.1 – Програмне рішення CosmosDbResearch

Для створення ресурсів пов’язаних з дослідженням CosmosDB API була створена окрема ресурсна група cosmos-db-research у Azure Cloud (див. рис. 3.2).

Azure Resource Group (група ресурсів Azure) – це логічний контейнер, який містить ресурси Azure, такі як віртуальні машини, бази даних, мережеві інтерфейси та інші ресурси. Група ресурсів надає зручний спосіб управління, організації та розгортання ресурсів Azure.

Кожен ресурс Azure має бути створений у межах певної групи ресурсів. Це полегшує управління ресурсами, оскільки всі ресурси, пов'язані з певним

додатком чи проектом, можуть бути розміщені в одній групі ресурсів. Крім того, група ресурсів дозволяє керувати доступом та дозволами на рівні групи, що полегшує керування безпекою.

У групі ресурсів Azure можуть бути налаштовані різні параметри, такі як розташування, теги, керування доступом, автоматичне масштабування та інші. Ці параметри можуть бути налаштовані для всіх ресурсів, що знаходяться в групі ресурсів, що полегшує їхнє керування [16].

Крім того, група ресурсів Azure може бути експортована до шаблону Azure Resource Manager, що дозволяє створювати документовані та повторно використовувані сценарії розгортання ресурсів в Azure.

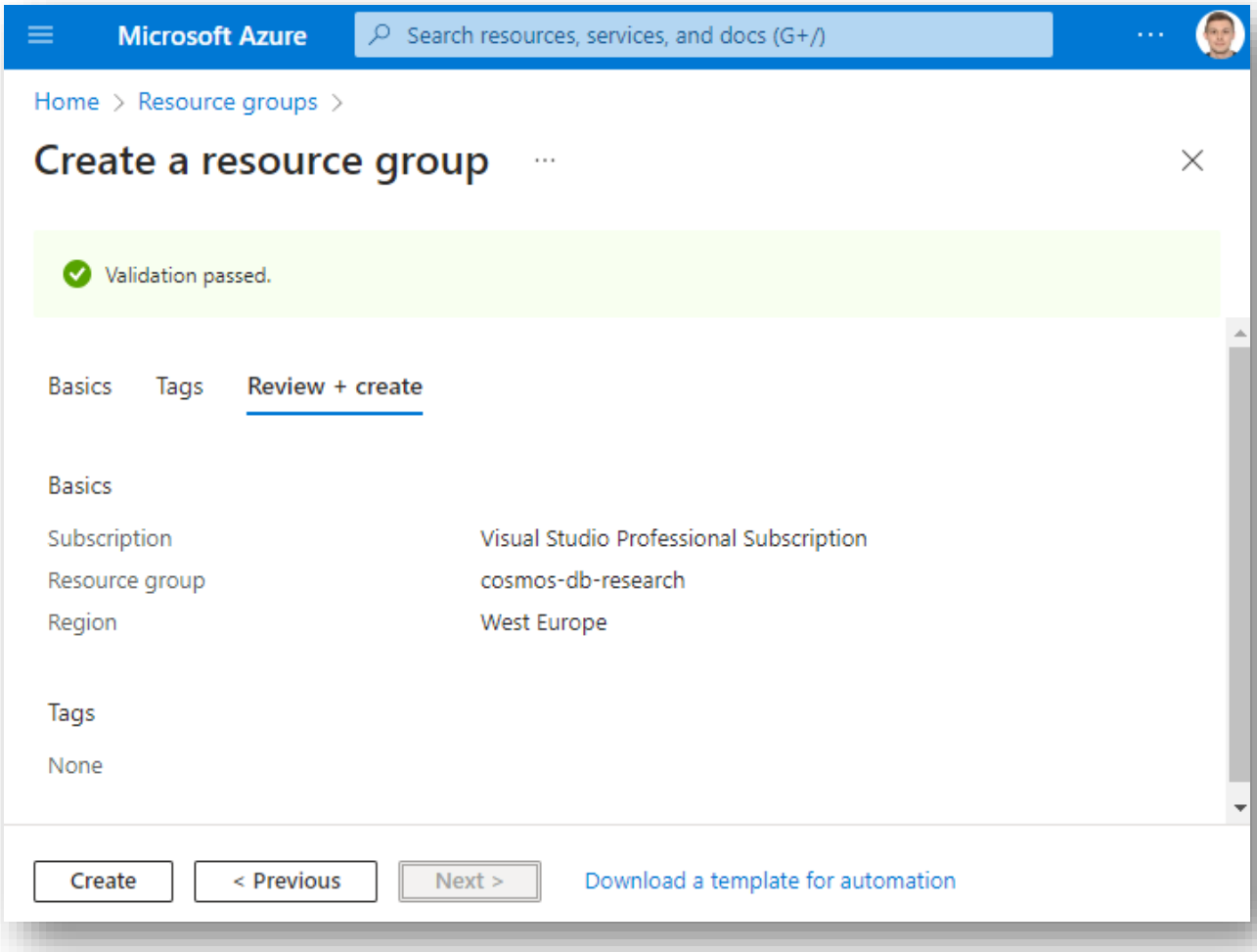

Рисунок 3.2 – Створення ресурсної групи cosmos-db-research

Для Cosmos Client та Entity Framework Core Cosmos підійде Cosmos Db for NoSQL API Account, який було створено у Azure (див. рис. 3.3). Назва акаунту cosmos-db-for-no-sql, також були встановлені налаштування резервного копіювання раз у 24 години та зберігання бекапів 2 дні, були обрані найнижчі параметри для зменшення вартості обслуговування створеної бази даних. Розташування бази даних було обрано найближчий сервер це (Europe) West Europe.

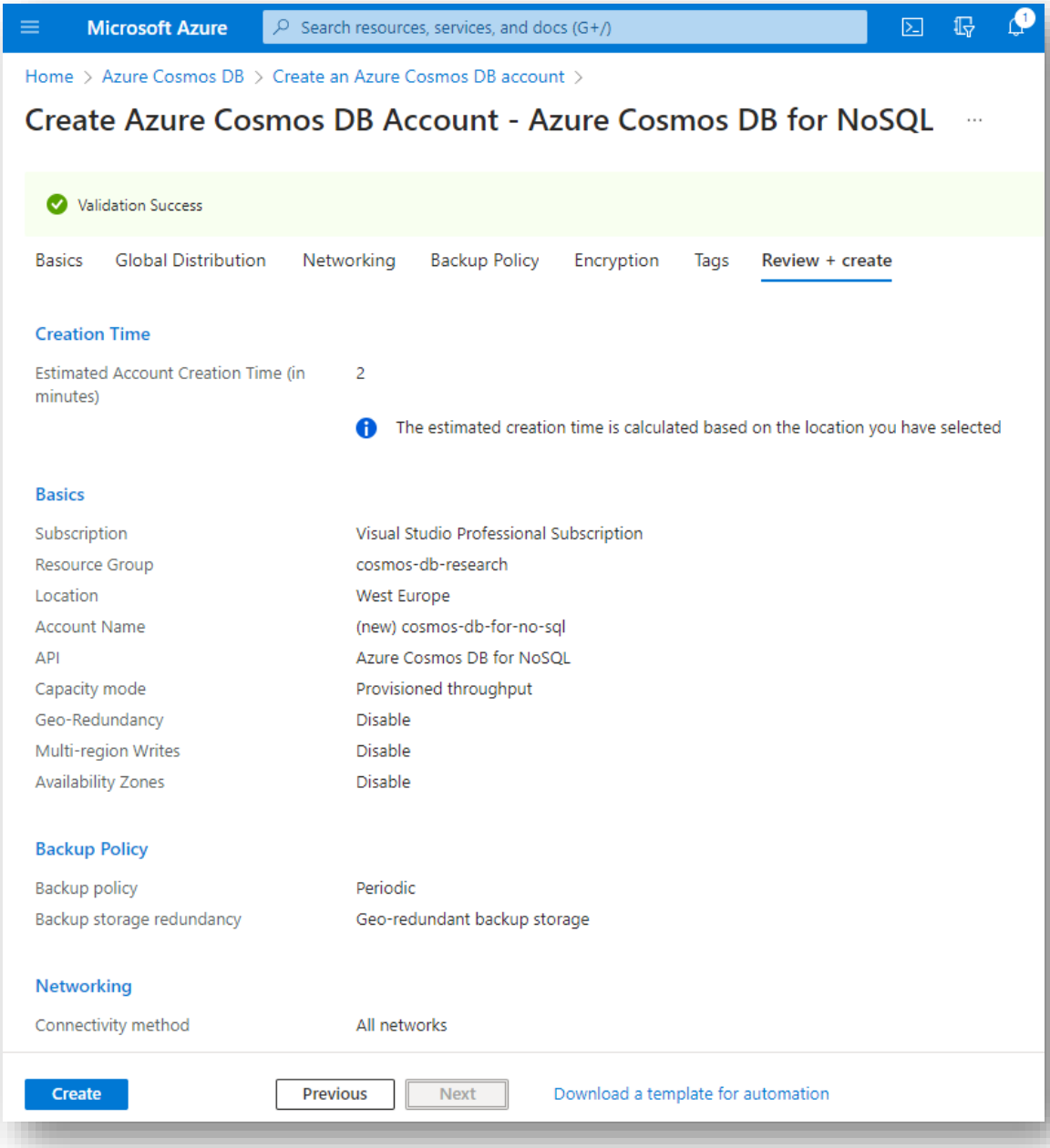

Рисунок 3.3 – Створення Azure Cosmos DB for NoSQL Account

Для MongoDB підійде Cosmos Db for MongoDB API Account, який було створено у Azure (див. рис. 3.4). Назва акаунту cosmos-db-for-mongo-db, також були встановлені налаштування резервного копіювання раз у 24 години та зберігання бекапів 2 дні, були обрані найнижчі параметри для зменшення вартості обслуговування створеної бази даних. Розташування бази даних було обрано найближчий сервер це (Europe) West Europe.

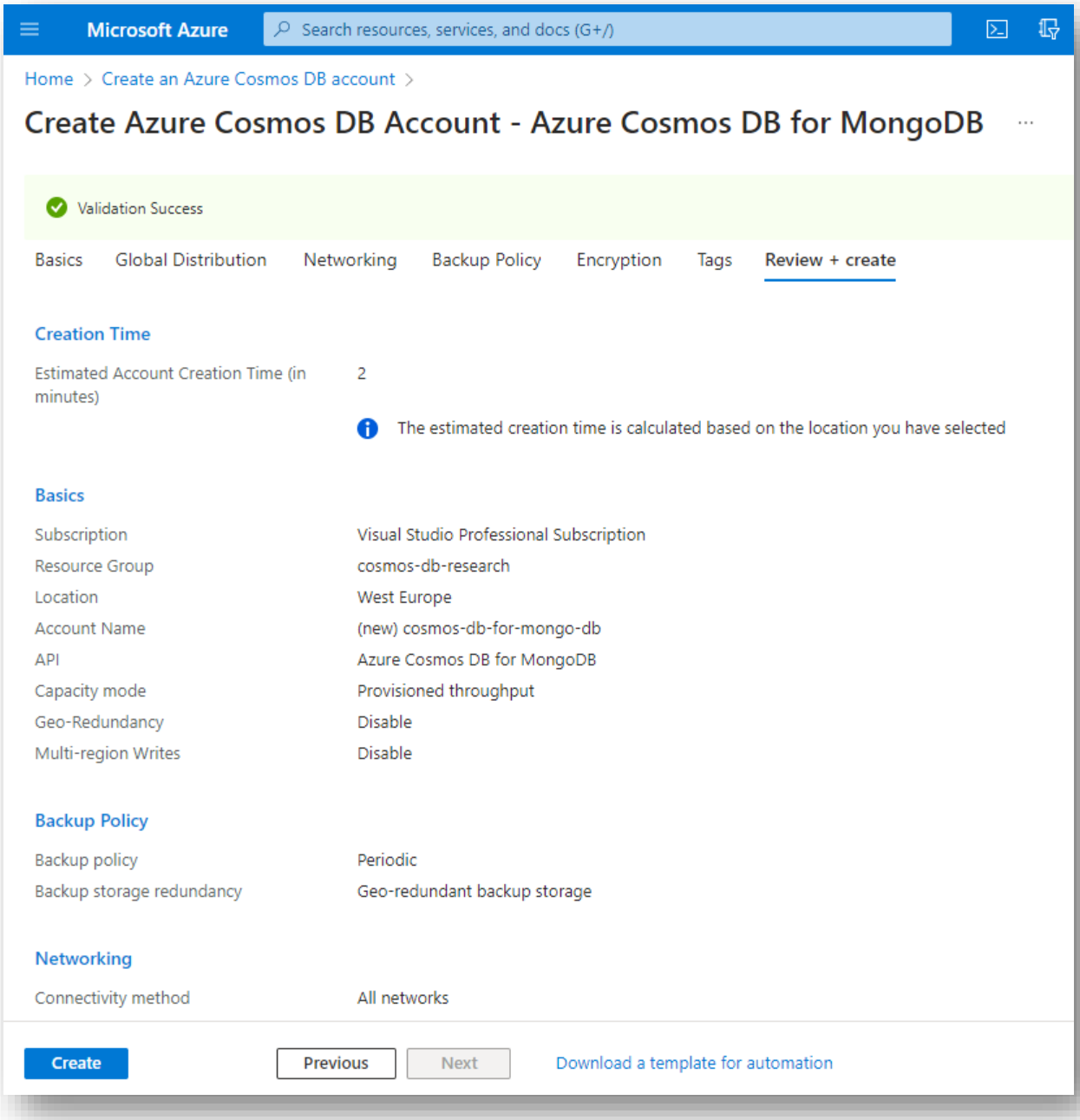

Рисунок 3.4 – Створення Azure Cosmos DB for MongoDB Account

Після створення Cosmos DB Account можемо перейти в розділ Data Explorer та створити нову базу даних Ecommerce (див. рис. 3.5). Оберемо ручне налаштування продуктивності бази даних та обмежимся базовими 400 RU/s для економії коштів за базу даних [17].

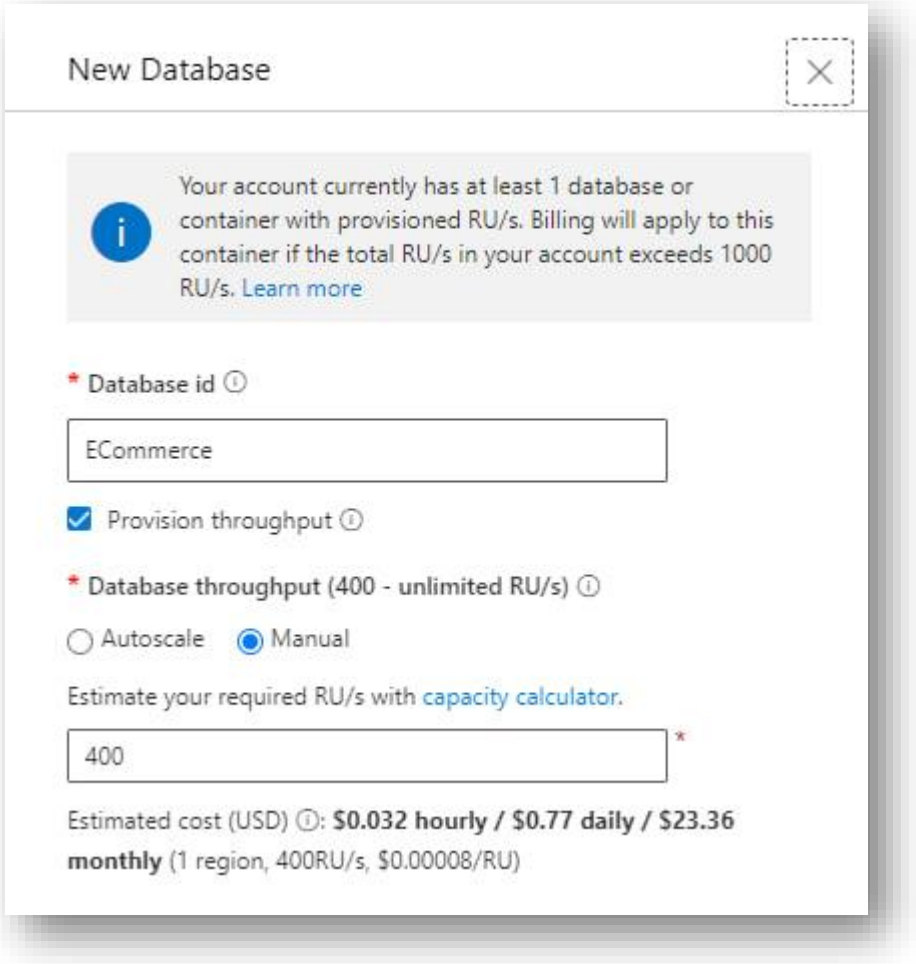

Рисунок 3.5 – Створення бази даних в розділі Data Explorer

Кількість RU, що потрібні для виконання операції в Cosmos DB, залежить від типу запиту, розміру документа, швидкості запису та інших факторів. Чим більше RU, тим вища продуктивність бази даних [18].

RU дозволяють керувати продуктивністю бази даних та гарантувати її стабільність при високих навантаженнях. Можна задати максимальну кількість

RU для контейнера Cosmos DB, що гарантує, що база даних оброблятиме запити із заданою швидкістю [19].

Azure Cosmos DB також дозволяє використовувати автоматичне масштабування RU, залежно від навантаження на базу даних. Якщо кількість запитів різко зростає, Cosmos DB може автоматично збільшити кількість RU для обробки додаткових запитів, і навпаки, якщо навантаження бази даних знижується, кількість RU може бути автоматично зменшено. Це дозволяє оптимізувати витрати на продуктивність бази даних, залежно від поточного навантаження.

Після створення бази даних можемо створити контейнер (див. рис. 3.6). Обираємо існуючу базу даних «Ecommerce», далі вказуємо назву контейнера «Products», обираємо автоматичне індексування та вказуємо назву поля, в якому буде зберігатися значення ключа секції «/partitionKey». Після цього натискаємо «Ok» для створення контейнера.

Cosmos DB Container – це логічний контейнер, який зберігає колекцію документів у базі даних Azure Cosmos DB. Контейнери Cosmos DB є одиницями масштабування для зберігання даних і є межі логічного поділу, які можна використовувати для управління доступом, продуктивністю та масштабованістю в базі даних.

Контейнери можуть містити будь-яку кількість документів, і кожен документ може містити різні поля та структури даних.

Контейнери Cosmos DB можуть бути створені в рамках однієї або декількох баз даних, і вони можуть бути налаштовані для масштабування вертикально (зміни розміру ресурсів) та горизонтально (зміни кількості екземплярів).

Контейнери також можуть бути налаштовані для реплікації даних у різних регіонах для забезпечення високої доступності та скорочення затримок при доступі до даних із різних регіонів.

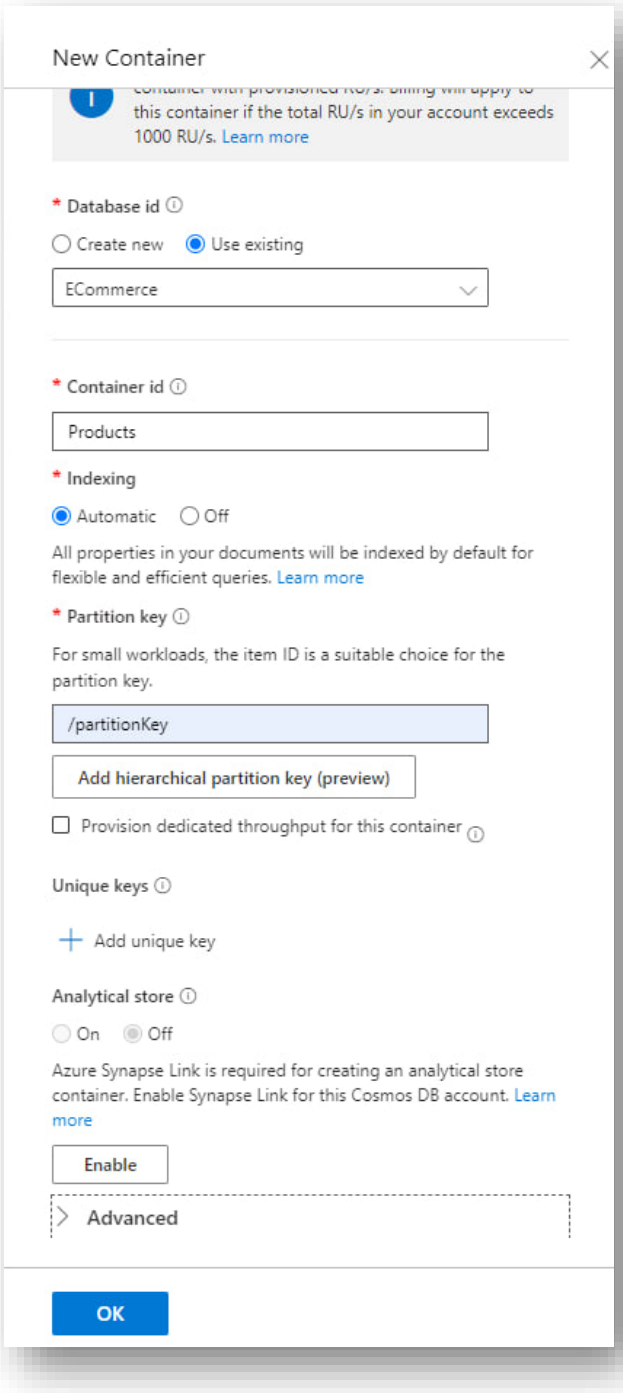

Рисунок 3.6 – Створення контейнера в розділі Data Explorer

Далі будем створювати контейнери використовуючи код, це зменшує час на створення бази даних та контейнерів у новому середовищі або регіоні, а також дає можливість швидко застосувати однакові налаштування до декількох контейнерів.

Також необхідно розрахувати пропускну здатність Cosmos DB акаунта. Коли ми створювали базу даних, то вказували throughput 400 RU/s, але потрібно враховувати, що в нас 2 контейнери і для кожного буде необхідно 400 RU/s тобто загалом 800 RU/s на акаунт. Якщо ця пропускна здатність буде недостатньої в налаштуваннях акаунта, ми отримаємо помилку обмеження в налаштуваннях акаунту під час запуску програми, не всі контейнери будуть створені та наш веб сервіс не запуститься. Для того, щоб встановити необхідне значення throughput необхідно перейти в розділ Cost Management та встановити нове значення (див. рис. 3.7).

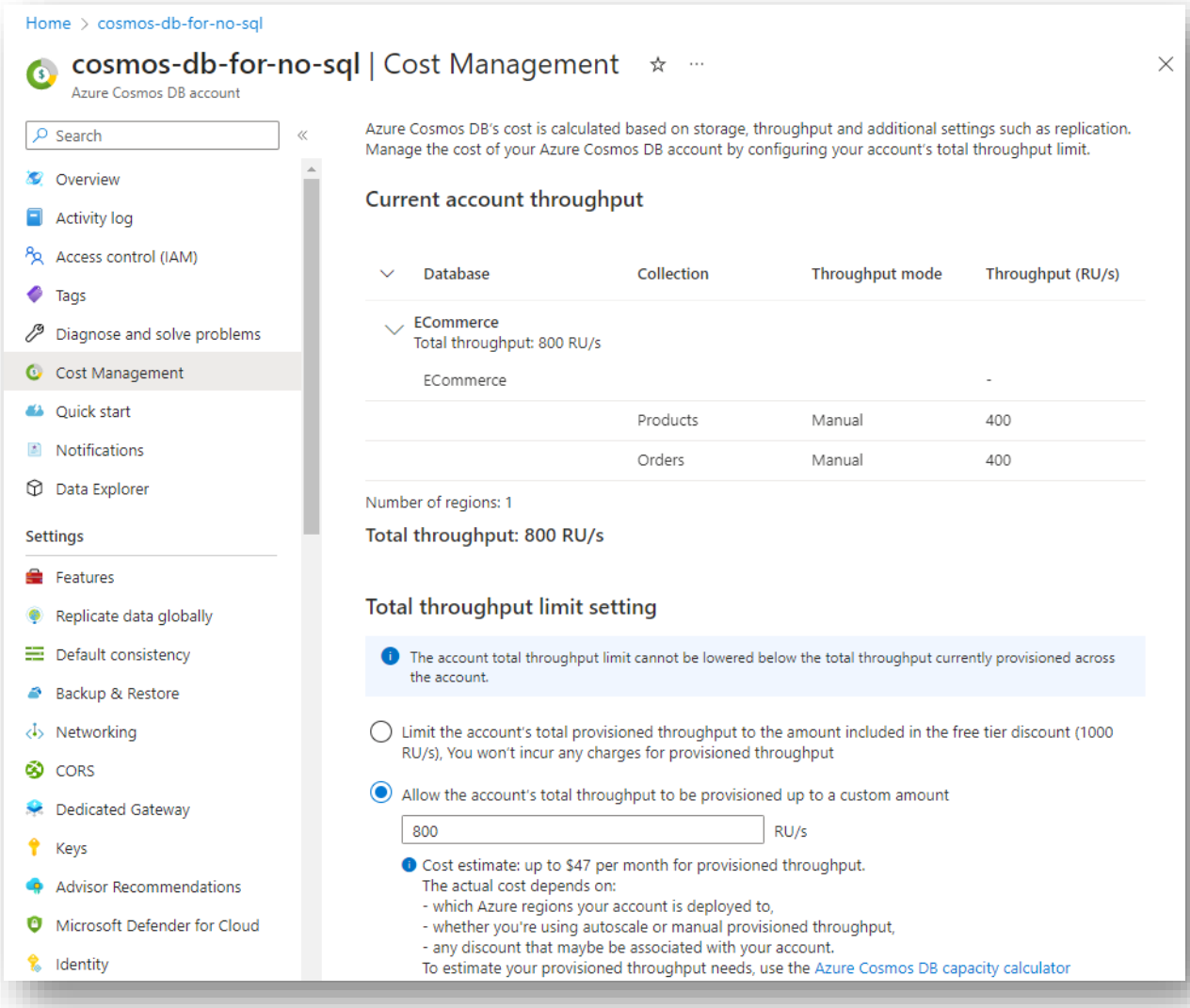

Рисунок 3.7 – Налаштування throughput limit у вкладці Cost Management

3.2 Опис загальних вимог до структури коду кожного з проектів

Для того, щоб тестування різних програмних реалізацій було достовірним необхідно мати структуру основних класів, які реалізують параметри підключення та налаштування бази даних. До таких класів належать CosmosDbSettings, CosmosDbExtensions та CosmosDbInitializer (див. рис. 3.8).

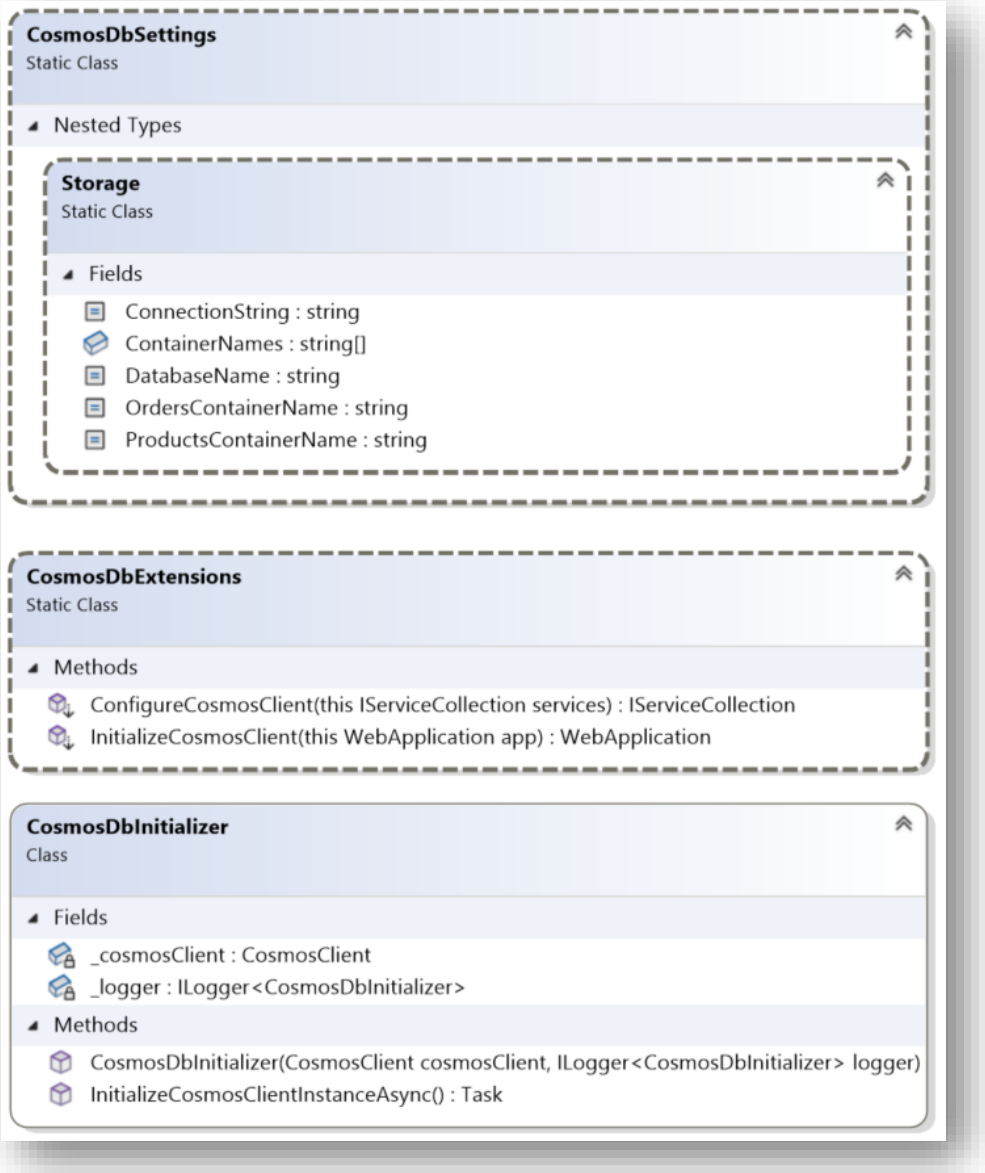

Рисунок 3.8 – Основні класи для роботи з базою даних.

За допомогою класу CosmosDbSettings ми можемо конфігурувати нашу програму, задавати такі параметри як строка підключення до бази даних (ConnectionString), назва бази даних (DatabaseName), список контейнерів (ContainerNames) та окремо назви для кожного із необхідних контейнерів, в нашому випадку для контейнерів Orders та Products (OrdersContainerName, ProductsContainerName).

CosmosDbExtensions клас містить в собі методи розширення для конфігурації системних класів для роботи з CosmosDb. Метод ConfigureCosmosClient() задає конфігурацію для підключення до бази даних, використовуючи параметри з класу CosmosDbSettings, та налаштовує параметри серіалізації об’єктів, після чого реєструє клас, який відповідає за роботу з базою даних, в контейнері інверсії залежностей.

CosmosDbInitializer клас необхідний для того, щоб перевірити чи створена база даних та усі необхідні контейнери під час підключення до Azure та створити їх, якщо необхідно.

Далі ці класи визиваються в головному класі Program, спочатку метод конфігурації, потім метод ініціалізації бази даних.

Кожна з програм повинна мати однаковий інтерфейс для проведення експериментів, тому повинна мати два контроллери для роботи з кожною з колекцій: OrdersController та ProductsController (див. рис. 3.9). Ці контроллери є реалізацією RESTful API. RESTful API — це архітектурний стиль інтерфейсу прикладної програми (API), який використовує запити HTTP для доступу та використання даних. Ці дані можна використовувати для типів даних GET, PUT, POST і DELETE, які стосуються читання, оновлення, створення та видалення операцій щодо ресурсів. Кожен з них має методи для виконання описаних запитів до БД: Create(), GetByPartitionKey(), GetById(), Update(), Delete(), SoftDelete().

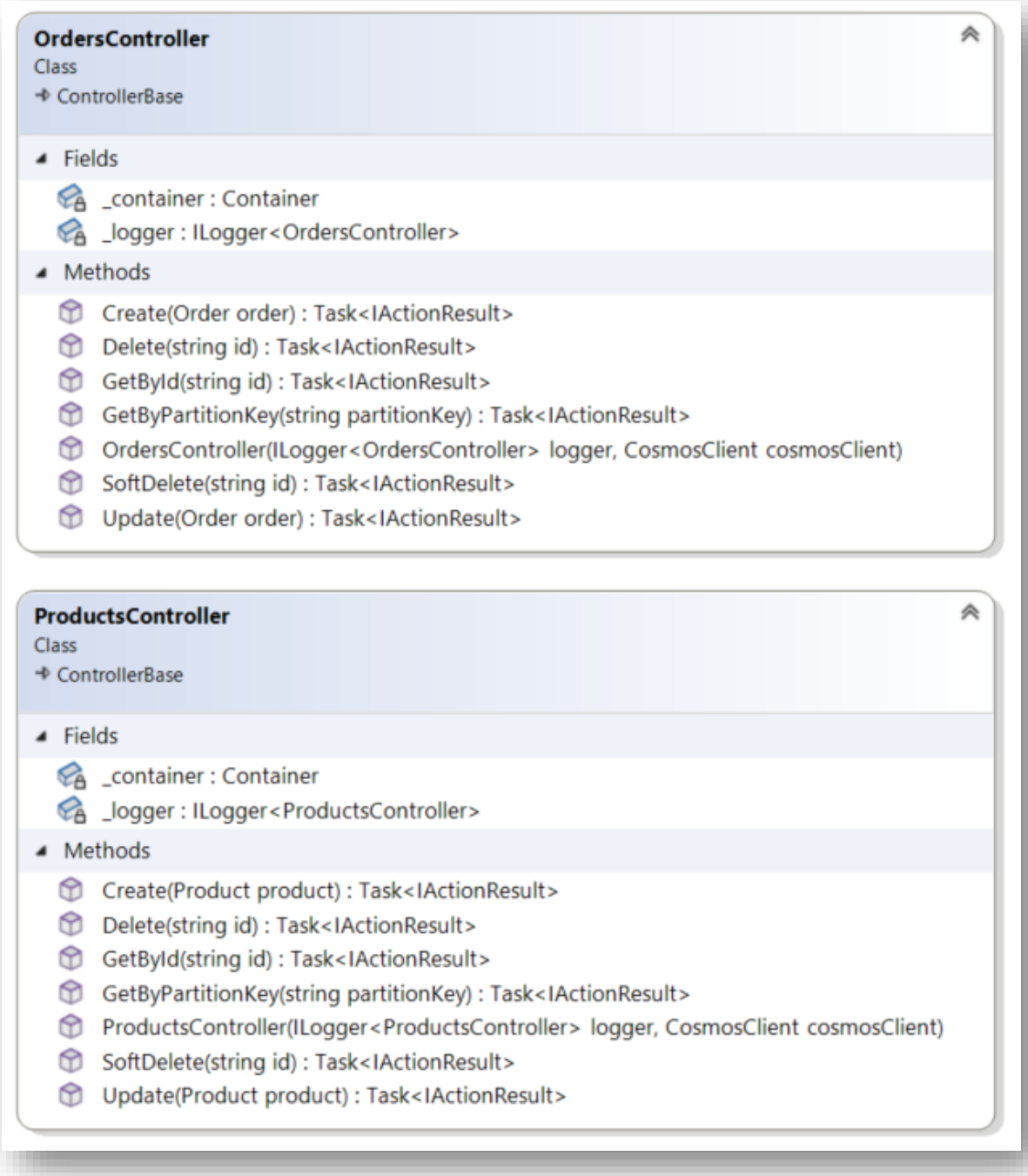

Рисунок 3.9 – Класи контроллерів для роботи з колекціями.

Кожна програма повинна мати налаштування Swagger. Swagger - це набір інструментів, який дозволяє автоматично описувати API на основі коду, який нам знадобиться для проведення експериментів.

Розглянемо клас Program.cs, який конфігурує та запускає веб додаток (див. рис. 3.10). Він буде однаковим для кожної з програм завдяки тому, що ми

інкапсулювали логіку підключення до БД в методах розширення описаних вище, ці методи лише матимуть різну назву для кожного з проектів:

– ConfigureCosmosClient() та InitializeCosmosClient() для Cosmos Client;

– ConfigureCosmosEf() та InitializeCosmosEf() для Entity Framework Core Cosmos;

– С ConfigureCosmosMongoDb() та InitializeCosmosMongoDb() для MongoDB.

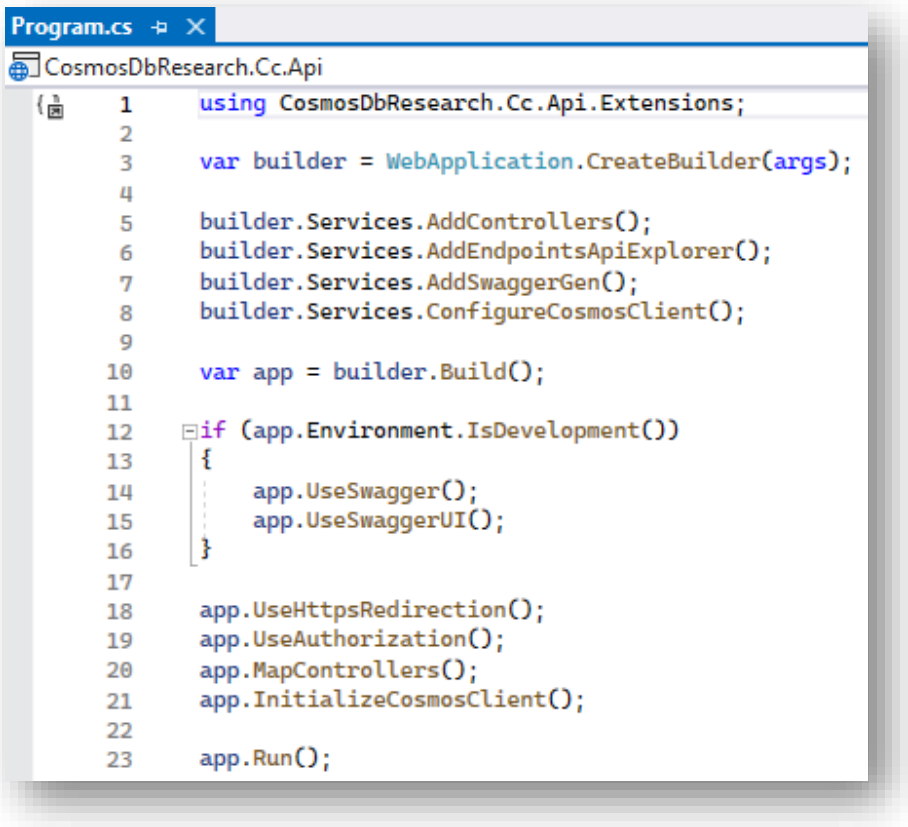

Рисунок 3.10 – Клас Program.cs в CosmosDbResearch.Cc.Api.csproj

3.3 Опис програмної реалізації Cosmos DB API for NoSQL (Cosmos Client)

Для реалізації підходу Cosmos Client було створено ASP.NET Core Web API проект CosmosDbResearch.Cc.Api у програмному рішенні CosmosDbResearch у Visual Studio 2022 [20] (див. рис. 3.11).

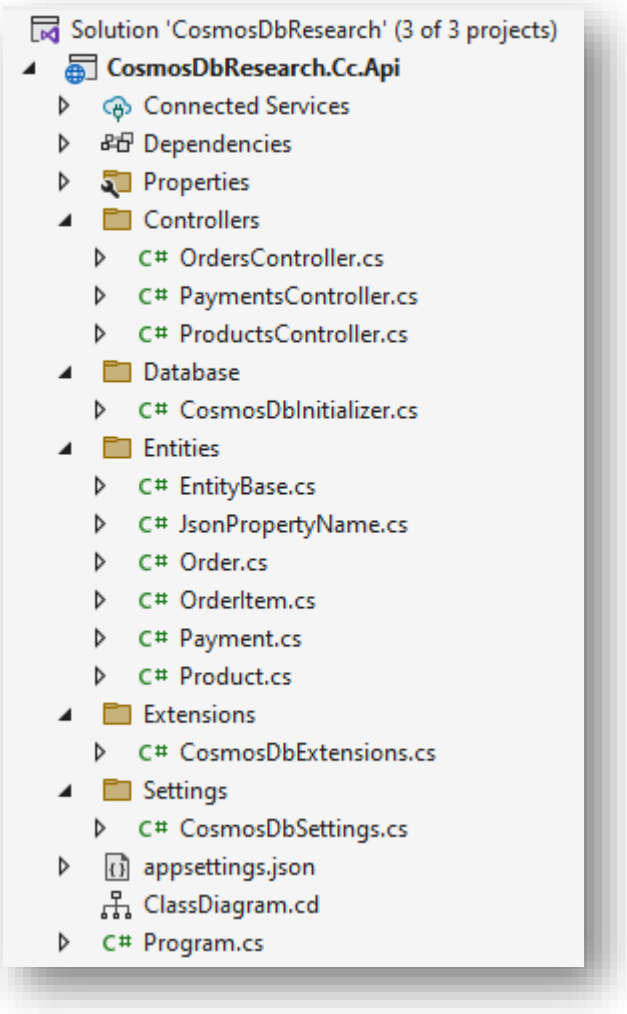

Рисунок 3.11 – ASP.NET Core Web API проект CosmosDbResearch.Cc.Api

Після завершення конфігурації клієнта [21] для роботи з базою даних можемо використовувати його у контролері для проведення експериментальних досліджень (див. рис. 3.12).

```
[Route("api/orders")]
[ApiController]
public class OrdersController : ControllerBase
{
    private readonly ILogger<OrdersController> _logger;

    private Container _container = null!;

    public OrdersController(
        ILogger<OrdersController> logger,
        CosmosClient cosmosClient)
    {
        _logger = logger;
        _container = cosmosClient.GetContainer(
            CosmosDbSettings.Storage.DatabaseName,
            CosmosDbSettings.Storage.OrdersContainerName);
    }
```

Рисунок 3.12 – Клас OrdersController.cs в CosmosDbResearch.Cc.Api.csproj.

Давайте розглянемо структуру класів моделей бази даних. Існує базовий клас EntityBase, який має спільні властивості для всіх сутностей, а саме це Id, IsDeleted, PartirionKey (див. рис. 3.13). Усі інші класи моделей є дочірніми для цього класу.

В якості прикладу розглянемо дочірній клас Order.cs (див. рис. 3.14). Він має перевизначений PartitionKey, який вказує на загальну вартість замовлення. В базовому класі це поле зроблено абстрактним для того щоб кожен дочірній клас міг перевизначити ключ розділу в залежності від своєї бізнес логіки. Також у замовлення є такі властивості як OrderDate, OrderItems, OrderPrice та Payment.

Можемо побачити над кожною властивістю атрибут [JsonPropertyName()], який використовується для визначення якому рядку в JSON файлі буде відповідати це поле. В цей атрибут передаються константи з назвою полів, які визначено в класі JsonPropertyName.cs [22]. Також цей клас має перевизначений метод ToString() для того щоб замовлення було легко виводити в консоль.

```
5 references
public abstract class EntityBase
{
    [JsonPropertyName(JsonPropertyName.Id)]
    9 references
    public virtual string Id { get; set; }

    [JsonPropertyName(JsonPropertyName.PartitionKey)]
    7 references
    public abstract string PartitionKey { get; }

    [JsonPropertyName(JsonPropertyName.IsDeleted)]
    1 reference
    public bool IsDeleted { get; set; }

    0 references
    public EntityBase()
    {
        Id = Guid.NewGuid().ToString();
    }
}
```

Рисунок 3.13 – Базовий клас для всіх моделей сутностей EntityBase.cs.

```
public class Order : EntityBase
{
    [JsonPropertyName(JsonPropertyName.PartitionKey)]
    public override string PartitionKey => OrderPrice.ToString();

    [JsonPropertyName(JsonPropertyName.Order.OrderDate)]
    public DateTime OrderDate { get; set; }

    [JsonPropertyName(JsonPropertyName.Order.OrderItems)]
    public List<OrderItem> OrderItems { get; set; } = null!;

    [JsonPropertyName(JsonPropertyName.Order.OrderPrice)]
    public decimal OrderPrice { get; set; }

    [JsonPropertyName(JsonPropertyName.Order.Payment)]
    public Payment Payment { get; set; } = null!;

    public override string ToString()
    {
        return JsonSerializer.Serialize(this);
    }
}
```

Рисунок 3.14 – Дочірній клас сутності Order.cs

Після запуску веб сервісу, ми побачимо в консолі, що база даних та всі контейнери були створені (див. рис. 3.15). Після цього можемо перейти на Azure Portal в акаунт бази даних та перевірити створені базу даних та контейнери на вкладці Data Explorer (див. рис. 3.16).

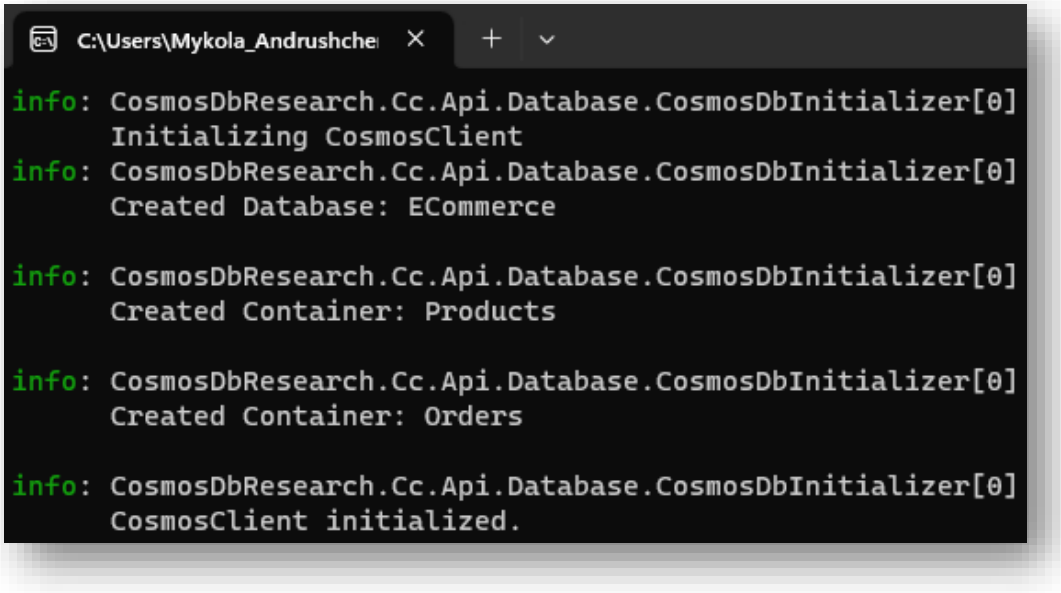

Рисунок 3.15 – Консоль запущеного розробленого додатку

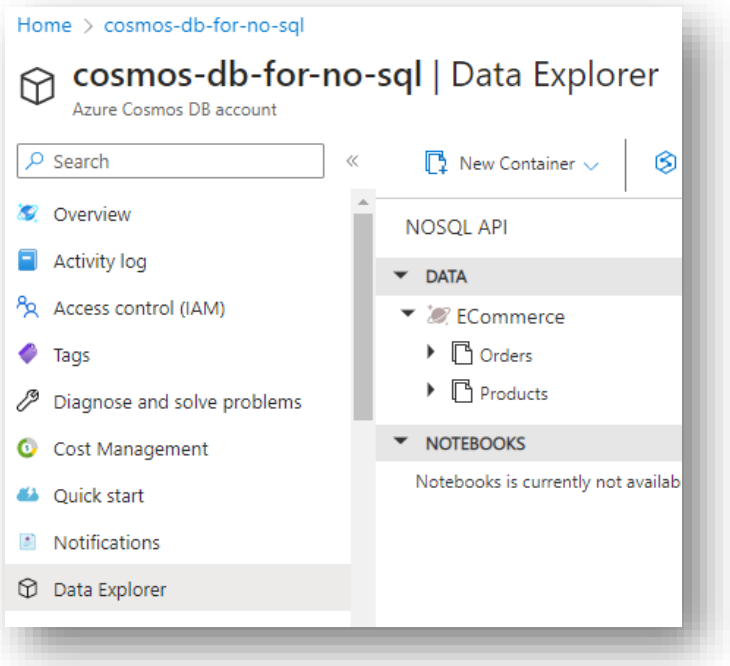

Рисунок 3.16 – Створена база даних і контейнері на Azure Portal

3.4 Опис програмної реалізації Cosmos DB API for NoSQL (Entity Framework Core Cosmos)

Для реалізації підходу Entity Framework Core Cosmos було створено ASP.NET Core Web API проект CosmosDbResearch.Ef.Api у програмному рішенні CosmosDbResearch у Visual Studio 2022 (див. рис. 3.17).

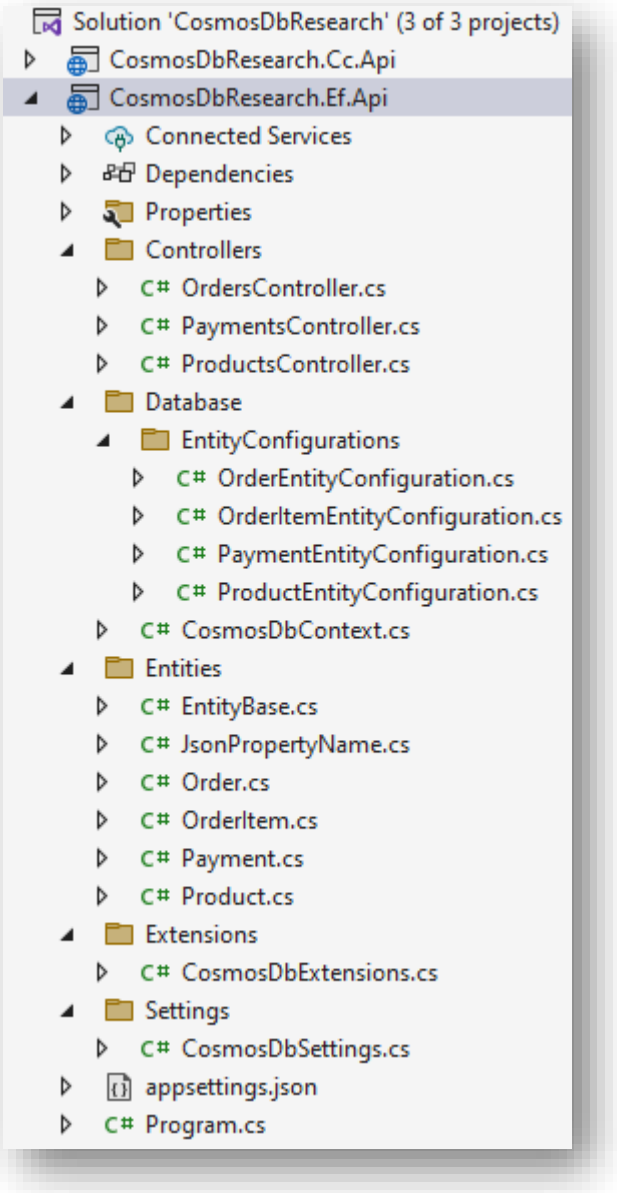

Рисунок 3.17 – ASP.NET Core Web API проект CosmosDbResearch.Ef.Api.

Давайте більш детально розглянемо клас CosmosDbContext (див. рис. 3.18). Для реалізації підходу Code First в контекст додані набори даних для кожної з колекцій. Також клас успадковується від базового класу DbContext, що необхідно зробити за інструкцією використання бібліотеки Entity Framework. В цьому класі ми бачимо перевизначений метод OnModelCreating(), який за допомогою рефлексії під час виконання застосунку отримує та застосовую всі класи типу IEntityTypeConfiguration<> в виконуваній збірці [23]. В цих класах знаходиться конфігурація для кожної колекції, в якій описується до якого контейнера належить певна сутність, які вона має поля, які вкладені моделі, який унікальний ідентифікатор, ключ розділу, яку назву мають відповідні поля в контейнері та інші (див. рис. 3.19).

```
public class CosmosDbContext : DbContext
{
    public CosmosDbContext(DbContextOptions<CosmosDbContext> options)
        : base(options) { }

    public DbSet<Product> Products { get; set; }

    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultContainer(CosmosDbSettings.Storage.OrdersContainerName);
        modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
    }
}
```

Рисунок 3.18 – Клас CosmosDbContext.cs в CosmosDbResearch.Ef.Api.csproj

Бачимо, що замовлення має дві вкладені сутності. Перша це платіж з відношенням OwnsOne() – має лише один, друга це товари в замовленні OwnsMany() – має декілька.

Після налаштування конфігурації сутностей та класу CosmosDbContext ми можемо використовувати його в контролері, щоб працювати з контекстом бази даних (див. рис. 3.20).

```
public class OrderEntityConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.ToContainer(CosmosDbSettings.Storage.OrdersContainerName);
        builder.HasNoDiscriminator();

        builder.HasKey(x => x.Id);
        builder.HasPartitionKey(x => x.PartitionKey);

        builder.OwnsOne(
            x => x.Payment,
            y =>
            {
                y.ToJsonProperty(JsonPropertyName.Order.Payment);
                y.Property(p => p.PaymentDate).ToJsonProperty(JsonPropertyName.Payment.PaymentDate);
                y.Property(p => p.Price).ToJsonProperty(JsonPropertyName.Payment.Price);
            });
        builder.OwnsMany(
            x => x.OrderItems,
            y =>
            {
                y.ToJsonProperty(JsonPropertyName.Order.OrderItems);
                y.Property(p => p.ProductId).ToJsonProperty(JsonPropertyName.OrderItem.ProductId);
                y.Property(p => p.Quantity).ToJsonProperty(JsonPropertyName.OrderItem.Quantity);
                y.Property(p => p.Price).ToJsonProperty(JsonPropertyName.OrderItem.Price);
            });

        builder.Property(x => x.Id).ToJsonProperty(JsonPropertyName.Id);
        builder.Property(x => x.PartitionKey).ToJsonProperty(JsonPropertyName.PartitionKey);
        builder.Property(x => x.IsDeleted).ToJsonProperty(JsonPropertyName.IsDeleted);
        builder.Property(x => x.OrderDate).ToJsonProperty(JsonPropertyName.Order.OrderDate);
        builder.Property(x => x.OrderPrice).ToJsonProperty(JsonPropertyName.Order.OrderPrice);
    }
}
```

Рисунок 3.19 – Клас OrderEntityConfiguration.cs в CosmosDbResearch.Ef.Api.csproj

Структура класів така сама як і при роботі з Cosmos Client, ми також маємо базовий клас EntityBase для усіх сутностей бази даних, поля також мають атрибути [JsonPropertyName()] для сериалізації в JSON [24].

```
[Route("api/orders")]
[ApiController]
public class OrdersController : ControllerBase
{
    private readonly ILogger<OrdersController> _logger;

    private CosmosDbContext _cosmosDbContext = null!;

    public OrdersController(
        ILogger<OrdersController> logger,
        CosmosDbContext cosmosDbContext)
    {
        _logger = logger;
        _cosmosDbContext = cosmosDbContext;
    }
```

Рисунок 3.20 – Клас OrdersController.cs в CosmosDbResearch.Ef.Api.csproj

Після запуску веб сервісу, ми побачимо в консолі, що база даних та всі контейнери були створені (див. рис. 3.21). Після цього можемо перейти на Azure Portal в акаунт бази даних та перевірити створені базу даних та контейнери на вкладці Data Explorer (див. рис. 3.22).

```
C:\Users\mandrushchenko\so
info: Program[0]
      Database and all containers were created
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7087
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5022
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

Рисунок 3.21 – Консоль запущеного розробленого додатку

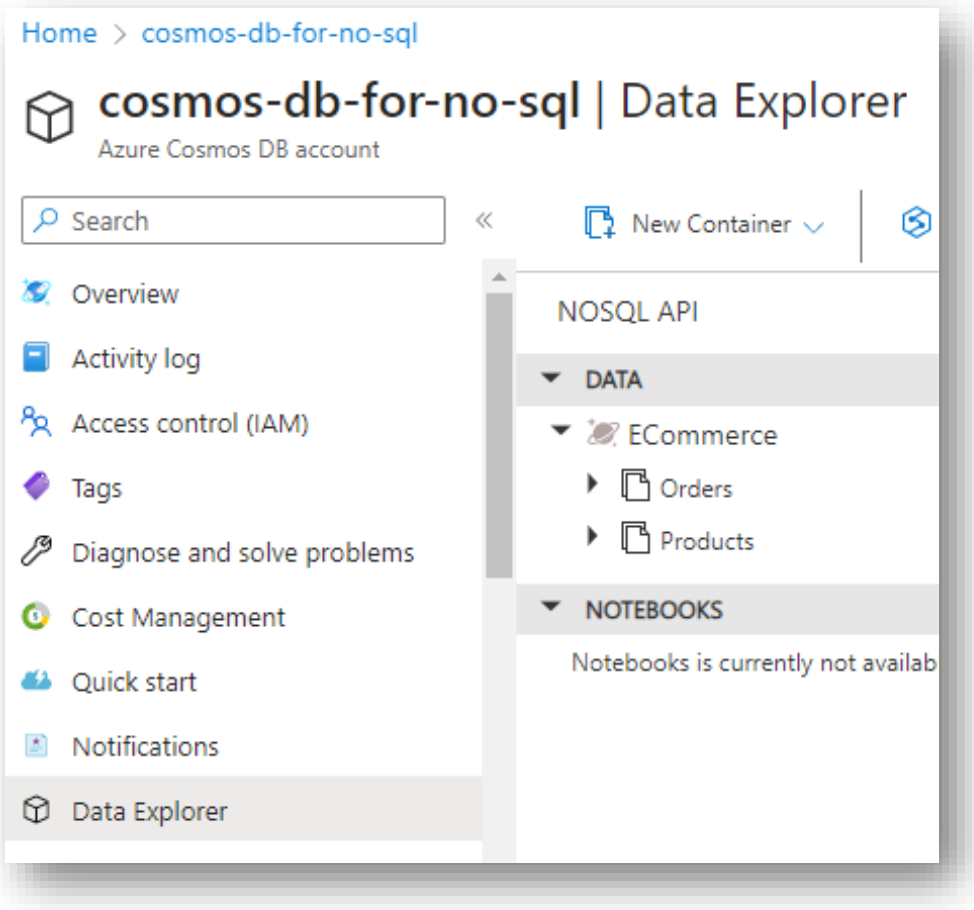

Рисунок 3.22 – Створена база даних і контейнері на Azure Portal

3.5 Опис програмної реалізації Cosmos DB API for MongoDB (Mongo Client)

Для реалізації підходу Mongo DB було створено ASP.NET Core Web API проект CosmosDbResearch.Md.Api у програмному рішенні CosmosDbResearch у Visual Studio 2022 [25] (див. рис. 3.23).

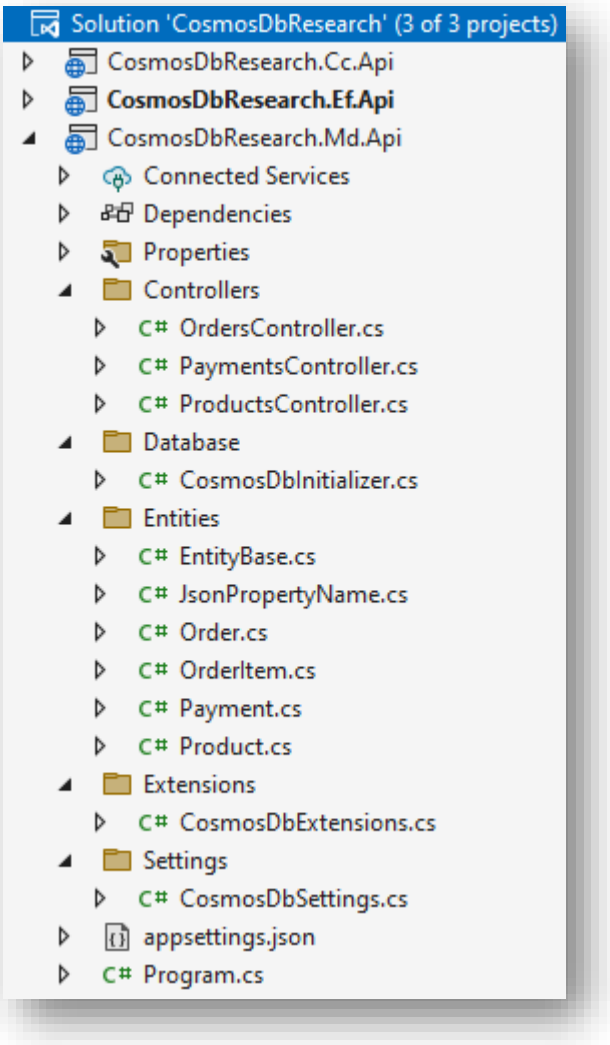

Рисунок 3.23 – ASP.NET Core Web API проект CosmosDbResearch.Md.Api.

Після завершення конфігурації клієнта для роботи з базою даних можемо використовувати його у контролері для проведення експериментальних досліджень [26] (див. рис. 3.24). Як бачите для Mongo DB ми додавали в контейнер і Mongo Client і IMongoDatabase і тепер в контролері можемо отримати обидва класи. IMongoDatabase використовується для отримання колекції з бази даних, а також буде використовуватися в методах контролера для отримання кількості RU які були використані на останній запит до бази даних, ці метрики нам потрібні для експериментів, але вони не приходять автоматично в відповіді від сервера як це реалізовано в інших підходах.

```
[Route("api/orders")]
[ApiController]
public class OrdersController : ControllerBase
{
    private readonly ILogger<OrdersController> _logger;
    private IMongoDatabase _database;
    private IMongoCollection<Order> _ordersCollection;

    public OrdersController(
        ILogger<OrdersController> logger,
        IMongoDatabase mongoDatabase)
    {
        _logger = logger;
        _database = mongoDatabase;
        _ordersCollection = mongoDatabase.GetCollection<Order>(
            CosmosDbSettings.Storage.OrdersContainerName);
    }
```

Рисунок 3.24 – Клас OrdersController.cs в CosmosDbResearch.Md.Api.csproj

Відмінність Mongo DB в тому, що тут документи зберігаються у форматі BSON, тому для конфігурації полів сутностей потрібно використовувати не [JsonPropertyName()] атрибут, а [BsonElement()] атрибут (див. рис. 3.25). Також для того щоб визначити поле унікального ключа необхідний атрибут [BsonId].

```
    6 references
6   public abstract class EntityBase
7   {
8       [BsonId]
        4 references
9       public string? Id { get; set; }
10
11      [BsonElement(BsonPropertyName.PartitionKey)]
        4 references
12      public virtual string PartitionKey { get; set; }
```

Рисунок 3.25 – Приклад атрибутів для властивостей Mongo DB

Після запуску веб сервісу, ми побачимо в консолі, що база даних та всі контейнери були створені (див. рис. 3.26). Після цього можемо перейти на Azure Portal в акаунт бази даних та перевірити створені базу даних та контейнери на вкладці Data Explorer (див. рис. 3.27).

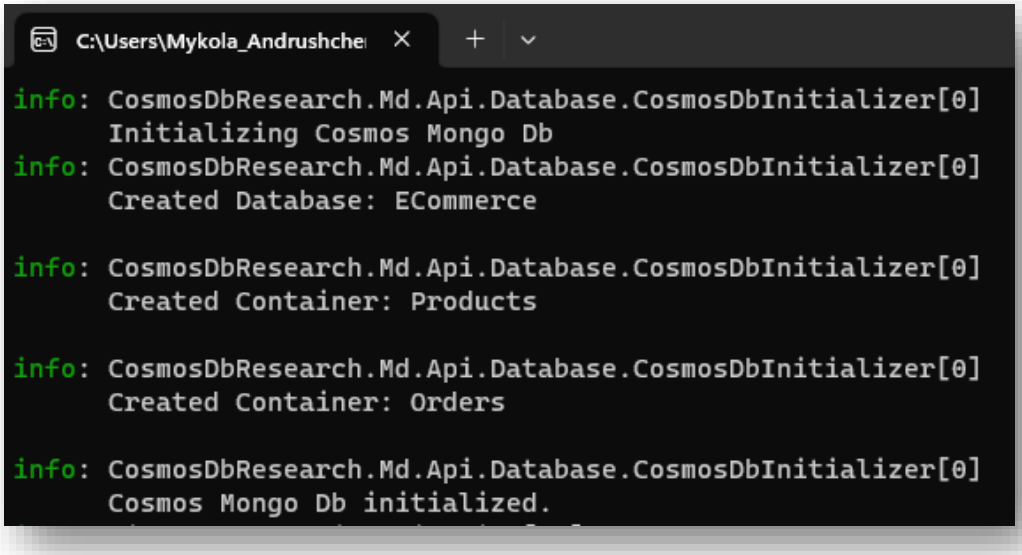

Рисунок 3.26 – Консоль запущеного розробленого додатку

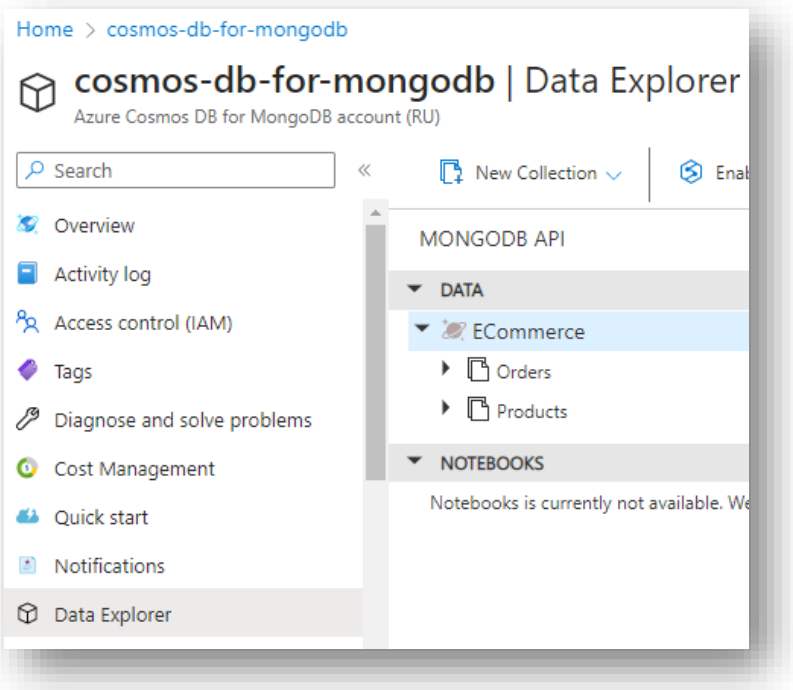

Рисунок 3.27 – Створена база даних і контейнері на Azure Portal

Після того як була створена програмна реалізація для всіх трьох різних Cosmos DB API можна переходити до написання коду для проведення досліджень.

3.6 Опис програмного підходу до проведення експериментів

Під час проведення експериментів нам необхідно отримувати три метрики: швидкість виконання запиту у мс, кількість витраченої пам’яті та кількість витрачених RU зі сторони БД. Для виконання замірів часу буде використовуватися клас Stopwatch. Для цього потрібно створити екземпляр класу, викликати метод Start() перед виконанням дії, час виконання якої нам необхідно заміряти, після цього викликати метод Stop() і отримати час виконання з властивості Elapsed. Для заміру кількості використаної пам’яті будемо використовувати системний клас GC, який має метод GetTotalMemoty(). Отримати RU ми можемо з відповіді на запит до БД у випадку Cosmos Client та Entity Framework Cosmos Core, а от для MongoClient доведеться робити додатковий запит в БД для отримання цієї метрики (див. рис. 3.28).

```
var memoryBefore = GC.GetTotalMemory(true);
var stopWatch = new Stopwatch();
stopWatch.Start();

var orderCreated = await _cosmosDbContext.Orders.AddAsync(order);
await _cosmosDbContext.SaveChangesAsync();

var memoryAfter = GC.GetTotalMemory(true);
stopWatch.Stop();
```

Рисунок 3.28 – Приклад коду для заміру метрик

Для того, щоб було зручно та швидко тестувати запити до БД було вирішено використовувати Swagger OpenAPI, який представляє опис всіх методів

контролера та має зручний інтерфейс для вводу даних, які будуть відправлені на сервер (див рис. 3.29).

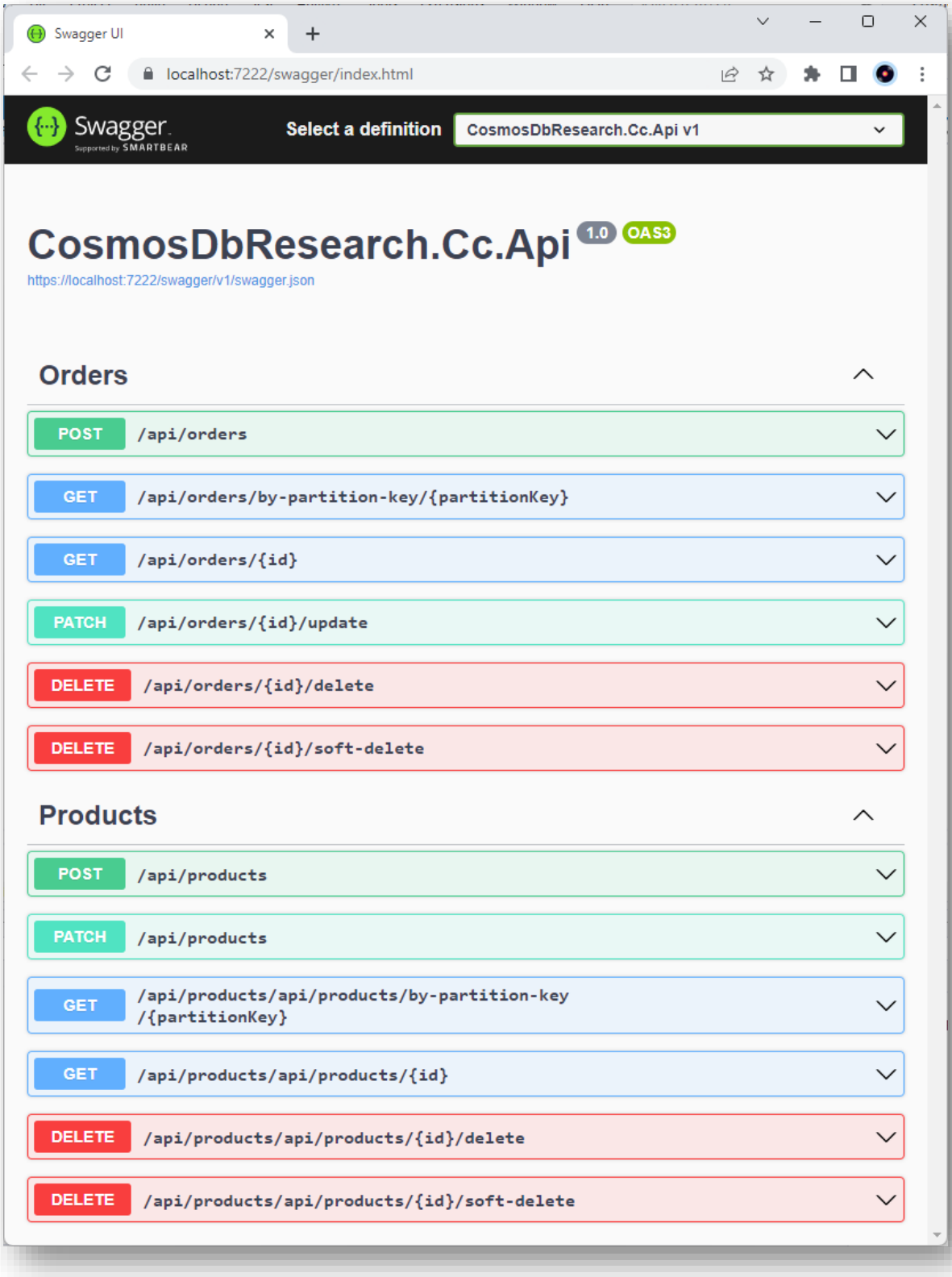

Рисунок 3.29 – Swagger OpenAPI на прикладі CosmosDbResearch.Cc.Api

4 ОПИС ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Проведення експериментальних досліджень

Після завершення планування експериментів та написання програм для їх проведення можна перейти до замірів усіх необхідних параметрів, а саме швидкості виконання запитів у мс, кількості витраченої програмою пам’яті на виконання запиту та кількості RU зі сторони БД. Для кожного експерименту проводилося виконання запитів та розраховувалося їх середнє значення.

Результати замірів за метриками для «INSERT-запит» представлені у таблиці 9.

Таблиця 9 – Результати виконання запитів на створення замовлення

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	48.55	27776	10.67
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	54.22	51999	10.67
Cosmos DB API for MongoDB (Mongo Client)	48	17232	9.05

Результати замірів за метриками для «SELECT-запит №1» представлені у таблиці 10.

Результати замірів метрик для «SELECT-запит №2» представлені у таблиці 11.

Результати замірів метрик для «UPDATE-запит» представлені у таблиці 12.

Таблиця 10 – Результати виконання запитів на отримання замовлень по полю PartitionKey

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	110.33	86605	3.12
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	2.77	8453	3.22
Cosmos DB API for MongoDB (Mongo Client)	64.55	35149	7.17

Таблиця 11 – Результати виконання запитів на отримання замовлення по Id

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	86.44	54949	2.83
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	55.44	29012	2.83
Cosmos DB API for MongoDB (Mongo Client)	130.77	25977	3.11

Таблиця 12 – Результати виконання запитів на оновлення замовлення

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	49.11	42915	10.67
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	106.77	21033	10.67
Cosmos DB API for MongoDB (Mongo Client)	179.11	32885	11.29

Результати замірів метрик для «SOFT DELETE-запит» представлені у таблиці 13.

Таблиця 13 – Результати виконання запитів на м’яке видалення замовлення

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	47.88	58895	10.67
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	104	28076	10.67
Cosmos DB API for MongoDB (Mongo Client)	108.77	28662	11.29

Результати замірів метрик для «DELETE-запит» представлені у таблиці 14.

Таблиця 14 – Результати виконання запитів на видалення замовлення

API /Метрики	Швидкість виконання запиту (мс)	Кількість витраченої пам’яті (Byte)	Вартість виконання (RU)
Cosmos DB API for NoSQL (Cosmos Client)	50.11	54949	10.67
Cosmos DB API for NoSQL (Entity Framework Core Cosmos)	106	18560	10.67
Cosmos DB API for MongoDB (Mongo Client)	99.22	31520	7.05

4.2 Аналіз отриманих результатів

Бачимо, що запит на створення замовлення найбільш ефективно та швидко виконало MongoDB API, на це було витрачено менше пам’яті, а також менше RU, що означає, що такий запит буде коштувати дешевше. Це пов’язано з тим, що MongoClient не повертає відповідь від серверу на відміну від CosmosClient та EF Core Cosmos.

Проаналізувавши запити на отримання замовлення по PartitionKey та по Id можемо побачити ефективність Entity Framework Core Cosmos, який краще

оптимізований для таких операцій. Якщо порівнювати кількість RU для цих запитів можемо побачити що CosmosClient та EF Core Cosmos мають приблизно однакову вартість виконання запиту, а от MongoClient програє по цьому параметру.

Розглядаючи результати запитів на оновлення замовлення та на м’яке видалення можемо побачити, що CosmosClient майже в два рази швидше виконує цю операцію ніж EF Core Cosmos та MongoClient, але витрачає на це майже в два рази більше пам’яті. По своїй суті м’яке видалення це той самий запит на оновлення сутності зі зміною лише одного поля. З точки зору ефективності у RU перші два підходи на рівних з показником у 10.67 RU проти 11.29 RU у MongoClient.

Якщо ми проаналізуємо запити на видалення замовлення, то побачимо, що EF Core Cosmos та Mongo Client виконують цю операцію майже в два рази швидше, але витрачають на це значно більше пам’яті. Але MongoClient робить видалення за 7.05 RU проти 10.67 RU у CosmosClient та EF Core Cosmos.

За обсягом коду, який треба написати для тої чи іншої реалізації, значно виграє Entity Framework Core Cosmos, для нього код виходить компактним. Cosmos Client та Mongo Client мають приблизно однаковий стиль написання коду, тому за об’ємом коду ці два підходи майже однакові.

Загалом кожен з цих програмних підходів має свої переваги та недоліки, і вибір залежить від конкретного завдання та переваг розробника. Проте, можна запропонувати загальні рекомендації:

– якщо ми говоримо про вартість бази даних або необхідно оптимізувати витрати на базу даних, та якщо є багато запитів на створення та видалення записів, краще обрати Cosmos DB API for MongoDB API (Mongo Client), бо цей програмний підхід використовує меншу кількість RU на виконання таких запитів. Якщо ж частіше використовуються операції оновлення та отримання даних, то однаково економними будуть Cosmos DB API for NoSQL (Cosmos Client та Entity Framework Core Cosmos) порівняно з Mongo Client;

– якщо необхідно багато та швидко читати дані з БД, слід використовувати Cosmos DB API for NoSQL з Entity Framework Core Cosmos, який може вивантажувати контекст бази даних в пам’ять та швидко оброблювати запити;

– якщо у нас є обмеження для програми по пам’яті, слід відмовитись від Entity Framework Core Cosmos, бо він використовує багато пам’яті для роботи з контекстом БД;

– якщо необхідна оптимізація налаштування швидкості отримання певної групи об’єктів, слід налаштувати ключ розділу Cosmos DB, за допомогою якого запити на отримання певного розділу оптимізуються. Найкраще цю оптимізацію можна помітити для Entity Framework Core Cosmos для SELECT-запиту за PartitionKey;

– якщо говорити про кількість коду, слід враховувати, що для невеликих проектів вибір якогось з підходів не буде мати значних переваг для кожного з API. Entity Framework Core Cosmos потребує найменше коду для роботи з базою даних, але найбільше для конфігурації підключення до БД, саме тому ці переваги будуть помітні лише у великих проектах зі складною структурованою логікою.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було проведено аналіз проблемної області дослідження, описано відмінності реляційних та нереляційних баз даних, їх переваги та недоліки. Після чого було проведено вирішення багатокритеріальної задачі з вибору найбільш підходящої бази даних для дослідження, нею виявилася Cosmos DB.

Було проведено аналіз Cosmos DB API та обрано шляхом вирішення багатокритеріальної задачі найбільш підходящі для дослідження API, а саме Cosmos DB API for NoSQL (Cosmos Client), Cosmos DB API for NoSQL (Entity Framework Core Cosmos) та Cosmos DB API for MongoDB (Mongo Client).

Було розроблено план експериментального дослідження, в якості предметної області була обрана електронна комерція. В якості критеріїв порівняння програмних реалізацій були обрані швидкість виконання запитів (мс), кількість витраченної пам’яті на виконання запиту (byte), вартість запиту (RU) та кількість коду, який необхідно написати.

Була спроектована логічна модель даних для обраної предметної області та розроблена база даних. Було створено Azure Acount з усіма необхідними ресурсами для роботи з Cosmos DB. Було розроблено три програмних реалізації, по одній для кожного досліджуваного програмного підходу та проведені експериментальні дослідження. На основі результатів проведених експериментів були сформовані рекомендації використання того чи іншого API.

За результатами роботи були створені тези доповіді на двадцять сьомий міжнародний молодіжний форум «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ В XXI ст.» , а також, підготовлено до подачі наукову статтю «Дослідження методів програмної реалізації Cosmos DB API на платформі .NET» у науковому журналі «Сучасний стан наукових досліджень і технологій в промисловості» (див. додаток Г).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mazurova, O. Research of ACID transaction implementation methods for distributed databases using replication technology / Mazurova, O., Naboka, A., Shirokopetleva, M. Innovative technologies and scientific solutions for industries, (2 (16), pp. 19-31. Doi: 10.30837/ITSSI.2021.16.019 (дата звернення 23.01.2023).

2. Analyzing and Comparison of NoSQL DBMS Kuzochkina, A., Shirokopetleva, M., Dudar, Z. 2018 International Scientific-Practical Conference on Problems of Infocommunications Science and Technology, PIC S and T 2018 - Proceedings, 2019, стр. 560–564, 8632133 DOI 10.1109/INFOCOMMST.2018.8632133 (дата звернення 23.01.2023).

3. Advantages of NoSQL Databases. URL: https://www.mongodb.com/nosql-explained/advantages#:~:text=NoSQL%20databases%20have%20become%20popular,the%20structure%20of%20the%20data (дата звернення: 24.01.2023).

4. DB-Engines is an initiative to collect and present information on database management systems (DBMS). URL: https://db-engines.com/en/ (дата звернення: 25.01.2023).

5. Azure Cosmos DB pricing. URL: https://azure.microsoft.com/en-us/pricing/details/cosmos-db/autoscale-provisioned/ (дата звернення: 26.01.2023).

6. Amazon DynamoDB pricing. URL: https://aws.amazon.com/dynamodb/pricing/ (дата звернення: 26.01.2023).

7. MongoDB Pricing. URL: https://www.mongodb.com/pricing (дата звернення: 26.01.2023).

8. Cloud Bigtable pricing. URL: https://cloud.google.com/bigtable/pricing (дата звернення: 26.01.2023).

9. Firestore in Datastore mode pricing. URL: https://cloud.google.com/datastore/pricing#firestore-in-datastore-mode-pricing (дата звернення: 26.01.2023).

10. NoSQL databases comparison: Cosmos DB vs DynamoDB vs Cloud Datastore and Bigtable. URL: https://acloudguru.com/blog/engineering/comparing-cloud-nosql-databases-dynamodb-vs-cosmos-db-vs-cloud-datastore-and-bigtable (дата звернення: 26.01.2023).

11. What is REST - REST API Tutorial. URL: https://restfulapi.net/ (дата звернення: 27.01.2023).

12. Welcome to Azure Cosmos DB. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/introduction (дата звернення: 13.02.2023).

13. Common Azure Cosmos DB use cases. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/use-cases (дата звернення: 14.02.2023).

14. SQL vs NoSQL Databases: What's The Difference? URL: https://www.bmc.com/blogs/sql-vs-nosql/ (дата звернення: 15.02.2023).

15. Swagger: API Documentation & Design Tools for Teams. URL: https://swagger.io/ (дата звернення: 16.02.2023).

16. Quickstart: Create an Azure Cosmos DB account, database, container, and items from the Azure portal. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/quickstart-portal (дата звернення: 17.02.2023).

17. Introduction to provisioned throughput in Azure Cosmos DB. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/set-throughput (дата звернення: 18.02.2023).

18. Limit the total throughput provisioned on your Azure Cosmos DB account. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/limit-total-account-throughput (дата звернення: 19.02.2023).

19. Request Units in Azure Cosmos DB. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/request-units (дата звернення: 20.02.2023).

20. Quickstart: Azure Cosmos DB for NoSQL client library for .NET. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/quickstart-dotnet?tabs=azure-portal%2Cwindows%2Cpasswordless%2Csign-in-azure-cli (дата звернення: 21.02.2023).

21. CosmosClient Class. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.azure.cosmos.cosmosclient?view=azure-dotnet (дата звернення: 21.02.2023).

22. JsonPropertyNameAttribute Class. URL: https://docs.microsoft.com/en-us/dotnet/api/system.text.json.serialization.jsonpropertynameattribute?view=net-7.0 (дата звернення: 24.02.2023).

23. EF Core Azure Cosmos DB Provider. URL: https://learn.microsoft.com/en-us/ef/core/providers/cosmos/?tabs=dotnet-core-cli (дата звернення: 25.02.2023).

24. Serialization and Model Binding in ASP.NET Web API. URL: https://learn.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/ (дата звернення: 25.02.2023).

25. Quickstart: Azure Cosmos DB for MongoDB for .NET with the MongoDB driver. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/mongodb/quickstart-dotnet?tabs=azure-cli%2Cwindows (дата звернення: 26.02.2023).

26. Create a web API with ASP.NET Core and MongoDB. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-7.0&tabs=visual-studio (дата звернення: 27.02.2023).