МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ВИКОРИСТАННЯ ХМАРНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗПОДІЛЕНИХ СИСТЕМ

Спеціальність 122 «Комп'ютерні наук»

122 – КРБ – 401.2010122

Виконав студент 4–го курсу, групи 401
_____ А. В. Сметаненко
«19» червня 2024 р.

Керівник: канд. фіз–мат. наук, доцент
_____І. В. Кулаковська
«19» червня 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____2024 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студенту групи 401 факультету комп'ютерних наук Сметаненку Артему Віталійовичу.

1. Тема кваліфікаційної роботи «Використання хмарних технологій для розподілених систем».

Керівник роботи Кулаковська Інесса Василівна, канд. фіз–мат. наук, доцент.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «28» грудня 2023 р. № 271

2. Строк представлення кваліфікаційної роботи студентом «19» червня 2024 р.

3. Вхідні (початкові) дані до роботи: що являє собою розподілена система; розподілені системи в сучасному ІТ; інформація про обрані хмарні сервіси та їх технічні характеристики; вимоги до надійності, масштабованості та безпеки розподіленої системи; специфікації серверів і мережевих налаштувань для реалізації проєкту; аналіз технологічних рішень, включаючи мови програмування та фреймворки.

Очікуваний результат: створена розподілена система із серверів на основі хмарних сервісів, що забезпечує високий рівень надійності, масштабованості та безпеки.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

- сутність розподіленої системи та її роль в сучасному ІТ;
- аналіз сучасних розподілених систем у сфері ІТ;
- інформація про обрані хмарні сервіси та їх технічні характеристики;
- вимоги до надійності, масштабованості та безпеки розподіленої системи;
- специфікації серверів і мережевих налаштувань для реалізації проєкту;
- аналіз технологічних рішень, включаючи мови програмування та фреймворки.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: «Розробка заходів з поліпшення умов праці»

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алексєєва А. О., доцент кафедри екології	

Керівник роботи __канд. фіз–мат. наук, доцент Кулаковська І. В._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання ___Сметаненко А. В._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «10» січня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Використання хмарних технологій для розподілених систем

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників КРБ	10.11.2023	15.11.2023	Виконано
2	Отримання завдання на виконання КРБ	10.01.2024	15.01.2024	Виконано
3	Складання календарного плану роботи на весь період виконання КРБ	16.01.2024	30.01.2024	Виконано
4	Отримання завдання на переддипломну практику	15.04.2024	29.04.2024	Виконано
5	Проходження переддипломної практики, збір та аналіз матеріалів до КРБ	29.04.2024	11.05.2024	Виконано
6	Розробка звіту з переддипломної практики	12.05.2024	15.05.2024	Виконано
7	Виконання КРБ: аналіз розподілених систем, хмарних сервісів та реалізація практичної частини	13.05.2024	22.06.2024	Виконано
8	Перший попередній захист КРБ на засіданні комісії кафедри	27.05.2024	27.05.2024	Виконано
9	Доробка та остаточне оформлення КРБ	28.05.2024	09.06.2024	Виконано
10	Другий попередній захист КРБ на засіданні комісії кафедри	10.06.2024	10.06.2024	Виконано
11	Подання КРБ рецензенту	13.06.2024	13.06.2024	Виконано
11	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	17.06.2024	21.06.2024	Виконано
12	Захист БКР перед екзаменаційною комісією (ЕК)	24.06.2024	28.06.2024	Виконано

Розробив студент _Сметаненко А. В._____ _____
(прізвище, ім'я, по батькові студента) (*підпис*)

Керівник роботи __канд. фіз–мат. наук, доцент Кулаковська І. В. _____
(*посада, прізвище, ім'я, по батькові)* *(підпис)*
«_29_» _____01_____ 2024 р.

АНОТАЦІЯ

кваліфікаційної роботи студента групи 401 ЧНУ ім. Петра Могили

Сметаненка Артема Віталійовича

Тема: «Використання хмарних технологій для розподілених систем»

Об’єктом роботи є процеси проектування та впровадження розподіленої системи на основі хмарних сервісів.

Предметом роботи є методи та технології, які використовуються для створення розподіленої системи на основі хмарних сервісів, що забезпечують надійність, масштабованість та безпеку.

Мета кваліфікаційної роботи – розробка розподіленої системи на основі хмарних сервісів, яка враховуватиме переваги та недоліки існуючих рішень на ринку та відповідатиме потребам користувачів, забезпечуючи надійність, масштабованість та безпеку.

Робота складається з фахового розділу і спеціальної частини з охорони праці. Пояснювальна записка складається зі вступу, трьох розділів та висновків.

У першому розділі було проведено аналіз сучасних розподілених систем та їх роль в ІТ.

У другому розділі проведено аналіз існуючий хмарних технологій та вибір хмарного провайдера для реалізації практичної частини.

У третьому розділі було змодельовано розподільну систему з використанням Amazon Web Services (AWS), Node.js, Bun, TypeScript, PostgreSQL, Fastify, Elysiajs, Drizzle ORM, Docker, Makefile, AWS CLI, JWT, Swagger. В результаті розроблено розподільну систему з трьома мікросервісами для книжкового додатку до складу якої входить сервіс авторизації та аутентифікації, сервіс менеджменту користувачів, сервіс менеджменту книжок.

Бакалаврська кваліфікаційна робота містить 145 сторінок, 24 рисунки, 4 таблиць, 32 використаних джерел та 8 додатків.

Ключові слова: розподільна система, AWS, Node.js, Bun, TypeScipt, Fastify, Elysiajs, Docker, Makefile, JWT.

ABSTRACT

for bachelor's qualification work of a student of group 401 of Petro Mohyla Black Sea Petro Mohyla Black Sea National University

Smetanenko Artem

The object of the research is the processes of designing and implementing a distributed system based on cloud services.

The subject of the research is the methods and technologies used to create a distributed system based on cloud services that ensure reliability, scalability, and security.

Purpose of the qualification work is to develop a distributed system based on cloud services, which takes into account the advantages and disadvantages of existing solutions on the market and meets the needs of users, ensuring reliability, scalability, and security.

The work consists of a professional section and a special part on labor protection. The explanatory note consists of an introduction, three chapters, and conclusions.

In the first chapter, an analysis of modern distributed systems and their role in IT was conducted.

The second chapter provides an analysis of existing cloud technologies and the selection of a cloud provider for the implementation of the practical part.

The third chapter models a distributed system using Amazon Web Services (AWS), Node.js, Bun, TypeScript, PostgreSQL, Fastify, Elysiajs, Drizzle ORM, Docker, Makefile, AWS CLI, JWT, and Swagger. As a result, a distributed system with three microservices for a book application was developed, including an authentication and authorization service, a user management service, and a book management service.

The thesis contains 145 pages, 24 figures, 4 tables, 32 used sources and 8 appendices.

Keywords: distributed system, AWS, Node.js, Bun, TypeScript, Fastify, Elysiajs, Docker, Makefile, JWT.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AWS – Amazon Web Services

API – Application Programming Interface

EC2 – Elastic Compute Cloud

S3 – Simple Storage Service

RDS – Relational Database Service

IoT – Internet of Things

PaaS – Platform as a Service

SaaS – Software as a Service

IaaS – Infrastructure as a Service

CQRS – Command Query Responsibility Segregation

GPU – Graphics Processing Unit

TPU – Tensor Processing Unit

WAN – Wide Area Network

LAN – Local Area Network

GFS – Google File System

SOA – Service–Oriented Architecture

IAM – Identity and Access Management

ECS – Elastic Container Service

HTTP – Hypertext Transfer Protocol

IP – Internet Protocol

IT – Information Technology

JSON – JavaScript Object Notation

JWT – JSON Web Token

ORM – Object–Relational Mapping

ВСТУП

Актуальність теми. У сучасному світі швидкий розвиток технологій диктує необхідність впровадження інноваційних рішень для підвищення ефективності бізнес–процесів. Розподілені системи, створені на основі хмарних сервісів, стають ключовими елементами для забезпечення надійності, масштабованості та безпеки в організаціях. Завдяки хмарним технологіям, компанії можуть гнучко адаптувати свої ресурси до змінних потреб, що значно підвищує їхню конкурентоспроможність і ефективність управління інформаційними потоками.

Особливу актуальність ця тема набула через стрімкий розвиток технологій та зростаючі вимоги до ефективності та гнучкості бізнес–процесів. У сучасному світі компанії все частіше стикаються з необхідністю обробки великих обсягів даних, забезпечення безперебійної роботи та швидкого реагування на зміни ринку. Розподілені системи на основі хмарних сервісів дозволяють вирішувати ці завдання завдяки своїм перевагам у надійності, масштабованості та безпеці.

Хмарні технології забезпечують централізоване управління ресурсами, що дозволяє організаціям оптимізувати витрати на інфраструктуру та підтримку. Крім того, вони надають можливість швидко масштабувати системи у відповідь на зростаючі потреби, що є ключовим фактором для успішного розвитку бізнесу в умовах сучасної конкуренції. Таким чином, впровадження розподілених систем на основі хмарних сервісів стає все більш важливим для підвищення ефективності, надійності та безпеки організацій.

Об'єкт: процеси проектування та впровадження розподіленої системи на основі хмарних сервісів.

Предмет: методи та технології, які використовуються для створення розподіленої системи на основі хмарних сервісів, що забезпечують надійність, масштабованість та безпеку.

Мета: розробка розподіленої системи на основі хмарних сервісів, яка враховуватиме переваги та недоліки існуючих рішень на ринку та відповідатиме потребам користувачів, забезпечуючи надійність, масштабованість та безпеку.

Основні завдання

Для досягнення поставленої мети необхідно вирішити такі завдання:

– визначити ключові поняття щодо розподілених систем і хмарних технологій;

– дослідити сучасний стан розвитку розподілених систем на базі хмарних сервісів;

– провести огляд сучасних технологій для створення розподілених систем і вибрати відповідні для реалізації проекту;

– спроектувати архітектуру системи та її складові частини;

– реалізувати функції системи відповідно до запитів користувачів;

– здійснити тестування системи та виправити виявлені помилки.

Практичне значення: розроблена система менеджменту бібліотеки книг користувача складається з чотирьох сервісів, які працюють у єдиній екосистемі, але реалізовані як мікросервіси. Ця система була успішно розгорнута на AWS, використовуючи відповідні сервіси хмарної платформи. На основі цього проєкту можна дослідити підходи до реалізації розподілених систем. Розроблена система демонструє, як мікросервіси можуть бути інтегровані у єдиний бекенд–сервіс, що функціонує на хмарній платформі AWS. Це забезпечує високу надійність, масштабованість та безпеку, а також дозволяє гнучко адаптуватися до змінних потреб користувачів і організацій.

1 АНАЛІЗ РОЗПОДІЛЕНОЇ СИСТЕМИ ТА ЇЇ РОЛЬ В СУЧАСНОМУ ІТ

1.1 Сутність розподіленої системи

Розподілені системи — це комплекс комп'ютерних пристроїв, які функціонують як єдина цілісна система, незважаючи на фізичну віддаленість один від одного. Основною метою таких систем є узгоджене використання ресурсів та надання користувачам і додаткам можливості доступу до різноманітних сервісів, незалежно від місця їхнього розташування. В умовах швидкого розвитку технологій розподілені системи стають все більш важливими для сучасного інформаційного суспільства[18].

Розподілені системи можуть охоплювати різні типи пристроїв, від персональних комп'ютерів і серверів до мобільних пристроїв та IoT–датчиків. Вони взаємодіють між собою через локальні та глобальні мережі, що дозволяє об'єднувати обчислювальні ресурси та забезпечувати надійність і безперервність роботи. Завдяки цим системам, організації можуть ефективніше використовувати свої ресурси, підвищувати продуктивність та швидше реагувати на зміни ринку.

Розвиток розподілених систем бере свій початок з середини 20–го століття, коли з'явилася необхідність підвищення ефективності обчислень та забезпечення надійності і безперервності роботи обчислювальних систем. Перші комп'ютерні мережі, такі як ARPANET, створена в 1969 році, продемонстрували можливості розподілених обчислень. ARPANET стала прообразом сучасного Інтернету і показала, як різні комп'ютерні системи можуть обмінюватися даними, що стало основою для подальшого розвитку розподілених систем[1].

У 1970–х і 1980–х роках домінуючою моделлю обчислень були мейнфрейми, до яких підключалися термінали. Ця модель мала централізовану структуру: всі обчислення виконувалися на мейнфреймах, а термінали використовувалися для введення та виведення даних. Хоча це ще не були справжні розподілені системи, цей підхід заклав основи для подальшого розвитку клієнт–серверної архітектури.

З початком 1980–х років і до початку 1990–х, зростаючі вимоги до обчислювальної потужності та надійності призвели до розвитку клієнт–серверної архітектури. У цій моделі клієнтські комп'ютери виконували роль інтерфейсів, а сервери забезпечували обчислювальні ресурси та доступ до даних. Ця архітектура дозволила розподілити обчислювальні навантаження між клієнтами та серверами, підвищивши ефективність та масштабованість систем.

З розвитком технологій локальних мереж (Local Area Networks, LAN) у 1980–х роках стало можливим об'єднання комп'ютерів у межах однієї організації або будівлі в єдину мережу. Це сприяло підвищенню продуктивності та дозволило розподіляти ресурси між різними пристроями. З'явилися перші комерційні розподілені системи, такі як Novell NetWare, які забезпечували файловий обмін та доступ до спільних ресурсів.

З початку 1990–х років розвиток Інтернету призвів до появи глобальних мереж (Wide Area Networks, WAN), які об'єднали комп'ютери по всьому світу. Це дало поштовх до розвитку нових моделей розподілених систем, які могли працювати на різних континентах і забезпечувати доступ до даних і сервісів у глобальному масштабі. Виникли такі концепції, як веб–сервіси та сервіс–орієнтована архітектура (SOA).

У 2000–х роках почала розвиватися концепція хмарних обчислень, яка стала революційною в розвитку розподілених систем. Хмарні обчислення забезпечують доступ до обчислювальних ресурсів через Інтернет, дозволяючи користувачам орендувати необхідні потужності та зберігання даних за потребою. Хмарні платформи, такі як Amazon Web Services (AWS), Google Cloud Platform (GCP) та Microsoft Azure, дозволяють легко масштабувати ресурси, підвищуючи надійність та ефективність систем.

У 2010–х роках розвиток Інтернету речей (Internet of Things, IoT) привів до появи нових типів розподілених систем, що об'єднують безліч датчиків, пристроїв та

комп'ютерів. Граничні обчислення (Edge Computing) дозволяють обробляти дані ближче до джерела їх виникнення, знижуючи затримки і підвищуючи ефективність систем. Ці технології забезпечують обробку великих обсягів даних у реальному часі та дозволяють створювати інтелектуальні системи для різних галузей[2].

У сучасному світі розподілені системи є основою багатьох важливих додатків та сервісів, включаючи хмарні обчислення, великі дані, інтернет речей (IoT) та багато інших. Вони дозволяють реалізувати високонадійні, масштабовані та гнучкі рішення, що можуть адаптуватися до змінних потреб користувачів.

Підвищенню надійності сервісів полягає в можливості дублюванні даних і сервісів, що дозволяє системі продовжувати функціонувати навіть у разі відмови окремих компонентів. Це особливо важливо для критичних додатків, де безперервність роботи має вирішальне значення, таких як фінансові послуги, охорона здоров'я та електронна комерція.

Види розподілених систем[2]:

– клієнт–серверні системи: у таких системах є один або кілька серверів, які надають ресурси та сервіси клієнтам. Клієнти відправляють запити до серверів і отримують відповідні відповіді. Така архітектура дозволяє централізовано управляти ресурсами, але має обмеження на масштабованість через можливе перевантаження серверів;

– пірингові системи (Peer–to–Peer): всі учасники мають рівні права та можуть виступати як клієнтами, так і серверами. Це забезпечує високу гнучкість та масштабованість, оскільки кожен новий учасник додає ресурси до системи. Такі системи використовуються для файлообміну та децентралізованих обчислень;

– хмарні обчислення: надають доступ до ресурсів та сервісів через інтернет, використовуючи віддалені сервери. Користувачі можуть орендувати обчислювальні потужності та зберігання даних за потребою. Це дозволяє знизити витрати на IT–

інфраструктуру та забезпечити гнучке масштабування. Приклади хмарних платформ включають AWS, Google Cloud, та Microsoft Azure;

– грід–обчислення: об'єднують ресурси багатьох комп'ютерів для вирішення великомасштабних завдань. Вони використовуються для наукових досліджень, аналізу великих даних та інших обчислювально–інтенсивних задач. Грід–системи дозволяють розподіляти обчислення серед багатьох вузлів, що значно підвищує продуктивність;

– розподілені бази даних: зберігають дані на різних фізичних місцях, але працюють як єдина база даних. Це забезпечує високу доступність даних та стійкість до відмов. Користувачі можуть отримувати доступ до даних з різних локацій без втрати продуктивності.

Розподілені системи кардинально відрізняються від традиційного підходу до обчислень та обробки даних, який базується на централізованих системах. Основна відмінність між цими підходами полягає в архітектурі. У централізованих системах всі обчислювальні ресурси, дані та програмні засоби знаходяться на одному центральному сервері або групі серверів. Користувачі взаємодіють з системою через клієнтські пристрої, які виконують роль терміналів. Натомість у розподіленій системі обчислювальні ресурси та дані розподілені між кількома вузлами, які можуть бути географічно розподіленими. Вузли взаємодіють один з одним через мережу, що дозволяє системі працювати як єдине ціле, незважаючи на фізичну віддаленість компонентів[17].

Ще однією суттєвою відмінністю є надійність і стійкість до відмов. У централізованих системах надійність сильно залежить від центрального сервера. Якщо центральний сервер виходить з ладу, вся система перестає працювати, що може призвести до втрати даних і переривання сервісу. Розподілені системи мають вбудовану стійкість до відмов завдяки дублюванню даних та сервісів на кількох

вузлах. Якщо один вузол виходить з ладу, інші вузли продовжують працювати, забезпечуючи безперебійність сервісу та збереження даних.

Масштабованість є ще однією ключовою відмінністю між цими підходами. У централізованих системах масштабування може бути складним та дорогим, оскільки воно зазвичай вимагає оновлення центрального сервера або додавання нових серверів до централізованої інфраструктури. Розподілені системи легко масштабуються, оскільки нові вузли можна додавати до системи без значних змін в архітектурі. Це дозволяє системі ефективно обробляти зростаючі навантаження та адаптуватися до змінних умов[17].

Продуктивність централізованих систем обмежується можливостями центрального сервера. Високе навантаження може призвести до перевантаження серверу та зниження продуктивності. Розподілені системи можуть досягати високої продуктивності завдяки паралельному виконанню завдань на кількох вузлах. Це дозволяє обробляти великі обсяги даних та виконувати складні обчислення швидше.

Управління даними також має свої відмінності. У централізованих системах дані зберігаються на центральному сервері, що може призвести до проблем із доступністю та затримками у випадку високого навантаження на сервер. Розподілені системи зберігають дані на кількох вузлах, що забезпечує високу доступність та зменшує затримки. Розподілені бази даних дозволяють ефективно управляти даними та забезпечують доступ з різних географічних локацій.

Адміністрування централізованих систем відносно простіше, оскільки всі ресурси та дані зосереджені в одному місці. Однак це може створювати єдину точку відмови та підвищує ризик для безпеки. Адміністрування розподілених систем складніше через необхідність управління кількома вузлами, але це забезпечує кращу безпеку та надійність. Розподілені системи також можуть використовувати автоматизовані засоби моніторингу та управління для спрощення адміністрування.

Гнучкість і адаптивність розподілених систем значно вища порівняно з централізованими. У централізованих системах гнучкість обмежена можливостями центрального сервера, і будь–які зміни можуть вимагати значних зусиль та часу. Розподілені системи більш гнучкі та адаптивні до змін, вони можуть швидко адаптуватися до нових умов та вимог, дозволяючи легко впроваджувати нові технології та підходи.

1.2 Роль в сучасному IT

Розподілені системи відіграють важливу роль у сучасних інформаційних технологіях, забезпечуючи широкий спектр можливостей та переваг для різних галузей. Вони дозволяють створювати високонадійні, масштабовані та гнучкі рішення, що можуть адаптуватися до змінних потреб бізнесу та користувачів[19].

Застосування розподілених систем:

– фінансові послуги: розподілені системи використовуються для обробки транзакцій, управління ризиками та аналізу даних. Банки та фінансові установи застосовують розподілені системи для забезпечення безперебійної роботи своїх сервісів, оскільки відмова однієї компоненти не призводить до зупинки всієї системи. Розподілені системи також допомагають у виявленні та запобіганні шахрайству, забезпечуючи високу надійність та безпеку фінансових операцій;

– охорона здоров'я: розподілені системи використовуються для зберігання та обробки медичних даних, управління електронними медичними записами та проведення дистанційних консультацій. Вони дозволяють лікарям швидко отримувати доступ до медичних записів пацієнтів, підвищуючи якість медичного обслуговування та скорочуючи час на діагностику та лікування. Крім того, розподілені системи забезпечують надійне зберігання великих обсягів медичних даних та їх захист від несанкціонованого доступу;

– електронна комерція: розподілені системи забезпечують управління інвентарем, обробку замовлень та персоналізацію рекомендацій. Вони дозволяють інтернет–магазинам обробляти тисячі замовлень щодня, забезпечуючи безперебійну роботу навіть при високому навантаженні. Розподілені системи також допомагають аналізувати поведінку покупців і пропонувати персоналізовані рекомендації, що підвищує задоволеність клієнтів та збільшує продажі;

– наукові дослідження: розподілені системи використовуються для моделювання, симуляцій та обробки великих обсягів даних. Вони дозволяють науковцям виконувати складні обчислення та аналізувати великі набори даних у реальному часі. Наприклад, у фізиці високих енергій розподілені системи використовуються для аналізу даних, зібраних великими колайдерами, такими як CERN's Large Hadron Collider. У біоінформатиці вони застосовуються для обробки геномних даних та моделювання біологічних процесів.

Багато провідних компаній у різних галузях використовують розподілені системи для покращення своєї ефективності, забезпечення надійності сервісів та обробки великих обсягів даних. Нижче наведений розгляд, як деякі з найбільших світових компаній використовують розподілені системи у своїй діяльності:

– Amazon Web Services (AWS): є одним із найбільших провайдерів хмарних обчислень у світі. AWS використовує розподілені системи для надання своїх послуг, таких як зберігання даних (Amazon S3), обчислювальні ресурси (Amazon EC2), бази даних (Amazon DynamoDB) та аналітика (Amazon Redshift). AWS дозволяє своїм клієнтам швидко масштабувати ресурси відповідно до зростаючих потреб, забезпечуючи високу надійність і доступність. Розподілені системи AWS використовуються у всьому світі для розгортання веб–додатків, обробки великих даних, машинного навчання та багато іншого;

– Google: широко використовує розподілені системи для підтримки своїх численних сервісів, включаючи пошук, рекламу, YouTube та Google Cloud Platform

(GCP). Google File System (GFS) і Bigtable є прикладами розподілених систем, розроблених Google для зберігання та обробки великих обсягів даних. Google використовує розподілені обчислювальні кластери для індексації веб–сторінок та обробки запитів пошуку, забезпечуючи високу швидкість і надійність своїх сервісів. Крім того, Google Cloud Platform надає хмарні послуги для інших компаній, дозволяючи їм використовувати розподілені обчислення для своїх потреб;

– Facebook: використовує розподілені системи для забезпечення надійності та масштабованості своєї соціальної мережі, яка обслуговує мільярди користувачів по всьому світу. Cassandra, розподілена база даних, розроблена Facebook, забезпечує зберігання великих обсягів даних з високою доступністю. Facebook також використовує Hadoop для обробки великих даних, аналізу поведінки користувачів та покращення персоналізованих рекомендацій. Розподілені системи дозволяють Facebook ефективно керувати своїми величезними обсягами даних і забезпечувати безперебійне функціонування платформи;

– Netflix: використовує розподілені системи для надання потокових відеосервісів мільйонам користувачів по всьому світу. Вони використовують AWS для зберігання та розподілу відеоконтенту, що дозволяє Netflix швидко масштабувати свої ресурси відповідно до попиту. Розподілені системи також використовуються для аналізу даних про перегляди, що дозволяє Netflix надавати персоналізовані рекомендації та покращувати якість обслуговування. Завдяки розподіленим системам Netflix може забезпечувати високу якість потокового відео навіть при значних навантаженнях;

– Microsoft Azure: є ще одним провідним провайдером хмарних обчислень, який використовує розподілені системи для надання своїх послуг. Azure надає широкий спектр хмарних рішень, включаючи обчислювальні ресурси, зберігання даних, бази даних та аналітичні інструменти. Використовуючи розподілені системи, Azure забезпечує високу надійність, масштабованість та безпеку своїх сервісів. Крім

того, Microsoft Azure використовує розподілені обчислення для підтримки своїх внутрішніх сервісів, таких як Office 365 та Dynamics 365;

– Uber: використовує розподілені системи для управління своєю платформою райдшерінгу, яка обслуговує мільйони поїздок щодня. Вони використовують розподілені бази даних для зберігання інформації про водіїв, пасажирів та маршрути, що дозволяє забезпечити швидкий і надійний доступ до даних. Крім того, Uber використовує розподілені обчислення для аналізу даних про поїздки, оптимізації маршрутів та покращення сервісу. Завдяки розподіленим системам Uber може ефективно керувати своїми ресурсами та забезпечувати високий рівень обслуговування;

– Spotify: використовує розподілені системи для надання потокових музичних сервісів своїм користувачам. Вони використовують хмарні платформи для зберігання та розподілу музичного контенту, що дозволяє Spotify масштабувати свої ресурси відповідно до попиту. Розподілені системи також використовуються для аналізу даних про прослуховування, що дозволяє Spotify надавати персоналізовані рекомендації та покращувати якість обслуговування. Завдяки розподіленим системам Spotify може забезпечувати високу якість потокової музики навіть при значних навантаженнях. На рис. 1.2 показана схема роботи розподіленої системи у компанії Netflix.

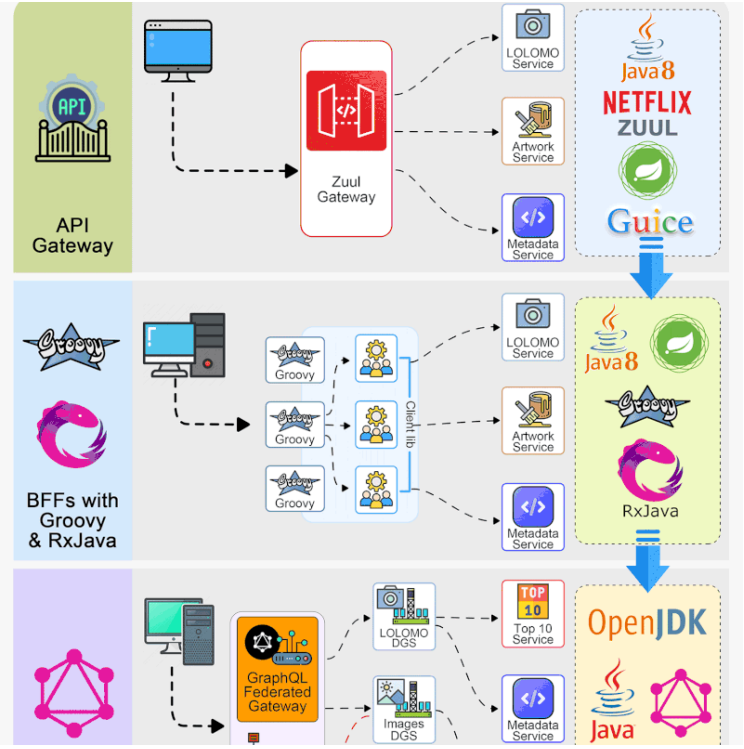

Рисунок 1.2 – Приклад реалізації розподіленої системи у компанії Netflix

1.3 Сучасні реалізації розподілених систем

Сучасні реалізації розподілених систем базуються на низці архітектурних патернів та технологій, які забезпечують надійність, масштабованість, продуктивність та гнучкість. Нижче наведено розгляд деяких з найпоширеніших патернів та технологій, які використовуються для створення розподілених систем[28].

Патерни розподілених систем:

– мікросервісна архітектура: є одним з найпопулярніших патернів для створення розподілених систем. Вона передбачає розбиття додатка на невеликі, незалежні сервіси, кожен з яких виконує окрему бізнес–функцію. Ці сервіси взаємодіють один з одним через добре визначені API, що дозволяє легко масштабувати та оновлювати окремі компоненти без впливу на інші частини системи. Мікросервісна архітектура використовується такими компаніями, як Netflix, Amazon та Uber;

– CQRS (Command Query Responsibility Segregation): передбачає розділення операцій читання та запису даних на різні моделі. Це дозволяє оптимізувати кожну операцію окремо, покращуючи продуктивність та масштабованість системи. CQRS часто використовується разом з Event Sourcing, де всі зміни даних зберігаються як події, що дозволяє легко відстежувати історію змін та відновлювати стан системи;

– Saga: патерн для управління розподіленими транзакціями в мікросервісних архітектурах. Він розбиває транзакції на серію кроків, кожен з яких є окремою транзакцією. Якщо один з кроків не вдається, Saga забезпечує компенсаційні дії для скасування попередніх кроків, забезпечуючи цілісність даних;

– Event–Driven Architecture: передбачає обробку подій у реальному часі. Компоненти системи взаємодіють через події, що дозволяє створювати асинхронні та високопродуктивні системи. Цей патерн часто використовується в поєднанні з брокерами повідомлень, такими як Apache Kafka або RabbitMQ, для забезпечення надійного обміну подіями.

Технології для реалізації розподілених систем[17]:

– контейнеризація та оркестрація: контейнеризація, за допомогою технологій таких як Docker, дозволяє ізолювати додатки та їхні залежності в окремих контейнерах, забезпечуючи портативність та легкість розгортання. Оркестрація контейнерів, зокрема Kubernetes, автоматизує управління, масштабування та розгортання контейнеризованих додатків у розподілених середовищах;

– брокери повідомлень: брокери повідомлень, такі як Apache Kafka, RabbitMQ та ActiveMQ, забезпечують надійний обмін повідомленнями між компонентами розподілених систем. Вони підтримують асинхронну обробку подій, що підвищує продуктивність та гнучкість системи;

– інструменти моніторингу та логування: інструменти моніторингу та логування, такі як Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana) та

Splunk, забезпечують моніторинг стану системи, збір логів та аналіз продуктивності. Вони допомагають виявляти проблеми та оптимізувати роботу розподілених систем;

– розподілені бази даних: розподілені бази даних, такі як Google Spanner, Amazon DynamoDB та Apache Cassandra, забезпечують зберігання даних на кількох вузлах, що підвищує доступність і стійкість до відмов. Вони підтримують горизонтальне масштабування та дозволяють обробляти великі обсяги даних у реальному часі.

На рис. 1.2 – 1.3 наведено приклад реалізацій патернів розподілених систем.

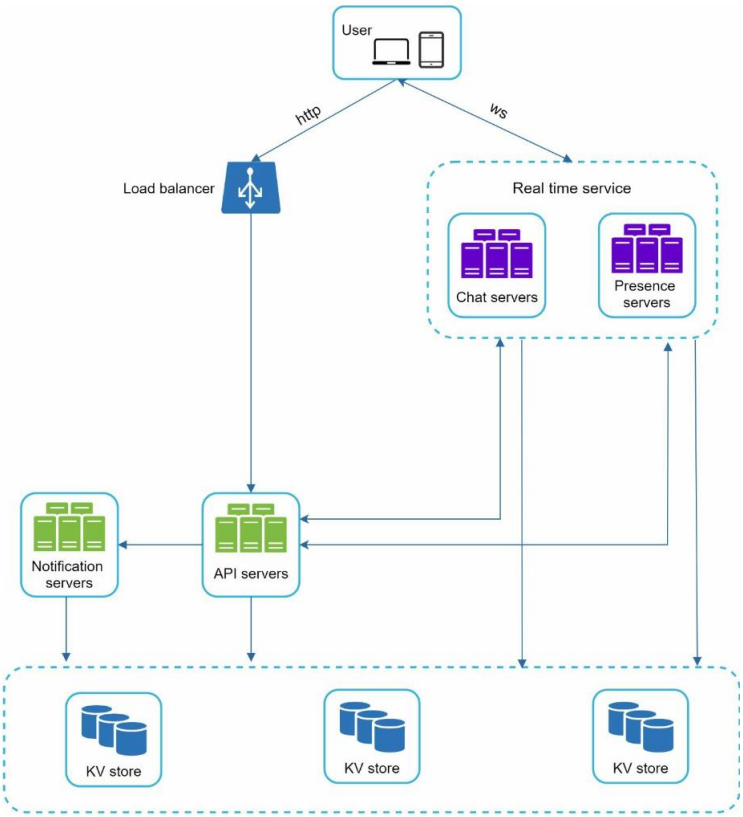

Рисунок 1.2 – Приклад реалізації чат системи використовуючи мікросервіси

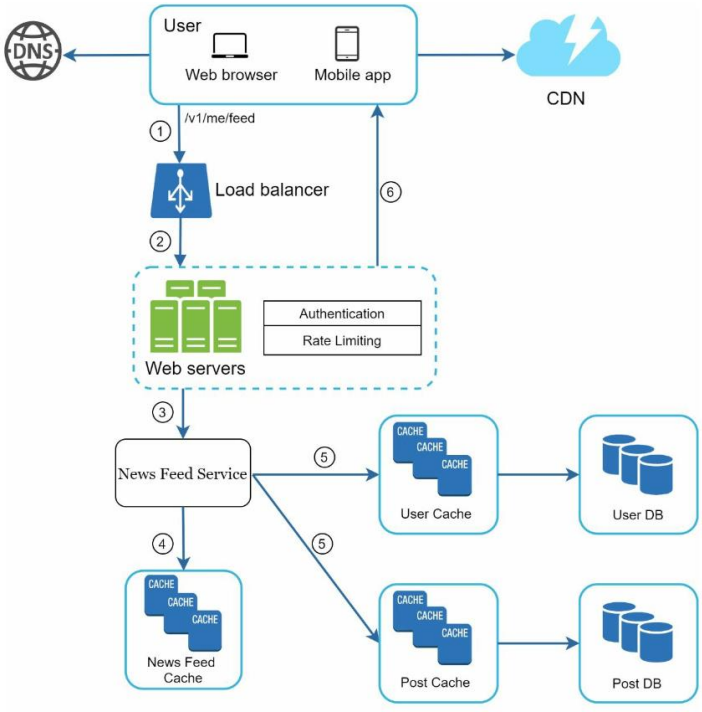

Рисунок 1.3 – Приклад реалізації системи новинного сайту використовуючи мікросервіси

1.4 Майбутнє розподілених систем

Розподілені системи продовжують активно розвиватися, зберігаючи ключову роль у сучасних інформаційних технологіях. Їхнє майбутнє тісно пов'язане з прогресом у галузі штучного інтелекту (ШІ), оскільки ці дві сфери мають багато точок дотику.

Розподілені системи будуть вдосконалюватися у напрямку підвищення масштабованості та продуктивності. Розвиток технологій контейнеризації та оркестрації, таких як Docker та Kubernetes, забезпечить більш ефективне управління ресурсами та швидке масштабування додатків. Нові архітектурні підходи, такі як serverless computing, дозволять створювати ще більш гнучкі та продуктивні системи.

Хмарні платформи продовжуватимуть відігравати важливу роль у розвитку розподілених систем. Інтеграція з хмарними сервісами дозволить компаніям легко

масштабувати свої ресурси та забезпечувати високу доступність і надійність сервісів. Нові сервіси, такі як AWS Outposts, Google Anthos та Azure Arc, забезпечують розширені можливості для гібридних та мультихмарних середовищ, що додатково підвищує гнучкість розподілених систем.

Зростання кількості IoT–пристроїв створює нові виклики та можливості для розподілених систем. Граничні обчислення дозволяють обробляти дані ближче до джерела їх виникнення, знижуючи затримки та підвищуючи ефективність систем. Інтеграція IoT та Edge Computing з розподіленими системами забезпечить нові можливості для автоматизації та аналізу даних у реальному часі.

З розвитком розподілених систем зростає важливість забезпечення безпеки та конфіденційності даних. Нові технології шифрування, блокчейн та інші методи захисту даних будуть інтегровані в розподілені системи для забезпечення їхньої безпеки. Використання Zero Trust архітектури допоможе підвищити захист від кіберзагроз.

Навчання великих моделей ШІ вимагає значних обчислювальних ресурсів, які можуть бути забезпечені тільки за допомогою розподілених систем. Використання кластерів GPU та TPU у хмарних середовищах дозволяє паралельно обробляти великі обсяги даних та прискорювати процес навчання моделей. Розподілене навчання, наприклад, за допомогою технологій таких як TensorFlow та PyTorch, дозволяє розподілити завдання між кількома вузлами, що значно підвищує ефективність та швидкість навчання.

Федеративне навчання є підходом до розподіленого навчання моделей ШІ, при якому дані залишаються на пристроях користувачів, а обчислення відбуваються локально. Це підвищує конфіденційність даних та забезпечує зменшення затримок. Федеративне навчання використовується для розробки персоналізованих моделей ШІ, які враховують специфічні потреби користувачів.

ШІ значною мірою залежить від великих обсягів даних для навчання та аналізу. Розподілені системи забезпечують ефективне зберігання та обробку великих даних у реальному часі. Технології такі як Hadoop та Apache Spark дозволяють розподіляти обчислювальні завдання між кількома вузлами, забезпечуючи високу продуктивність та швидкість обробки.

Інтеграція ШІ з IoT та Edge Computing дозволяє створювати інтелектуальні системи, які можуть аналізувати дані та приймати рішення на місці, без необхідності передачі даних до центрального серверу. Це особливо важливо для додатків, де низька затримка є критичною, наприклад, в автономних транспортних засобах, промисловому Інтернеті речей та розумних містах.

Розподілені системи у поєднанні з ШІ можуть використовуватися для автоматизації та оптимізації бізнес–процесів. Вони дозволяють збирати та аналізувати великі обсяги даних, що допомагає виявляти закономірності та приймати оптимальні рішення. ШІ може використовуватися для автоматизації рутинних завдань, прогнозування попиту, управління запасами та багатьох інших бізнес–процесів.

Розподілені системи мають великі перспективи розвитку, зокрема у поєднанні з технологіями штучного інтелекту. Вони забезпечують необхідні обчислювальні ресурси для навчання моделей ШІ, обробки великих даних та реалізації інтелектуальних систем. З розвитком технологій хмарних обчислень, IoT та Edge Computing, розподілені системи ставатимуть ще більш важливими для створення гнучких, масштабованих та продуктивних рішень у різних галузях.

Висновки до розділу 1

У розділі було розглянуто сутність та основні характеристики розподілених систем, а також їх роль у сучасному ІТ–середовищі. Аналіз показав, що розподілені системи відіграють ключову роль у забезпеченні високої доступності, масштабованості та надійності додатків. Вони дозволяють організаціям ефективно

обробляти великі обсяги даних, забезпечувати безперебійну роботу сервісів та швидко реагувати на зміни ринкових умов. Розподілені системи стали невід'ємною частиною сучасного IT завдяки своїм перевагам, таким як гнучкість, можливість швидкої адаптації до змінних вимог бізнесу, масштабованість, легкість збільшення або зменшення ресурсів відповідно до поточних потреб, та надійність, підвищення стійкості до відмов завдяки розподілу навантаження та дублюванню даних.

Використання хмарних технологій, таких як AWS, дозволяє значно спростити розгортання та управління розподіленими системами. Хмарні провайдери надають широкий спектр сервісів, які підтримують різні аспекти розподілених систем, від обчислювальних ресурсів до інструментів для моніторингу та безпеки. Таким чином, розподілені системи, підтримувані хмарними технологіями, є важливим інструментом для сучасних організацій, що прагнуть підвищити ефективність своїх IT–інфраструктур, забезпечити надійність своїх сервісів та зберегти конкурентоспроможність на ринку.

2 ХМАРНІ ТЕХНОЛОГІЇ

2.1 Сутність хмарних технологій

Хмарні технології є одним із найважливіших нововведень у сучасних інформаційних технологіях, які значно змінили підходи до зберігання, обробки та доступу до даних. Вони забезпечують новий рівень гнучкості, масштабованості та економічної ефективності для бізнесу та кінцевих користувачів[3][2].

Хмарні обчислення (cloud computing), представляють модель надання обчислювальних ресурсів, таких як обчислювальна потужність, зберігання даних та програмне забезпечення, як послуг через Інтернет. Користувачі можуть орендувати ці ресурси на вимогу, без необхідності купувати та обслуговувати фізичну IT–інфраструктуру.

Хмарні технології мають багату історію, яка почалася з розвитку комп'ютерних мереж і розподілених обчислень.

Етапи розвитку:

– 1950–1960–ті роки: початки обчислювальних мереж. Перші ідеї, що лягли в основу хмарних обчислень, з'явилися у 1950–1960–х роках. В цей період обчислювальні центри використовували великі мейнфрейми, до яких підключалися користувачі через термінали. Ця модель забезпечувала централізоване обчислення, але була дуже дорогою і складною в управлінні;

– 1960–ті роки: концепція часу поділу задач. У 1960–х роках була розроблена концепція часу поділу (time–sharing), яка дозволяла багатьом користувачам одночасно використовувати обчислювальні ресурси одного мейнфрейму. Ця ідея стала ключовою для розвитку майбутніх хмарних технологій, оскільки дозволяла ефективніше використовувати обчислювальні потужності;

– 1970–ті роки: розвиток локальних мереж. У 1970–х роках почали активно розвиватися локальні мережі (LAN), що дозволило об'єднувати комп'ютери в межах

однієї організації для спільного використання ресурсів. В цей час з'явилися перші файлові сервери та мережеві операційні системи, які забезпечили основу для розподілених обчислень;

– 1980–ті роки: клієнт–серверна архітектура. У 1980–х роках набула популярності клієнт–серверна архітектура, яка дозволяла розподіляти обчислювальні задачі між клієнтами та серверами. Ця модель забезпечувала більшу гнучкість та масштабованість порівняно з мейнфреймами. З'явилися такі системи, як Novell NetWare та Microsoft Windows NT, які стали основою для розвитку корпоративних мереж;

– 1990–ті роки: виникнення Інтернету та веб–сервісів. У 1990–х роках Інтернет став масово доступним, що призвело до появи нових моделей обчислень та комунікацій. Веб–сервіси та сервіс–орієнтована архітектура (SOA) стали основою для створення розподілених додатків, які могли взаємодіяти через Інтернет. В цей період з'явилися перші компанії, що надавали хостинг та інфраструктурні послуги через Інтернет;

– початок 2000–х: поява терміну "хмарні обчислення". Термін "хмарні обчислення" (cloud computing) почав активно використовуватися на початку 2000–х років. У 2006 році Amazon запустила свою платформу Amazon Web Services (AWS), яка запропонувала новий підхід до оренди обчислювальних ресурсів через Інтернет. Першим сервісом AWS став Amazon S3 (Simple Storage Service), який дозволяв користувачам зберігати та отримувати дані через Інтернет. Незабаром після цього був запущений сервіс Amazon EC2 (Elastic Compute Cloud), який дозволяв орендувати віртуальні машини для виконання обчислювальних задач;

– 2008 рік: вихід Google та Microsoft на ринок хмарних обчислень. У 2008 році Google представила свою платформу Google App Engine, яка надавала розробникам можливість розгортати та масштабувати веб–додатки на інфраструктурі Google. У цьому ж році Microsoft анонсувала свою платформу Windows Azure (нині Microsoft

Azure), яка запропонувала широкий спектр хмарних сервісів для зберігання, обробки та аналізу даних;

– 2010–ті роки: швидке зростання та розвиток хмарних технологій. У 2010–ті роки хмарні технології продовжували активно розвиватися та набувати популярності у різних галузях. Багато компаній почали переходити на хмарні платформи для зберігання та обробки даних, що дозволило їм знижувати витрати на IT–інфраструктуру та підвищувати ефективність роботи. З'явилися нові моделі послуг, такі як Platform as a Service (PaaS) та Software as a Service (SaaS), які дозволяли розробникам та кінцевим користувачам використовувати хмарні ресурси для створення та використання програмного забезпечення.

2.2 Послуги та переваги хмарних технологій для бізнес задач

Хмарні технології забезпечують доступ до обчислювальних ресурсів, зберігання даних та програмного забезпечення через Інтернет. Вони надають організаціям та кінцевим користувачам можливість використовувати потужні IT–інфраструктури без необхідності власного фізичного обладнання[3].

Моделі хмарних послуг:

– IaaS (Infrastructure as a Service): надає користувачам базову інфраструктуру, таку як віртуальні машини, зберігання даних, мережеві ресурси та обчислювальні потужності. Користувачі можуть орендувати ці ресурси та керувати ними через веб–інтерфейси або API, що дозволяє організаціям швидко масштабувати свої обчислювальні потужності відповідно до потреб. IaaS забезпечує високу гнучкість, оскільки користувачі можуть налаштовувати та керувати своїм середовищем відповідно до специфічних вимог;

– PaaS (Platform as a Service): надає платформу для розробки, тестування та розгортання додатків. Це включає середовища розробки, бази даних, сервери застосунків та інші сервіси, які необхідні для створення та запуску програмного

забезпечення. PaaS дозволяє розробникам зосередитися на написанні коду, не піклуючись про управління інфраструктурою. Платформи надають всі необхідні інструменти та сервіси для швидкого створення, тестування та розгортання додатків;

– SaaS (Software as a Service): надає готові до використання програмні додатки, які розміщуються та управляються постачальником послуг. Користувачі можуть отримувати доступ до цих додатків через веб–браузер або API, не піклуючись про управління інфраструктурою або платформою. SaaS дозволяє організаціям використовувати складні програмні рішення без необхідності їх встановлення та обслуговування. Це забезпечує високу гнучкість та економічну ефективність, оскільки користувачі платять тільки за фактично використані ресурси.

На рис. 2.1 наведено існуючи моделі хмарних послуг.

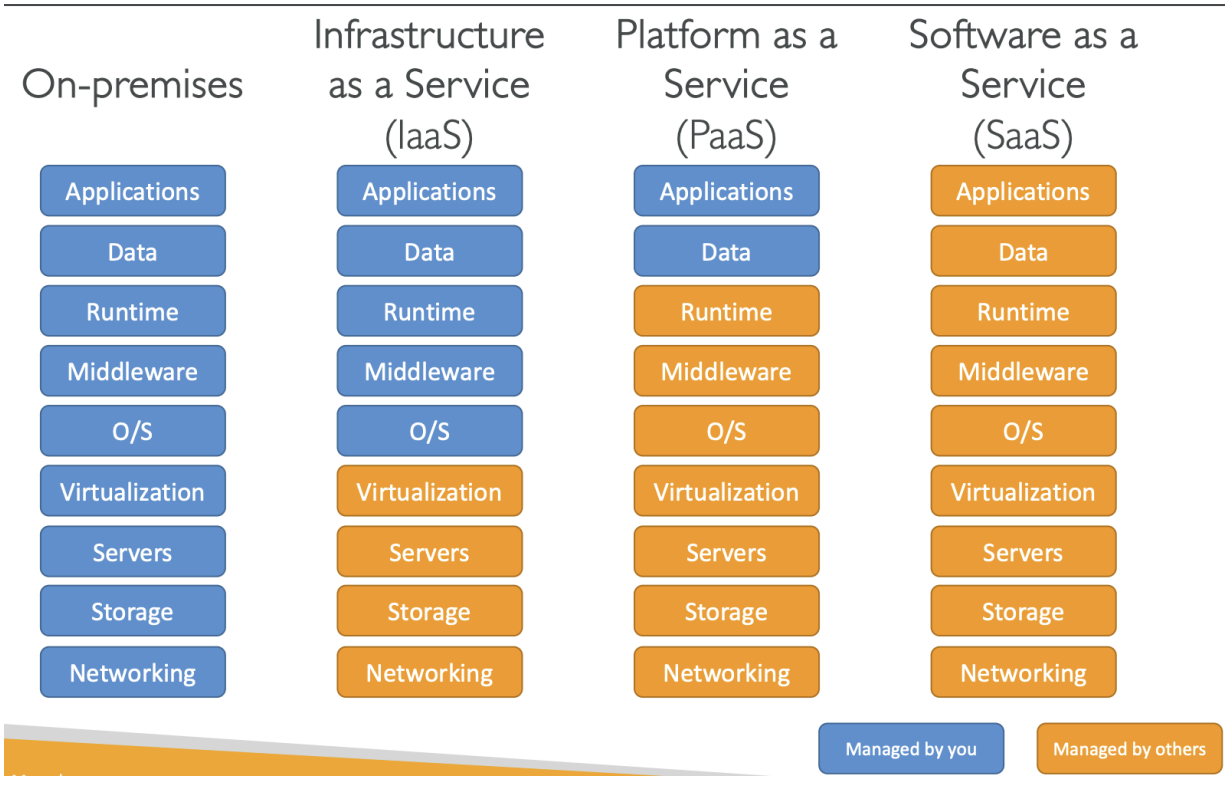

Рисунок 2.1 – Моделі хмарних послуг

Хмарні технології мають значні переваги, які роблять їх популярними серед різних типів організацій, від стартапів до великих підприємств. Ці переваги включають економічну ефективність, гнучкість, масштабованість, надійність та підвищений рівень безпеки[5]:

– економічна ефективність: хмарні технології дозволяють організаціям знизити витрати на IT–інфраструктуру. Користувачі можуть орендувати необхідні обчислювальні ресурси замість того, щоб купувати і підтримувати власне обладнання. Це дозволяє уникнути великих капіталовкладень і перейти на модель операційних витрат, де організації платять тільки за фактично використані ресурси. Крім того, витрати на обслуговування та оновлення обладнання беруть на себе хмарні провайдери, що також знижує загальні витрати;

– гнучкість та мобільність: хмарні сервіси забезпечують доступ до ресурсів з будь–якого місця та з будь–якого пристрою, що має підключення до Інтернету. Це забезпечує високу мобільність для користувачів та дозволяє організаціям легко підтримувати віддалену роботу та глобальні команди. Співробітники можуть працювати з будь–якого куточка світу, отримуючи доступ до необхідних додатків та даних у будь–який час;

– масштабованість: однією з ключових переваг хмарних технологій є можливість швидкого масштабування ресурсів відповідно до змін потреб користувачів. Організації можуть легко збільшувати або зменшувати обчислювальні потужності та зберігання даних без необхідності купувати нове обладнання. Це особливо важливо для бізнесів, які мають сезонні піки навантаження або швидко зростаючі стартапи, яким потрібно швидко масштабувати свої ресурси;

– надійність та доступність: хмарні провайдери забезпечують високий рівень надійності та доступності своїх сервісів. Вони використовують передові технології для резервування даних та автоматичного відновлення у разі відмови обладнання. Завдяки цьому, організації можуть бути впевнені у безперебійному функціонуванні

своїх додатків та збереженні даних. Багато хмарних провайдерів надають гарантії високої доступності (SLA), що забезпечує надійний доступ до сервісів у будь–який час;

– безпека: хмарні провайдери інвестують значні ресурси у забезпечення безпеки своїх платформ. Вони використовують сучасні технології захисту даних, такі як шифрування, автентифікація та контроль доступу, для забезпечення конфіденційності та цілісності даних. Хмарні провайдери також проводять регулярні аудити безпеки та тестування на проникнення для виявлення і усунення вразливостей. Це забезпечує високий рівень захисту даних і знижує ризик кіберзагроз.

Хмарні технології дозволили бізнесу знизити витрати на IT–інфраструктуру, що включає купівлю, обслуговування та оновлення серверів, зберігання даних та мережевих ресурсів. Замість великих капіталовкладень, організації можуть орендувати необхідні ресурси за підпискою, сплачуючи тільки за фактично використані потужності. Це зменшує фінансові ризики та дозволяє спрямовувати більше ресурсів на розвиток основного бізнесу. Хмарні провайдери забезпечують високий рівень надійності та безпеки своїх сервісів. Вони використовують сучасні технології для резервування даних, автоматичного відновлення у разі відмови обладнання та захисту від кіберзагроз. Це забезпечує безперебійну роботу бізнес–додатків та збереження даних. Компанії можуть бути впевнені, що їхні дані захищені та доступні у будь–який час.

2.3 Моделі розгортання хмари

Моделі розгортання хмари — це різні підходи до надання, використання та керування хмарними ресурсами. Кожна модель визначає, як хмарні ресурси надаються користувачам, які вони мають можливості та які обмеження існують[3].

У моделі публічної хмари ресурси надаються стороннім провайдером через Інтернет і доступні для загального використання. Користувачі можуть орендувати обчислювальні потужності, зберігання даних та інші ресурси за потребою.

Приватна хмара надає хмарні ресурси виключно одній організації. Вона може бути розташована на власному апаратному забезпеченні організації або управлятися стороннім провайдером, але доступ до неї мають лише уповноважені користувачі цієї організації.

Гібридна хмара поєднує елементи як приватних, так і публічних хмарних рішень, дозволяючи обмін даними та додатками між ними. Це забезпечує більшу гнучкість та оптимізацію існуючої інфраструктури.

Спільна хмара обслуговує конкретну спільноту користувачів з спільними інтересами або вимогами (наприклад, безпека, відповідність нормативам). Інфраструктура може управлятися однією або кількома організаціями зі спільної спільноти, третьою стороною або їх комбінацією.

Публічна хмара має значні переваги, які роблять її привабливою для багатьох організацій та користувачів. Однією з головних переваг є масштабованість та гнучкість. Користувачі можуть швидко збільшувати або зменшувати обчислювальні ресурси відповідно до своїх потреб без необхідності значних капітальних витрат. Це особливо важливо для бізнесів, які стикаються зі змінними робочими навантаженнями і потребують швидкої адаптації. Публічна хмара є відмінним наочним прикладом для побудови мікросервісів завдяки своїм численним перевагам та можливостям. Мікросервіси – це архітектурний підхід, при якому додаток складається з невеликих, незалежних сервісів, що взаємодіють між собою через чітко визначені інтерфейси. Публічна хмара забезпечує ідеальне середовище для реалізації цієї архітектури.

2.4 Хмарні провайдери

Хмарні провайдери пропонують різноманітні сервіси та ресурси, які дозволяють організаціям і кінцевим користувачам використовувати потужні IT–інфраструктури через Інтернет.

Таблиця 2.1 – Порівняльна таблиця хмарних провайдерів

Параметр	**Amazon Web Services (AWS)**	**Microsoft Azure**	**Google Cloud Platform (GCP)**	**IBM Cloud**	**Oracle Cloud**
Обчислювальні сервіси	Amazon EC2	Azure Virtual Machines	Google Compute Engine	IBM Cloud Virtual Servers	Oracle Cloud Infrastructure (OCI) Compute
Сервіси зберігання	Amazon S3	Azure Blob Storage	Google Cloud Storage	IBM Cloud Object Storage	Oracle Cloud Object Storage
Сервіси баз даних	Amazon RDS, DynamoDB	Azure SQL Database	Google BigQuery, Cloud SQL	IBM Db2	Oracle Autonomous Database
Безсерверні сервіси	AWS Lambda	Azure Functions	Google Cloud Functions	IBM Cloud Functions	Oracle Cloud Functions
Сервіси ШІ/МЛ	Amazon SageMaker	Azure Machine Learning	Google AI Platform	IBM Watson	Oracle AI and Machine Learning
Аналітичні сервіси	Amazon Redshift	Azure Synapse Analytics	Google BigQuery	IBM Analytics	Oracle Analytics Cloud
Гібридна хмара	AWS Outposts	Azure Arc	Google Anthos	IBM Hybrid Cloud	Oracle Cloud at Customer
Модель ціноутворення	Оплата за фактом використання, зарезервовані інстанси	Оплата за фактом використання, зарезервовані інстанси	Оплата за фактом використання, знижки за постійне використання	Оплата за фактом використання, підписка	Оплата за фактом використання, підписка
Глобальна присутність	24 регіони, 76 зони доступності	60+ регіонів	25 регіонів, 73 зони	18 регіонів	30+ регіонів

AWS було обрано для реалізації проекту через його широкі можливості та перевірену надійність. AWS пропонує широкий спектр сервісів, включаючи

обчислювальні ресурси (Amazon EC2), сервіси зберігання даних (Amazon S3), бази даних (Amazon RDS, DynamoDB). Ці сервіси забезпечують високу гнучкість та масштабованість, що дозволяє легко адаптуватися до змінних вимог проекту.

Крім того, AWS пропонує безкоштовний рівень доступу до багатьох своїх сервісів, що дозволяє новим користувачам випробувати можливості платформи без початкових витрат. Це особливо корисно для прототипування та тестування.

Для наочного прикладу, вибір хмарного провайдера не має критичного значення, оскільки основні концепції та методи залишаються схожими незалежно від обраної платформи. Однак, завдяки своїй популярності, потужній інфраструктурі та широкому спектру сервісів, AWS є оптимальним вибором для реалізації практичної частини.

2.5 AWS

Історія створення Amazon Web Services (AWS) є цікавою ілюстрацією того, як компанія Amazon перетворилася з інтернет–магазину книг у одного з найбільших постачальників хмарних обчислень у світі. У середині 1990–х років Amazon, заснована Джеффом Безосом у 1994 році, почала свій шлях як онлайн–книгарня. Протягом наступних років компанія розширювала свій асортимент товарів і вдосконалювала свої внутрішні IT–системи для підтримки зростаючих потреб бізнесу. До початку 2000–х років Amazon вже мала значний досвід у створенні масштабованих IT–систем. Компанія стикалася з численними проблемами масштабування своїх інфраструктур, і для вирішення цих проблем інженери Amazon почали розробляти модульні та автоматизовані системи управління ресурсами. У 2000 році Бенджамін Блек, інженер Amazon, разом зі своїм колегою розробив ідею про те, що внутрішні IT–ресурси Amazon можуть бути використані для надання зовнішніх послуг. Вони підготували документ, у якому описували концепцію масштабованої хмарної інфраструктури. Ця ідея отримала підтримку від вищого керівництва компанії,

включаючи Безоса. У 2003 році Amazon офіційно розпочала роботу над створенням AWS. Основною ідеєю було надання інфраструктури як послуги (IaaS), що дозволило б компаніям орендувати обчислювальні потужності, зберігання даних та інші IT–ресурси через Інтернет. Перший сервіс AWS, Amazon S3 (Simple Storage Service), був запущений у березні 2006 року. S3 надавав клієнтам масштабоване і надійне зберігання даних. Відразу після цього, у серпні 2006 року, Amazon запустила Amazon EC2 (Elastic Compute Cloud), який дозволяв користувачам орендувати віртуальні сервери для запуску своїх додатків. Після успішного запуску S3 та EC2, AWS швидко почала розширювати свій асортимент послуг. У 2007 році було запущено Amazon Simple Queue Service (SQS) і Amazon SimpleDB. В наступні роки AWS додавала нові сервіси, такі як Amazon RDS (Relational Database Service) у 2009 році, Amazon VPC (Virtual Private Cloud) у 2009 році та багато інших. У 2010 році Amazon представила свою Marketplace, де клієнти могли знаходити і використовувати програмні рішення від третіх сторін. Це значно розширило екосистему AWS. До 2014 року AWS стала провідним гравцем на ринку хмарних обчислень, забезпечуючи широкий спектр сервісів, включаючи обчислення, зберігання, бази даних, аналітику, машинне навчання, штучний інтелект, Інтернет речей (IoT), мобільні сервіси, безпеку та корпоративні додатки. AWS продовжувала розширювати свою глобальну інфраструктуру, відкриваючи нові центри обробки даних і регіони по всьому світу. На рис. 2.2 наведено історію становлення та розвитку платформи AWS.

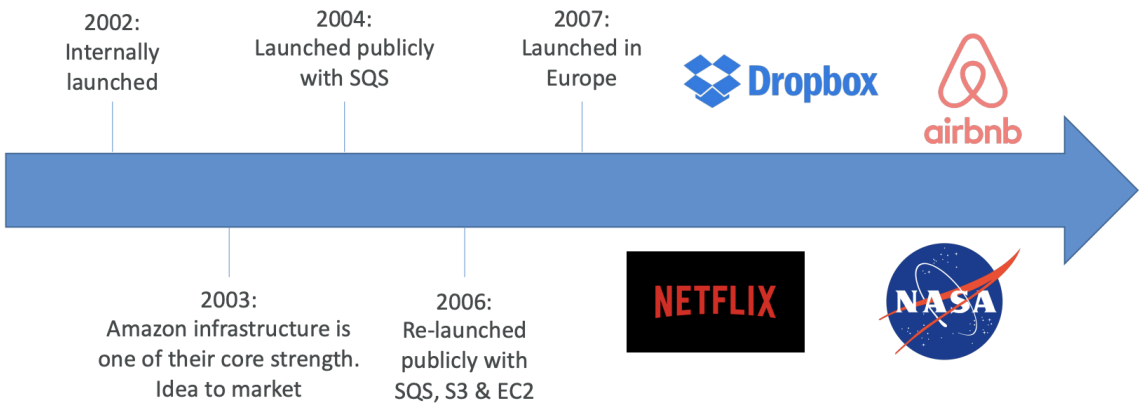

Рисунок 2.2 – Історія AWS

Висновки до розділу 2

У розділі було досліджено сутність та основні характеристики хмарних технологій, а також їх переваги для бізнес–завдань. Аналіз показав, що хмарні технології забезпечують централізоване управління ресурсами, що дозволяє оптимізувати витрати на інфраструктуру та підтримку. Вони надають можливість швидко масштабувати системи у відповідь на зростаючі потреби бізнесу, що є ключовим фактором для успішного розвитку в умовах сучасної конкуренції.

Хмарні провайдери, такі як AWS, надають широкий спектр послуг, включаючи обчислювальні ресурси, зберігання даних, бази даних, аналітику, машинне навчання та інструменти для забезпечення безпеки. Це дозволяє організаціям легко інтегрувати різні сервіси та створювати комплексні рішення, що відповідають їхнім специфічним потребам.

Хмарні технології також сприяють підвищенню надійності та безпеки інформаційних систем. Вони забезпечують високий рівень відмовостійкості завдяки географічно розподіленій інфраструктурі, що мінімізує ризики простою та втрати даних. Крім того, хмарні провайдери пропонують потужні засоби для моніторингу та управління безпекою, що допомагає захищати дані від несанкціонованого доступу.

Таким чином, хмарні технології є невід'ємною складовою сучасного IT–середовища, що дозволяє організаціям підвищувати ефективність своїх бізнес–процесів, забезпечувати надійність та безпеку інформаційних систем, а також швидко адаптуватися до змінних вимог ринку.

3 РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ СИСТЕМИ НА БАЗІ ХМАРНОГО ПРОВАЙДЕРА AWS

3.1 Вибір технологічного стеку та архітектури

Ціллю було створити систему мікросервісів, яка складається з трьох серверів, що взаємодіють по різним протоколам та написані на різних фреймворках та середовищах виконання. Система повинна бути побудована на основі AWS.

Таблиця 3.1 – Опис середовищ запуску JavaScript та їх порівняння

Технологія	**Версія**	**Опис**	**Використання**	**Переваги**	**Вимоги до системи**	**Інструменти**
Node.js	Остання стабільна версія	Серверне середовище, яке дозволяє запускати JavaScript поза браузером.	Створення серверних додатків, API, мікросервісів та реальних додатків	Асинхронність, велика екосистема модулів (npm), висока продуктивність	Підтримується на всіх основних ОС	npm (Node Package Manager), npx, nvm (Node Version Manager)
Bun	Остання стабільна версія	Швидкий JavaScript runtime з вбудованим тестуванням, пакуванням та API для Web.	Альтернатива Node.js для швидкого виконання JavaScript коду на сервері	Висока швидкість, вбудовані інструменти, легкість налаштування	Підтримується на всіх основних ОС	Вбудовані тестування, пакування, підтримка TypeScript з коробки

Node.js було обрано через його асинхронну природу, яка дозволяє ефективно обробляти велику кількість одночасних запитів без блокування потоку. Крім того, велика екосистема модулів і бібліотек, доступних через npm (Node Package Manager),

значно пришвидшує розробку. Node.js використовує V8 JavaScript engine від Google, що забезпечує високу продуктивність і швидкість виконання коду. Використання JavaScript як на клієнті, так і на сервері дозволяє розробникам працювати з єдиною мовою, що спрощує процес розробки та підтримки додатків. Node.js чудово підходить для створення мікросервісної архітектури, що робить додатки більш масштабованими і гнучкими[6].

Bun було обрано завдяки його високій швидкості виконання JavaScript коду, що робить його привабливим для високонавантажених серверних додатків. Bun включає в себе вбудовані інструменти для тестування та пакування, що спрощує розробку і підтримку проекту. Мінімальні налаштування дозволяють швидко почати роботу і сфокусуватись на розробці. Bun має вбудовану підтримку TypeScript, що дозволяє використовувати сучасні можливості JavaScript та забезпечити високу якість коду[29].

TypeScript було обрано через його статичну типізацію, яка дозволяє виявляти помилки ще на етапі компіляції, що значно знижує кількість багів у кінцевому продукті. TypeScript інтегрується з багатьма популярними IDE, надаючи інструменти для автозаповнення, рефакторингу та налагодження. TypeScript підтримує нові стандарти JavaScript, дозволяючи використовувати останні можливості мови. Завдяки статичній типізації, великі проекти стають легше підтримувати і розвивати[7].

Docker було обрано завдяки його здатності забезпечувати ізоляцію середовища, що робить додатки портативними і незалежними від середовища. Контейнери дозволяють легко масштабувати додатки, розподіляючи їх по різним серверам. Docker дозволяє швидко розгортати та оновлювати додатки, інтегруючись з багатьма інструментами CI/CD, що спрощує автоматизацію процесів розробки та розгортання[10].

PostgreSQL було обрано через його надійність та стабільність роботи. База даних підтримує різноманітні розширення, такі як PostGIS для геопросторових даних, і повністю відповідає стандартам SQL, що забезпечує високу сумісність з іншими

системами. PostgreSQL дозволяє виконувати складні SQL запити та має потужний механізм індексації, що підвищує продуктивність запитів[8].

Makefile було обрано для автоматизації рутинних завдань, таких як компіляція, тестування та деплоймент. Синтаксис Makefile досить простий, що дозволяє швидко створювати скрипти для автоматизації. Makefile підтримується на всіх основних операційних системах і автоматично відстежує залежності між файлами, що дозволяє уникати повторної компіляції незмінних частин проекту[11].

Hurl було обрано для тестування HTTP API завдяки його простому синтаксису і мінімальним налаштуванням, що дозволяє швидко розпочати роботу. Hurl дозволяє легко виконувати HTTP запити і перевіряти відповіді, що робить його ідеальним для тестування API. Інструмент підтримує автоматизацію складних сценаріїв запитів, що спрощує інтеграційне тестування, і дозволяє виконувати складні перевірки відповідей, що підвищує якість тестування[30]. Hurl підтримує різні типи запитів, такі як GET, POST, PUT, DELETE та інші. Можна задавати заголовки, параметри, тіла запитів та інші налаштування. Крім того, Hurl дозволяє перевіряти статус коди, заголовки та тіло відповіді на відповідність очікуваним значенням. Також можна створювати складні сценарії тестування з умовами, циклами та змінними. Тому це надає можливості якісного та всеобічного тестування проєктів, які потребують автоматизованого тестування.

Для реалізації проекту було обрано підхід монорепозиторію. Монорепозиторій дозволяє зберігати всі сервіси в одному репозиторії, що забезпечує ряд переваг у розробці, підтримці та розгортанні.

Переваги використання монорепозиторію включають спільне використання коду між різними мікросервісами. Наприклад, загальні утиліти, моделі даних або конфігураційні файли можуть бути розміщені в спільних директоріях і використовуватись усіма сервісами. Єдине середовище розробки дозволяє уникнути необхідності кожному мікросервісу мати окремий репозиторій з окремими

налаштуваннями, що полегшує налаштування інструментів для збірки, тестування та розгортання, а також спрощує управління залежностями.

Таблиця 3.2 – Опис фреймворків та бібліотек, використаних у проекті

Параметр	**Fastify**	**Elysiajs**	**Drizzle ORM**	**Zod**	**Ts Pattern**	**Fast–jwt**
Опис	Високопродуктивний веб–фреймворк для Node.js	Легкий веб–фреймворк, оптимізований для швидкості та простоти	TypeScript ORM, орієнтований на простоту та ефективність	Бібліотека для перевірки та валідації даних на основі TypeScript	Бібліотека для роботи з шаблонами та патернами у TypeScript, підтримує алгебраїчні структури та рекурсивні шаблони	Легка і швидка бібліотека для роботи з JWT (JSON Web Tokens) у Node.js
Основне призначення	Створення швидких та масштабованих веб–додатків та API	Створення високопродуктивних веб–додатків з мінімальним кодом	Робота з базами даних з використанням TypeScript	Валідація та парсинг даних, створення схем	Супровід складних шаблонів та патернів у TypeScript, забезпечення типобезпеки	Генерація та перевірка JWT токенів, забезпечення безпеки додатків
Особливості	Плагінна архітектура, підтримка багатьох плагінів, асинхронність, підтримка TypeScript	Підтримка TypeScript з коробки, легкість у розширенні, побудований для сучасних веб–додатків	Підтримка TypeScript, зручний DSL для роботи з SQL	Підтримка складних схем, компіляція схем у швидкий код	Підтримка алгебраїчних структур, рекурсивні шаблони, висока типобезпека	Підтримка різних алгоритмів шифрування, сумісність з іншими бібліотеками Node.js

3.2 Опис архітектури мікросервісів для побудови API для книжкового додатку

Метою побудови цієї системи було створення розподіленої архітектури, яка складається з трьох окремих мікросервісів, що взаємодіють між собою через різні протоколи. Кожен мікросервіс відповідає за певну частину функціональності системи, що забезпечує гнучкість, масштабованість та надійність.

Перший сервіс, сервіс авторизації, відповідає за аутентифікацію та авторизацію користувачів. Він є приватним, тобто доступний тільки для інших мікросервісів системи і не відкритий для зовнішніх запитів. Цей сервіс побудований на базі фреймворка Fastify та використовує Node.js як середовище виконання. Для роботи з JWT токенами використовується бібліотека Fast–jwt. Комунікація з цим сервісом відбувається через звичайний протокол HTTP. Основні функції сервісу авторизації включають обробку запитів на реєстрацію та вхід користувачів, генерацію та перевірку JWT токенів, а також управління сесіями користувачів. Висока швидкість обробки запитів досягається завдяки асинхронному виконанню та оптимізаціям Fastify. Сервіс забезпечує безпечне зберігання та перевірку токенів, а також легкість інтеграції з іншими сервісами завдяки HTTP протоколу.

Другий сервіс, сервіс менеджменту користувачів, відповідає за управління інформацією про користувачів. Він використовує сервіс авторизації для перевірки аутентифікації користувачів. Цей сервіс побудований на базі фреймворка Elysiajs та використовує Node.js як середовище виконання. Для роботи з базою даних використовується Drizzle ORM, а для валідації даних – бібліотека Zod. Комунікація з сервісом авторизації відбувається через HTTP. Основні функції сервісу включають обробку запитів на створення, редагування та видалення профілів користувачів, валідацію та обробку даних користувачів, а також взаємодію з базою даних для зберігання інформації про користувачів. Висока продуктивність досягається завдяки

легкому та швидкому фреймворку Elysiajs. Сервіс забезпечує надійну валідацію даних за допомогою Zod та інтеграцію з сервісом авторизації для забезпечення безпеки.

Третій сервіс, сервіс менеджменту книжок, відповідає за управління інформацією про книжки. Він використовує сервіс авторизації для перевірки аутентифікації та сервіс менеджменту користувачів для отримання інформації про користувачів. Цей сервіс побудований на базі фреймворка та використовує Bun як середовище виконання. Для роботи з базою даних використовується Drizzle ORM, а для валідації даних – бібліотека Zod. Комунікація з сервісом авторизації та сервісом менеджменту користувачів відбувається через HTTP, а для асинхронної обробки запитів між сервісами використовується Amazon SQS. Основні функції сервісу включають обробку запитів на створення, редагування та видалення інформації про книжки, валідацію та обробку даних про книжки, управління запасами книжок та обробку замовлень. Сервіс забезпечує високу продуктивність та масштабованість завдяки використанню асинхронної обробки, надійну валідацію даних за допомогою Zod та ефективну обробку запитів завдяки використанню Amazon SQS для асинхронної комунікації між сервісами.

Загальна архітектура системи включає сервіс авторизації, який є приватним і забезпечує аутентифікацію та авторизацію користувачів, сервіс менеджменту користувачів, який відповідає за управління інформацією про користувачів і використовує сервіс авторизації для перевірки аутентифікації, та сервіс менеджменту книжок, який відповідає за управління інформацією про книжки і взаємодіє з іншими сервісами через HTTP та Amazon [3][32]. На рис. 3.1 наведено приклад роботи системи мікросервісів.

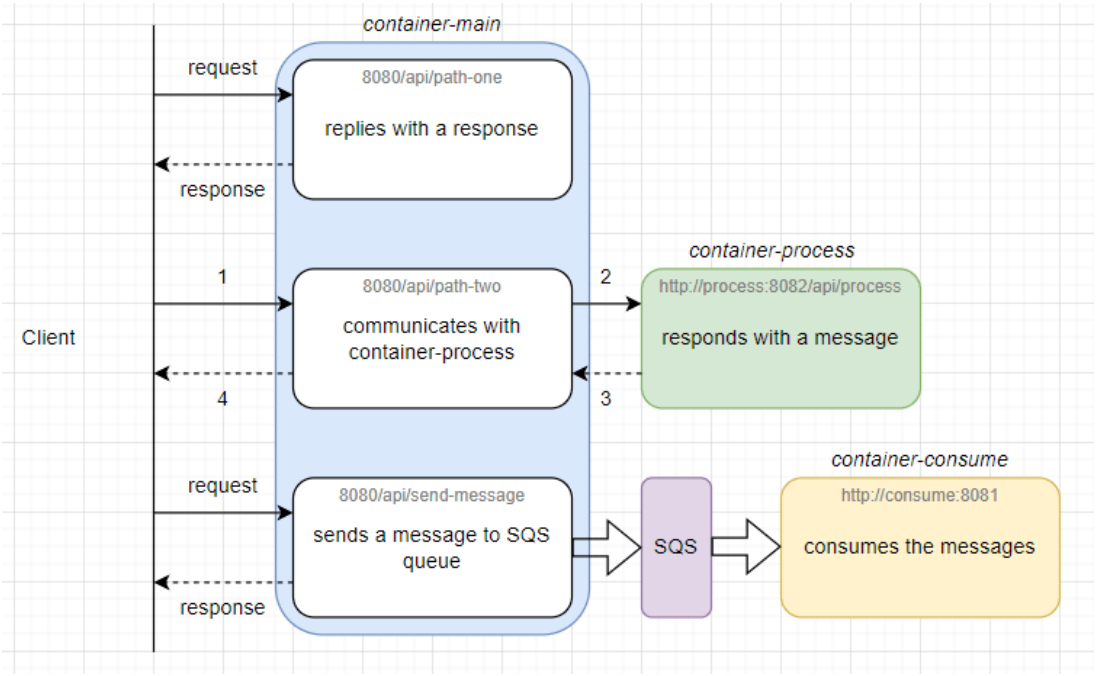

Рисунок 3.1 – Приклад організації мікросервісів

Для організації коду в проєкті було вирішено написати власну структуру та архітектуру з підтримкою MVC (Model–View–Controller) патерну, чистої архітектури (Clean Architecture)[31] та домен–орієнтованого дизайну (DDD)[22]. Ці підходи були обрані для створення архітектури, яку можна легко перевикористовувати у проектах, забезпечуючи модульність, масштабованість та підтримуваність.

Цілі архітектури:

– забезпечення чіткої роздільності між різними частинами системи, що дозволяє легко змінювати або додавати нові функціональності без впливу на інші частини системи;

– створення архітектури, яка дозволяє легко масштабувати систему шляхом додавання нових мікросервісів або розширення існуючих;

– забезпечення легкої підтримки та розширення системи завдяки використанню стандартних патернів проектування;

– створення загальної архітектури, яку можна використовувати в інших проектах з мінімальними змінами.

Основні принципи:

– MVC (Model–View–Controller): забезпечує чітке розділення логіки додатку на три основні компоненти: модель, представлення і контролер. Це допомагає розділити відповідальності та спростити підтримку і розвиток додатку;

– чиста архітектура (Clean Architecture): спрямована на забезпечення незалежності від фреймворків, можливості легкого тестування та підвищення гнучкості системи. Вона передбачає розділення системи на кілька шарів, кожен з яких відповідає за свою частину функціональності;

– DDD (Domain–Driven Design): домен–орієнтований дизайн фокусується на створенні системи, яка відображає бізнес–логіку та вимоги. Це досягається за допомогою використання доменних моделей, агрегатів, репозиторіїв та інших концепцій DDD[22].

На рис. 3.2 наведено структуру та архітектуру проєкту.

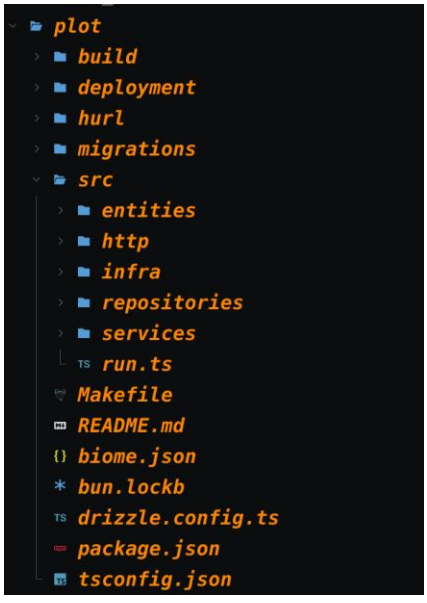

Рисунок 3.2 – Структура та архітектура коду

Така організація коду дозволяє створювати архітектуру, яка легко підтримується, масштабована та може бути перевикористана в інших проектах. Використання MVC, чистої архітектури та DDD забезпечує модульність, чітке розділення відповідальностей та легкість у тестуванні та розвитку системи.

3.3 Використання AWS

Для побудови додатку було вирішено використовувати сервіси Amazon Web Services (AWS), що забезпечує інтеграцію, масштабованість, безпеку та надійність системи.

Таблиця 3.3 – Опис основних задіяних у проєкті AWS сервісів

Сервіс	**Опис**	**Призначення**	**Використання**
SQS	**Amazon Simple Queue Service** – це повністю керована черга повідомлень, що дозволяє ізолювати та масштабувати мікросервіси.	Забезпечення асинхронної обробки запитів та передачі повідомлень між мікросервісами.	Використовується для комунікації між сервісами системи.
RDS	**Amazon Relational Database Service** – це керована реляційна база даних, яка спрощує налаштування, експлуатацію та масштабування баз даних.	Зберігання структурованих даних з високою надійністю та продуктивністю.	Використовується для зберігання даних.
EC2	**Amazon Elastic Compute Cloud** – це веб–сервіс, що надає масштабовану обчислювальну потужність в хмарі.	Розгортання та запуск віртуальних серверів для виконання додатків та сервісів.	Використовується для запуску контейнерів з мікросервісами та інших обчислювальних завдань.
ECS	**Amazon Elastic Container Service** – це повністю керована служба контейнерів для запуску Docker контейнерів.	Управління та оркестрація контейнерів у кластері.	Використовується для запуску, масштабування та управління контейнерами з мікросервісами.
ECR	**Amazon Elastic Container Registry** – це сервіс контейнерних реєстрів, який дозволяє зберігати, управляти та розгортати Docker образи.	Зберігання та управління Docker образами для контейнерів.	Використовується для зберігання Docker образів мікросервісів, які використовуються в ECS.

Ці сервіси AWS забезпечують повний цикл розробки, тестування, розгортання та моніторингу додатку, дозволяючи зосередитися на бізнес–логіці та функціональності системи без необхідності турбуватися про інфраструктуру та її управління. На рис. 3.3 наведено огляд інтерфейсу AWS Console.

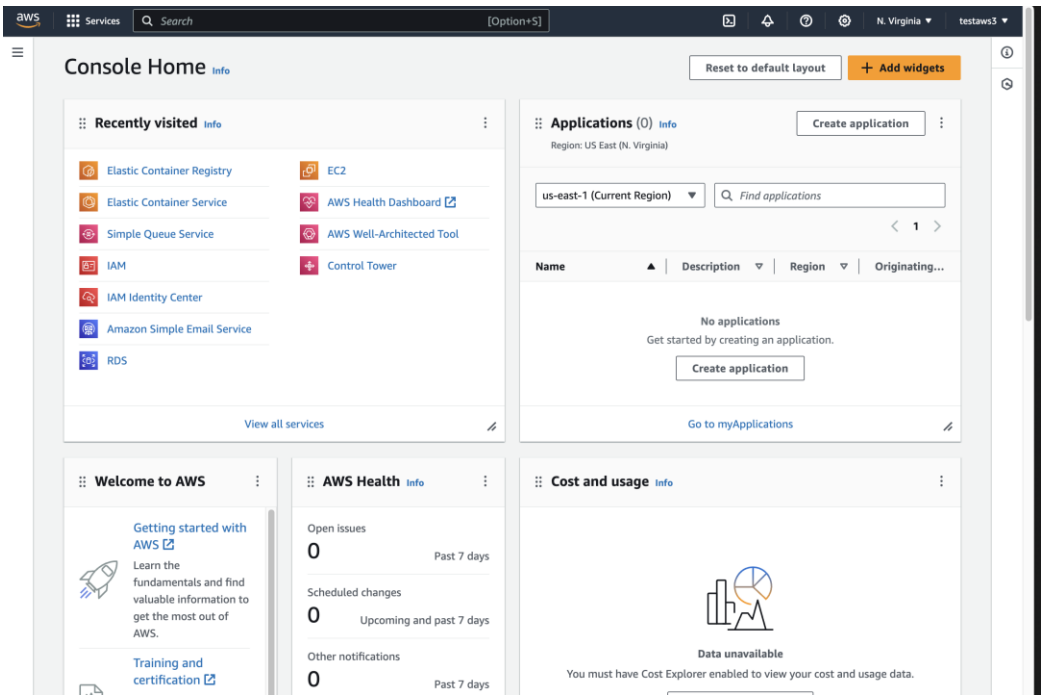

Рисунок 3.3 – Огляд інтерфейсу AWS Console

3.4 Реалізація проєкту на основі AWS

Для початку роботи з AWS і використання сервісів необхідно створити IAM[26] (Identity and Access Management) користувача та налаштувати йому доступ до AWS CLI (Command Line Interface). На рис. 3.3 – рис. 3.7 наведено основні можливості IAM сервісу.

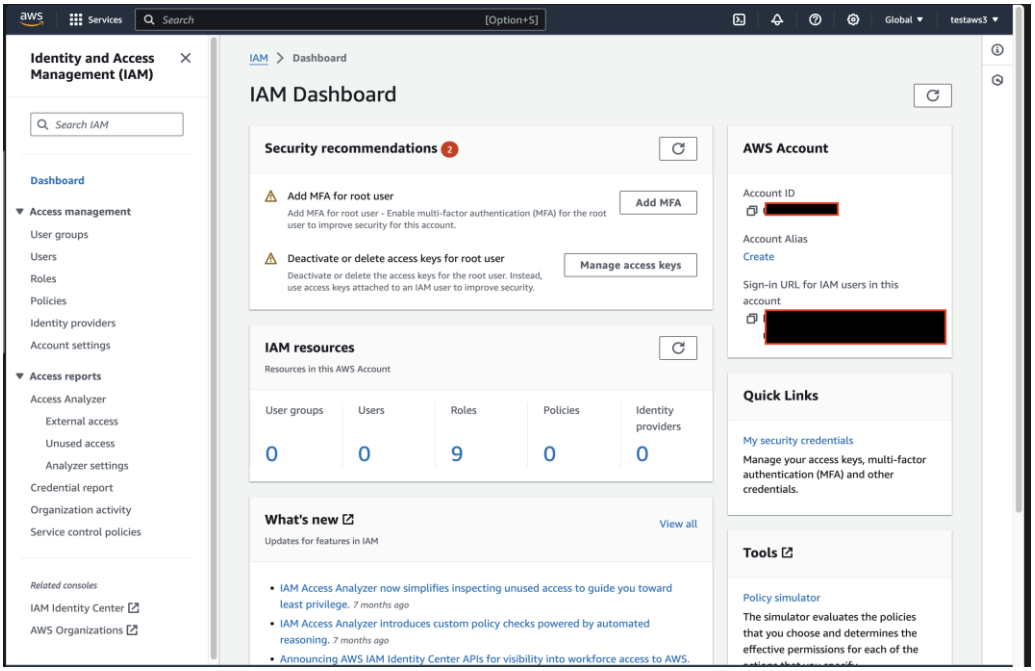

Рисунок 3.4 – Огляд сервісу IAM

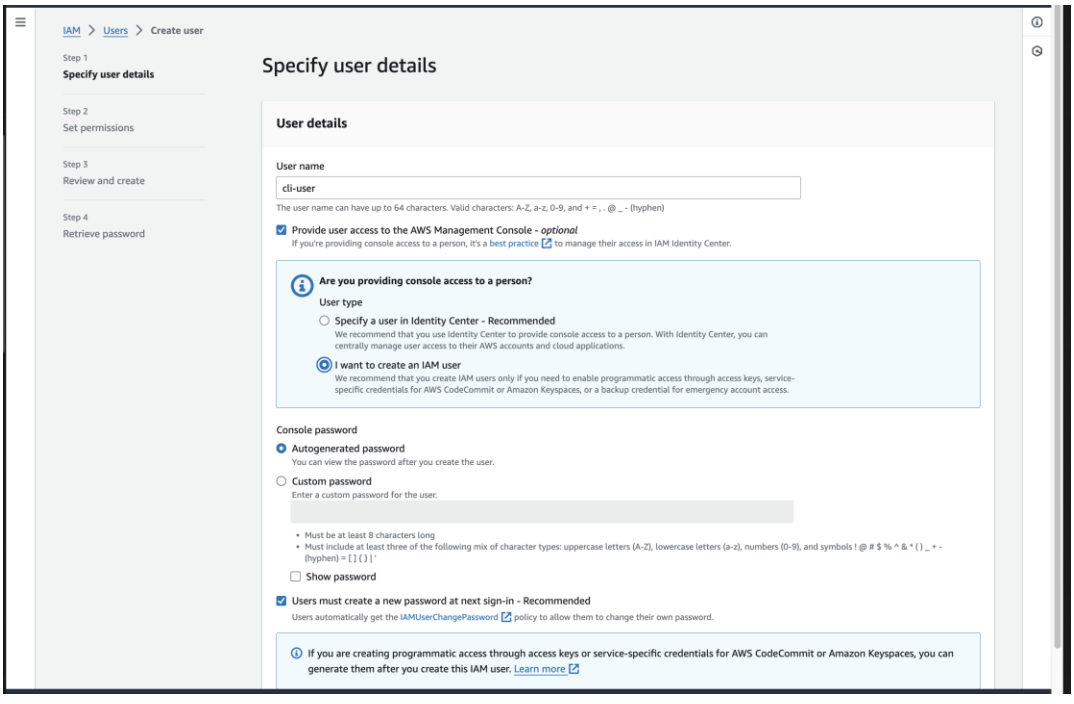

Рисунок 3.5 – Створення IAM користувача

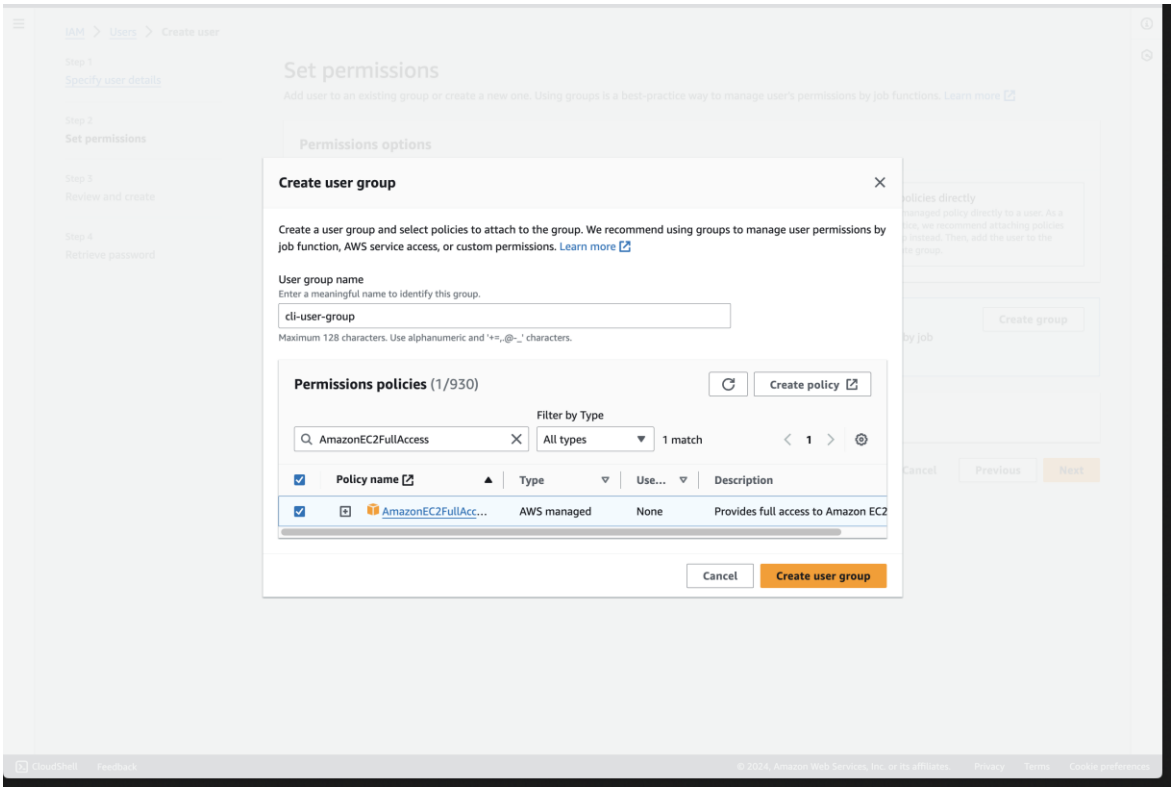

Рисунок 3.6 – Створення необхідних політик для доступу

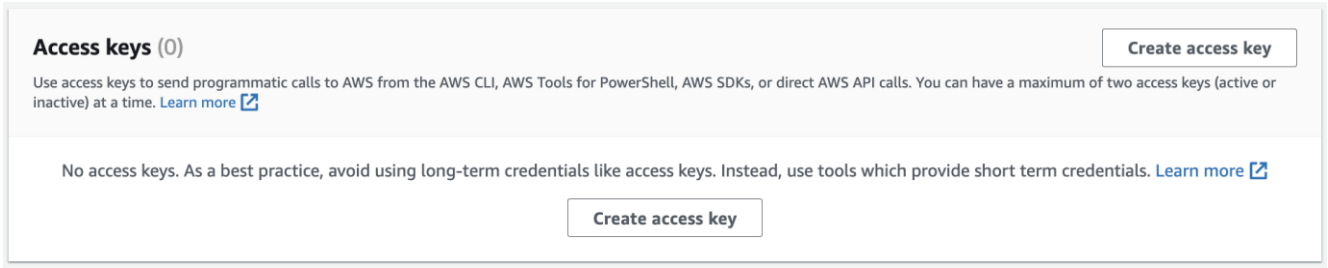

Рисунок 3.7 – Створення ключів доступу до AWS CLI

Далі, щоб налаштувати AWS CLI, треба його встановити, якщо він ще не встановлений. Після встановлення відкрити командний рядок (термінал) і виконати команду aws configure. Ввести Access Key ID та Secret Access Key, отримані раніше, вказати регіон за замовчуванням (наприклад, us–west–2) та формат виводу за замовчуванням (наприклад, json). Після виконання цих кроків IAM користувач буде

створений, налаштований з необхідними правами доступу і зможе використовувати AWS CLI для взаємодії з сервісами AWS.

Після створення IAM користувача та налаштування доступу до AWS CLI, наступним кроком є налаштування Amazon SQS (Simple Queue Service). Amazon SQS дозволяє створювати черги повідомлень для асинхронної обробки даних між мікросервісами. На рис. 3.8 – 3.9 наведено створення та налаштування SQS черги.

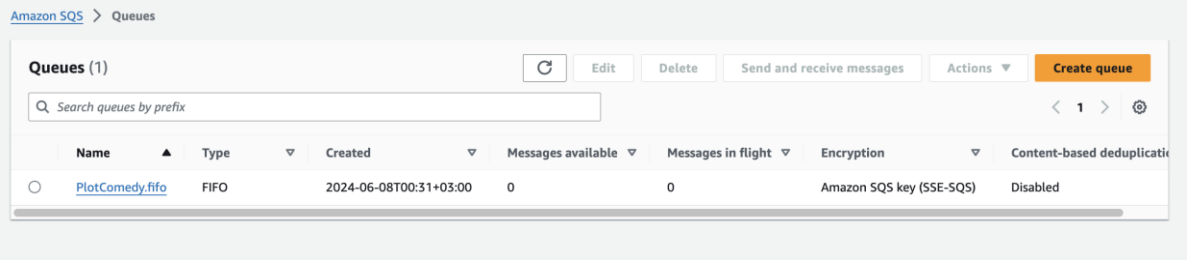

Рисунок 3.8 – Створення SQS черги

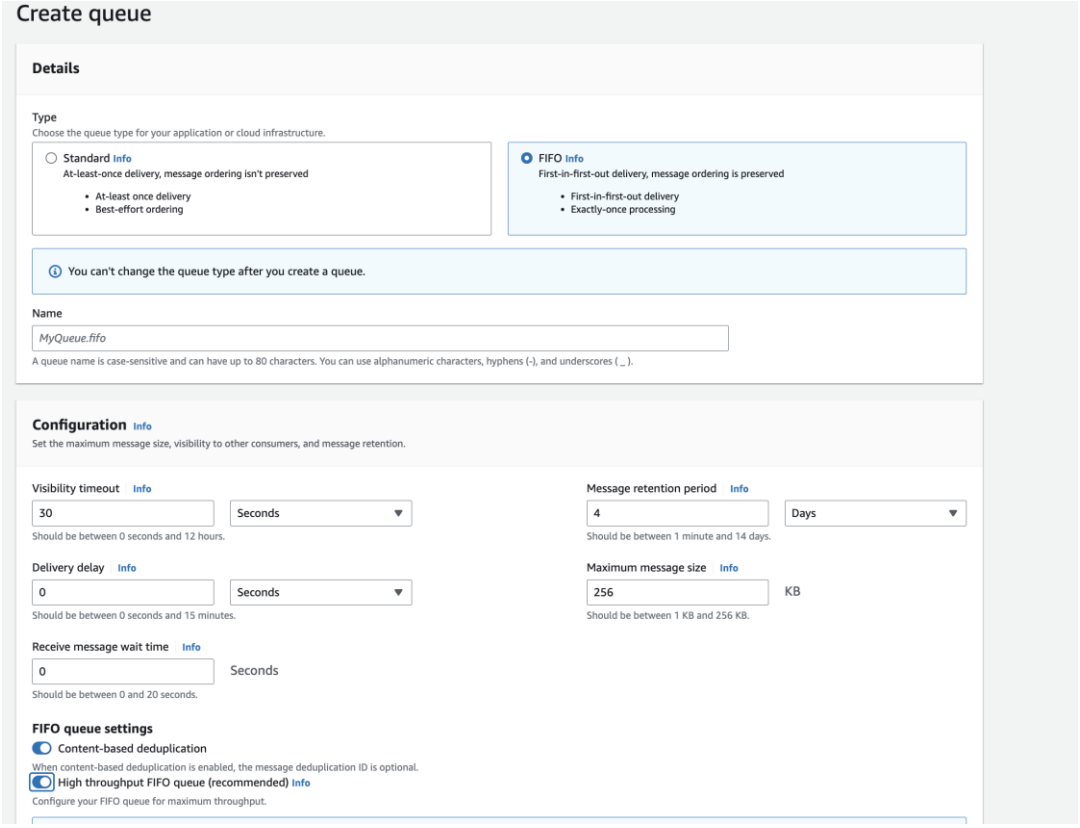

Рисунок 3.9 – Налаштування SQS черги

Amazon SQS FIFO (First–In–First–Out) – це тип черги, який забезпечує зберігання та доставку повідомлень у точному порядку, в якому вони були відправлені. FIFO черги гарантують, що кожне повідомлення буде оброблено лише один раз і в правильному порядку.

Основні принципи роботи Amazon SQS FIFO включають порядок доставки повідомлень, гарантію унікальної доставки, групи повідомлень та використання ідентифікаторів груп і унікальності. FIFO черги гарантують, що повідомлення будуть доставлені та оброблені у тому ж порядку, в якому вони були відправлені. Це особливо важливо для додатків, де правильний порядок обробки має критичне значення, наприклад, обробка транзакцій. Повідомлення в FIFO черзі обробляються лише один раз, що гарантує, що кожне повідомлення буде доставлене та оброблене лише один раз, запобігаючи дублюванню.

FIFO черги підтримують концепцію груп повідомлень. Кожне повідомлення може бути частиною певної групи (Message Group). Повідомлення в одній групі доставляються та обробляються у точному порядку, в якому вони були відправлені. Однак різні групи повідомлень можуть оброблятися паралельно. Кожне повідомлення у FIFO черзі повинно мати ідентифікатор групи (Message Group ID). Повідомлення з одним і тим же Message Group ID доставляються у порядку їх надходження. FIFO черги також використовують ідентифікатор унікальності (Deduplication ID) для уникнення дублювання повідомлень. Повідомлення з однаковим Deduplication ID, відправлені в межах 5 хвилин, будуть вважатися дублюючими і лише одне з них буде доставлено. На рис. 3.10 – 3.11 наведено опис FIFO черги.

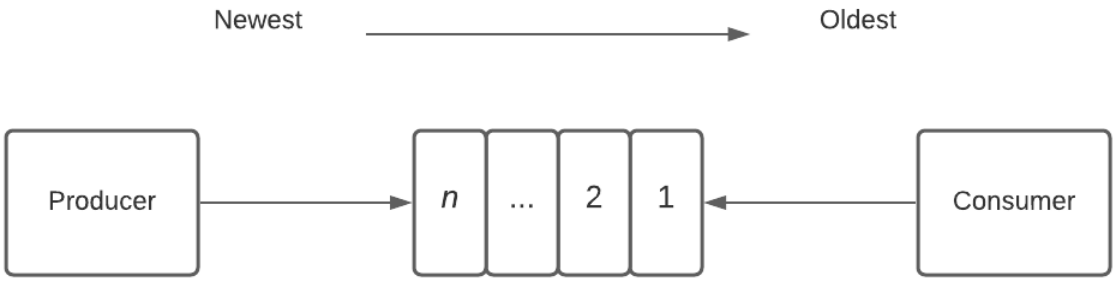

Рисунок 3.10 – Опис FIFO черги

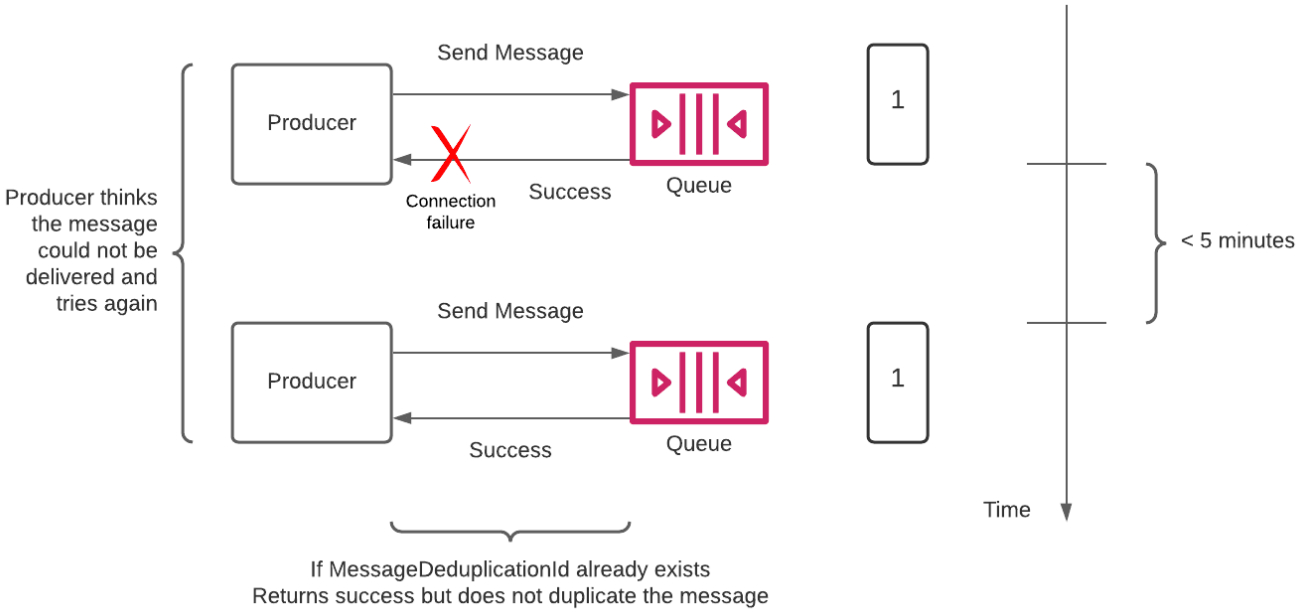

Рисунок 3.11 – Опис роботи FIFO черги

Для збірки та деплою застосунку до хмарного сервісу є потреба в реєстрі Docker контейнерів. Amazon Elastic Container Registry (Amazon ECR) – це повністю керований сервіс реєстру Docker[10] контейнерів, який дозволяє легко зберігати, управляти та розгортати Docker образи. Він інтегрується з Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service (EKS), Docker CLI та іншими інструментами, що робить процес управління контейнерними образами простим та ефективним.

Основні функції та принципи роботи Amazon ECR[25]:

– Amazon ECR забезпечує надійне та масштабоване зберігання контейнерних образів. Образи зберігаються в приватних репозиторіях, які можуть бути захищені політиками доступу. Це забезпечує безпечне зберігання контейнерів та захист від несанкціонованого доступу;

– ECR підтримує управління версіями контейнерних образів, дозволяючи розробникам зберігати різні версії образів і легко перемикатися між ними. Це важливо для тестування, розгортання та відкату версій додатків;

– ECR легко інтегрується з конвеєрами CI/CD, такими як AWS CodePipeline, Jenkins, CircleCI та інші. Це дозволяє автоматизувати процеси збірки, тестування та розгортання контейнерних додатків;

– ECR підтримує IAM (Identity and Access Management) для контролю доступу до репозиторіїв. Це дозволяє встановлювати точні політики доступу, визначаючи, хто може завантажувати, завантажувати та управляти образами;

– ECR використовує Amazon S3 для зберігання образів, забезпечуючи високу доступність і масштабованість. Крім того, ECR підтримує реплікацію образів у різних регіонах, що оптимізує використання мережі і знижує затримки при завантаженні образів.

Після створення Docker образів можна використати Docker CLI для завантаження образів до Amazon ECR. Команди docker push і docker pull дозволяють завантажувати образи до ECR та завантажувати їх з нього. Код, який описує даний алгоритм роботи:

```
.PHONY: login
login:
    @aws ecr–public get–login–password —region $(AWS_REGION) | docker login —username AWS —password–stdin $(ECR_REPO_URL)
```

```
.PHONY: create–repos
create–repos:
	@echo "Checking and creating repositories if they do not exist..."
	@aws ecr–public describe–repositories —region $(AWS_REGION) —repository–names prelude || true
	@aws ecr–public describe–repositories —region $(AWS_REGION) —repository–names comedy || true
	@aws ecr–public describe–repositories —region $(AWS_REGION) —repository–names plot || true
	@aws ecr–public create–repository —region $(AWS_REGION) —repository–name prelude || echo "Repository 'prelude' already exists."
	@aws ecr–public create–repository —region $(AWS_REGION) —repository–name comedy || echo "Repository 'comedy' already exists."
	@aws ecr–public create–repository —region $(AWS_REGION) —repository–name plot || echo "Repository 'plot' already exists."

.PHONY: tag–images
tag–images: use–default–context
	@docker tag $(LOCAL_PRELUDE_IMAGE) $(PRELUDE_IMAGE)
	@docker tag $(LOCAL_COMEDY_IMAGE) $(COMEDY_IMAGE)
	@docker tag $(LOCAL_PLOT_IMAGE) $(PLOT_IMAGE)

.PHONY: push
push: login create–repos tag–images
	@docker push $(PRELUDE_IMAGE)
	@docker push $(COMEDY_IMAGE)
```

```
	@docker push $(PLOT_IMAGE)
```

Цей Makefile описує набір команд для роботи з Amazon Elastic Container Registry (ECR). Він включає цілі для входу в ECR, створення репозиторіїв, тегування Docker образів і їх завантаження в ECR. Опис цілей: login, create–repos, tag–images, і push. Ціль login забезпечує вхід до ECR, використовуючи AWS CLI для отримання пароля і Docker CLI для входу. Ціль create–repos перевіряє наявність та створює необхідні репозиторії у ECR для образів prelude, comedy, і plot. Ціль tag–images створює теги для локальних Docker образів, прив'язуючи їх до відповідних репозиторіїв у ECR. Нарешті, ціль push завантажує Docker образи у ECR після виконання попередніх цілей.

Для розгортання контейнерів у хмарі використовується Amazon Elastic Container Service (ECS) та Docker Compose. ECS – це керована служба оркестрації контейнерів, яка дозволяє легко запускати, зупиняти та керувати контейнерними додатками в кластері. Docker Compose використовується для визначення та запуску багатоконтейнерних Docker додатків. У поєднанні з ECS, Docker Compose дозволяє спрощено визначити інфраструктуру додатків у файлі docker–compose.yml, а потім розгорнути цю інфраструктуру в AWS. Ця інтеграція забезпечує ефективне та гнучке управління контейнерами в хмарі, дозволяючи швидко масштабувати додатки та автоматизувати процеси розгортання та управління.

Для цього використовується docker context, що дозволяє налаштовувати та перемикатися між різними середовищами виконання Docker. docker context спрощує роботу з кількома середовищами (локальними і віддаленими) за допомогою єдиного інтерфейсу командного рядка. На рис. 3.12 – 3.14 показано загальний вигляд команд запуску.

Створення ECS контексту:

```
.PHONY: create–context
create–context:
```

```
	@docker context create ecs $(DOCKER_CONTEXT) —from–env || echo "Context $(DOCKER_CONTEXT) already exists"

.PHONY: use–context
use–context: create–context
	@docker context use $(DOCKER_CONTEXT)

.PHONY: use–default–context
use–default–context:
	@docker context use $(DEFAULT_CONTEXT)
```

Організація деплою через docker–compose.yml:

```
.PHONY: deploy
deploy: use–context
	@PRELUDE_IMAGE=$(PRELUDE_IMAGE) COMEDY_IMAGE=$(COMEDY_IMAGE) PLOT_IMAGE=$(PLOT_IMAGE) docker compose –f docker–compose.yml up –d
```

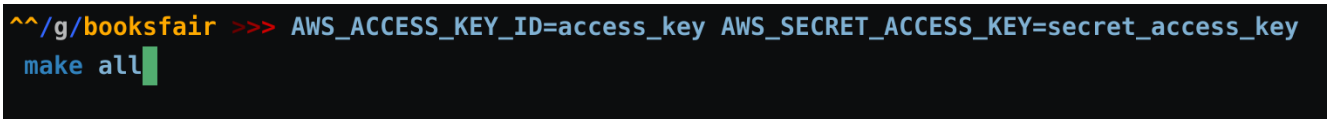

Рисунок 3.12 – Результат використання ECS та ECR

```
An error occurred (RepositoryAlreadyExistsException) when calling the CreateRepository
operation: The repository with name 'prelude' already exists in the registry with id '0
58264499802'
Repository 'prelude' already exists.

An error occurred (RepositoryAlreadyExistsException) when calling the CreateRepository
operation: The repository with name 'comedy' already exists in the registry with id '05
8264499802'
Repository 'comedy' already exists.

An error occurred (RepositoryAlreadyExistsException) when calling the CreateRepository
operation: The repository with name 'plot' already exists in the registry with id '0582
64499802'
Repository 'plot' already exists.
desktop-linux
Current context is now "desktop-linux"
The push refers to repository [public.ecr.aws/q8t1o1a1/prelude]
36d90fbef02f: Pushed
03a3d1f30514: Pushed
4f4fb700ef54: Pushed
94747bd81234: Pushed
52d2c58735d1: Pushed
a6f951e3161c: Pushed
e6aaceb23c53: Pushed
b0c84895a7cb: Pushed
cca8529f9e62: Pushed
7a2d08865d2a: Pushed
7478b37c7663: Pushed
131a87fe3f50: Pushed
36ae12be74a3: Pushed
3e87f2c76455: Pushed
40164d6062d4: Pushed
d052a34dadaf: Pushed
6c2bbcaf0fdd: Pushed
latest: digest: sha256:345eaa6c30a81dd228ea045829bd723c0a8d8a3ea83f23a433a1f188bc21c2e3
 size: 856
The push refers to repository [public.ecr.aws/q8t1o1a1/comedy]
af3020d7c22f: Pushed
c6b39de5b339: Pushed
b409500e040d: Pushed
```

Рисунок 3.13 – Результат використання ECS та ECR

```
^^/g/b/comedy >>> make build TARGET=run                (*S1-comedy-prelude+5) 09:39:53
PULL_FLAG=
NO_CACHE_FLAG=
DOCKER_BUILDKIT=1 docker build   --tag oxb4f/comedy:S1-comedy-prelude-9f8119b --file bu
ild/Dockerfile --target run ..
[+] Building 5.2s (24/24) FINISHED                                  docker:desktop-linux
 => [internal] load build definition from Dockerfile                                0.0s
 => => transferring dockerfile: 699B                                                0.0s
 => [internal] load metadata for docker.io/oven/bun:alpine                          2.1s
 => [internal] load .dockerignore                                                   0.0s
 => => transferring context: 2B                                                     0.0s
 => [base  1/12] FROM docker.io/oven/bun:alpine@sha256:6568a679b87107d3d7d46b829    0.0s
 => => resolve docker.io/oven/bun:alpine@sha256:6568a679b87107d3d7d46b829f614c44    0.0s
 => [internal] load build context                                                   0.7s
 => => transferring context: 67.27kB                                                0.7s
 => CACHED [base  2/12] RUN mkdir -p /app                                           0.0s
 => CACHED [base  3/12] WORKDIR /app                                                0.0s
 => CACHED [base  4/12] RUN mkdir -p comedy                                         0.0s
 => CACHED [base  5/12] COPY comedy/package.json ./comedy/package.json              0.0s
 => CACHED [base  6/12] COPY comedy/src/ comedy/src                                 0.0s
 => CACHED [base  7/12] COPY lib lib                                                0.0s
 => CACHED [base  8/12] COPY plot plot                                              0.0s
 => CACHED [base  9/12] COPY package.json ./                                        0.0s
 => CACHED [base 10/12] COPY bun.lockb ./                                           0.0s
 => CACHED [base 11/12] COPY tsconfig.json ./                                       0.0s
 => CACHED [base 12/12] RUN bun install                                             0.0s
 => CACHED [run 1/7] RUN mkdir -p /app                                              0.0s
 => CACHED [run 2/7] WORKDIR /app                                                   0.0s
 => CACHED [run 3/7] COPY  --from=base /app/. .                                     0.0s
 => CACHED [run 4/7] COPY  --from=base package*.json ./                             0.0s
 => CACHED [run 5/7] COPY comedy/tsconfig*.json ./                                  0.0s
 => CACHED [run 6/7] COPY comedy/drizzle.config.ts ./                               0.0s
 => CACHED [run 7/7] COPY comedy/migrations ./                                      0.0s
 => exporting to image                                                              2.1s
 => => exporting layers                                                             0.0s
 => => exporting manifest sha256:de28b580eb739bdedb6a6dd8491954019d236e1b7c67996    0.0s
 => => exporting config sha256:305a4a6b59e35f4e72bc84a10f129a1c95f3553ff45eaf6dc    0.0s
 => => exporting attestation manifest sha256:7895dac251c1b04b8128e671517e061c68f    0.0s
 => => exporting manifest list sha256:6e9bdff63e1e36b340686c1cd3bfec074e47b7fb67    0.0s
 => => naming to docker.io/oxb4f/comedy:S1-comedy-prelude-9f8119b                   0.0s
```

Рисунок 3.14 – Результат збірки окремого сервісу через Dockerfile

Наступним етапом буде налаштування значень змінних оточення (env) для кожного з трьох мікросервісів: prelude, comedy та plot. Це необхідно для того, щоб кожна служба мала правильні конфігурації та змогла взаємодіяти з необхідними ресурсами, такими як база даних, черги SQS та інші сервіси.

Приклад налаштування:

```
NODE_ENV=dev

APP_PORT=8081

POSTGRES_USER=comedy
POSTGRES_PASSWORD=postgres
POSTGRES_DB=app
POSTGRES_HOST=db
POSTGRES_PORT=5433

AWS_ACCESS_KEY_ID=access_id
AWS_SECRET_ACCESS_KEY=secret_access_key
AWS_REGION=eu–central–1
AWS_SQS_QUEUE_URL=https://sqs.eu–central–
1.amazonaws.com/111111111/Queue.fifo

PRELUDE_BASE_URL=http://prelude:8080
```

Окремі налаштування для сервісу авторизації та аутентифікації:

```
JWT_SECRET=secret
JWT_ACCESS_LIFETIME=1000000
```

```
REFRESH_TOKEN_LIFETIME=1000000
```

Для кожного мікросервісу було створено локальне середовище для розробкки, де можна розгорнути базу даних локально та провести тестування. Це забезпечує можливість перевіряти роботу сервісів в умовах, максимально наближених до реальних, без необхідності використання зовнішніх ресурсів.

Для кожного мікросервісу було окремо розроблено Makefile та Dockerfile, разом з docker–compose.yml, що дозволяє легко розгорнути та налаштувати локальне дев середовище під кожен сервіс.

Приклад локального запуску (рис. 3.15 – 3.16):

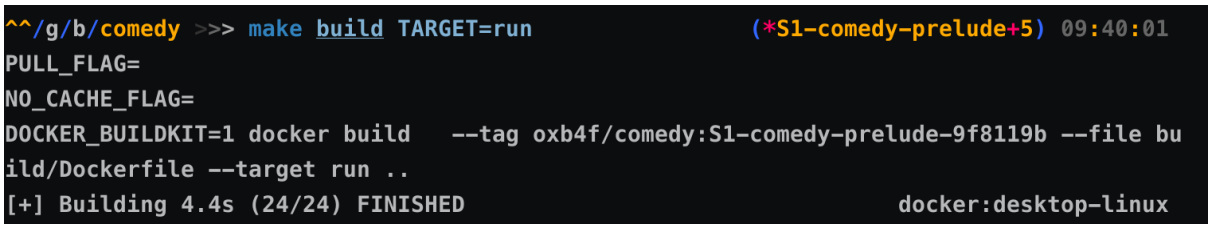

Рисунок 3.15 – Результат збірки Docker image

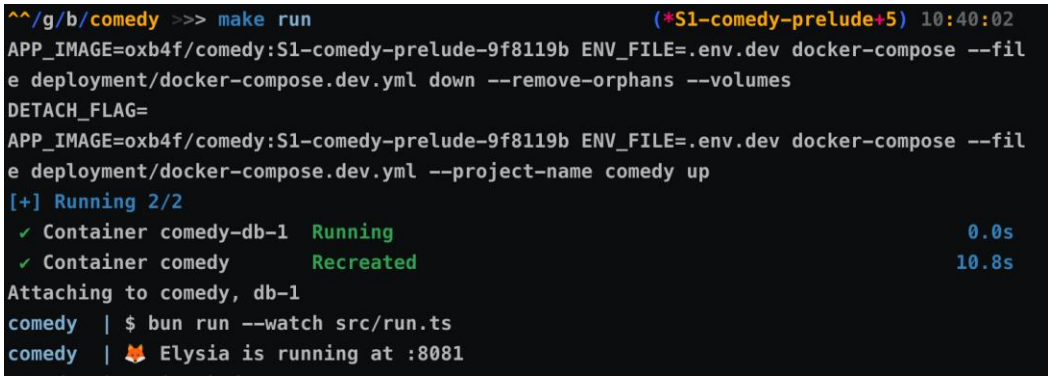

Рисунок 3.16 – Результат запуску за допомогою Docker Compose

Makefile включає дві основні цілі: build та run.

```
build: ## Build the Docker image for the application
```

```
ifeq ($(PULL),true)
	PULL_FLAG=—pull
else
	PULL_FLAG=
endif
ifeq ($(NO_CACHE),true)
	NO_CACHE_FLAG=—no–cache
else
	NO_CACHE_FLAG=
endif
	DOCKER_BUILDKIT=1 docker build $(PULL_FLAG) $(NO_CACHE_FLAG) —tag ${APP_IMAGE} —file ${DOCKER_FILE} —target ${TARGET} ..

run: stop ## Start the application using Docker Compose
ifeq ($(DETACH),true)
	DETACH_FLAG=–d
else
	DETACH_FLAG=
endif
	APP_IMAGE=${APP_IMAGE} ENV_FILE=${ENV_FILE} docker–compose —file ${DOCKER_COMPOSE} —project–name ${APP_NAME} up $(DETACH_FLAG)
```

Ціль build збирає Docker образ для додатку. Спочатку перевіряються змінні PULL і NO_CACHE: якщо вони встановлені на true, додаються відповідні флаги —pull та —no–cache. Потім виконується команда docker build з використанням BuildKit

(DOCKER_BUILDKIT=1), тега образу (${APP_IMAGE}), Dockerfile (${DOCKER_FILE}) та цільового етапу збірки (${TARGET}).

Ціль run запускає додаток за допомогою Docker Compose. Спочатку зупиняється додаток, якщо він вже запущений. Далі перевіряється змінна DETACH: якщо вона встановлена на true, додається флаг –d для запуску контейнерів у фоновому режимі. Виконується команда docker–compose з використанням змінних APP_IMAGE, ENV_FILE, файлу Docker Compose (${DOCKER_COMPOSE}) та імені проекту (${APP_NAME}). На рис. 3.17 показано локальний запуск застосунку.

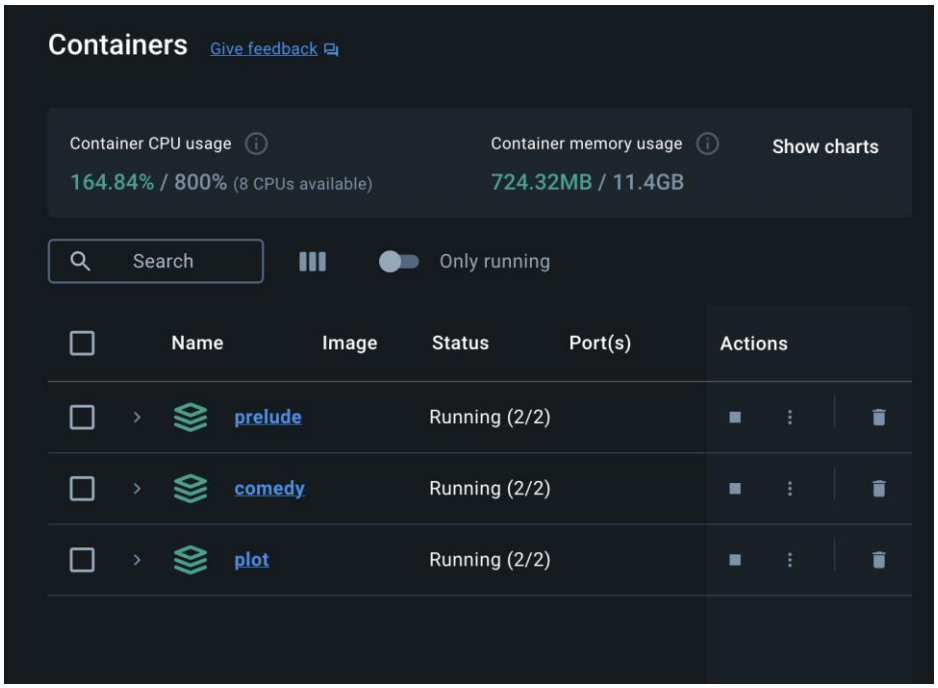

Рисунок 3.17 – Повністю налаштований локальний запуск

Для сервісів, розгорнутих на базі AWS, буде використано Amazon RDS (Relational Database Service) для забезпечення надійного, масштабованого та керованого рішення для роботи з реляційними базами даних. Amazon RDS підтримує кілька типів баз даних, таких як PostgreSQL, MySQL, MariaDB, Oracle та SQL Server.

Для проєкту було використано PostgreSQL. На рис. 3.18 показано налаштування Amazon RDS.

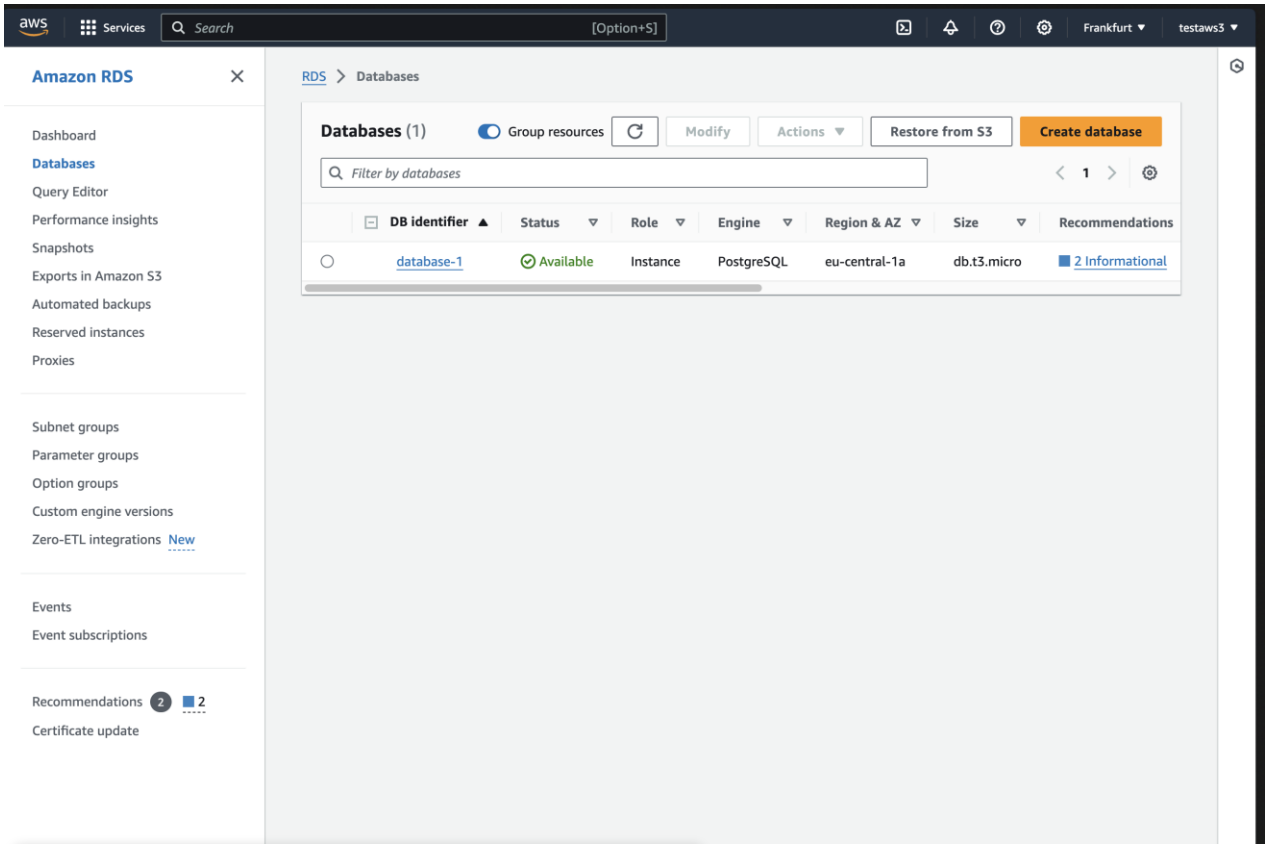

Рисунок 3.18 – Amazon RDS налаштування

Приклад налаштувань:

```
POSTGRES_USER=comedy
POSTGRES_PASSWORD=postgres
POSTGRES_DB=app
POSTGRES_HOST=mydatabase.c6c8h2dvew1z.eu–central–1.rds.amazonaws.com
POSTGRES_PORT=5433
```

Приклад запуску тестів (рис. 3.19) за допомогою Hurl, що тестує усю систему на коректну роботу:

```
^^/g/b/plot >>> make hurl
hurl/01-create.hurl: Running [1/5]
hurl/01-create.hurl: Success (2 request(s) in 369 ms)
hurl/02-get.hurl: Running [2/5]
hurl/02-get.hurl: Success (4 request(s) in 677 ms)
hurl/03-update.hurl: Running [3/5]
hurl/03-update.hurl: Success (4 request(s) in 654 ms)
hurl/04-remove.hurl: Running [4/5]
hurl/04-remove.hurl: Success (4 request(s) in 846 ms)
hurl/05-list.hurl: Running [5/5]
hurl/05-list.hurl: Success (5 request(s) in 898 ms)
--------------------------------------------------------------------------------
Executed files:  5
Succeeded files: 5 (100.0%)
Failed files:    0 (0.0%)
Duration:        3449 ms
```

Рисунок 3.19 – Приклад запуску тестів

Висновки до розділу 3

У цьому розділі було проведено аналіз і вибір технологічного стеку для побудови системи мікросервісів, яка складається з трьох серверів, що взаємодіють по різним протоколам та написані на різних фреймворках та середовищах виконання. Всі сервери побудовані на основі AWS, що забезпечує масштабованість, безпеку та надійність системи.

Основними технологіями для реалізації мікросервісної архітектури обрано:

– Node.js за його асинхронність, високу продуктивність і велику екосистему модулів;

– Bun за високу швидкість виконання JavaScript коду і вбудовані інструменти для тестування та пакування;

– TypeScript за статичну типізацію, яка дозволяє виявляти помилки на етапі компіляції та підтримує сучасні стандарти JavaScript;

– Docker за ізоляцію середовища, що робить додатки портативними і незалежними від середовища, а також спрощує процеси розгортання та оновлення додатків;

– PostgreSQL за його надійність, стабільність роботи, підтримку стандартів SQL і розширюваність;

– Makefile для автоматизації процесів збірки та деплою додатків;

– Hurl для тестування HTTP API завдяки простому синтаксису і підтримці багаторазових запитів.

Всі ці технології забезпечують створення гнучкої, масштабованої та легко підтримуваної системи. Для організації коду в проєкті було обрано підхід монорепозиторію, що дозволяє зберігати всі сервіси в одному репозиторії і спрощує процеси розробки, підтримки та розгортання.

Для кожного з трьох мікросервісів було окремо розроблено Makefile та Dockerfile, що дозволяє легко розгорнути та налаштувати локальне дев середовище під кожен сервіс. Застосування Amazon SQS для асинхронної обробки запитів між сервісами, Amazon ECR для зберігання Docker образів, Amazon ECS для оркестрації Docker та Amazon RDS для зберігання даних додало надійності та ефективності системі.

ВИСНОВКИ

Сучасні технології допомагають людству вирішувати багато проблем, створюючи інтелектуальні та практичні системи. В умовах швидкого розвитку інформаційних технологій важливість впровадження інноваційних рішень зростає. Розподілені системи, побудовані на основі хмарних сервісів, стали ключовими елементами для забезпечення надійності, масштабованості та безпеки в організаціях. Вони дозволяють гнучко адаптувати ресурси до змінних потреб, підвищуючи конкурентоспроможність та ефективність управління інформаційними потоками.

Метою проекту було розробити розподілену систему на основі хмарних сервісів, яка враховуватиме переваги та недоліки існуючих рішень на ринку, відповідатиме потребам користувачів і забезпечуватиме надійність, масштабованість та безпеку. У процесі роботи було виконано такі завдання:

– проведено аналіз сучасних розподілених систем та їх роль в IT;

– проведено аналіз існуючих хмарних технологій та вибір хмарного провайдера для реалізації практичної частини;

– моделювання розподіленої системи з використанням Amazon Web Services (AWS), Node.js, Bun, TypeScript, PostgreSQL, Fastify, Elysiajs, Drizzle ORM, Docker, Makefile, AWS CLI, JWT, Swagger.

Розробка розподіленої системи з трьома мікросервісами для книжкового додатку: сервіс авторизації та аутентифікації, сервіс менеджменту користувачів, сервіс менеджменту книжок.

Результатом роботи є розроблена система, яка демонструє ефективність хмарних сервісів у забезпеченні надійності, масштабованості та безпеки. Вона може бути використана як основа для подальших досліджень та впровадження у практику інших розподілених систем на основі хмарних технологій. Впроваджена система

менеджменту бібліотеки книг є прикладом ефективного використання хмарних сервісів для вирішення реальних бізнес–задач.

Таким чином, результати цієї роботи показують, що хмарні технології надають широкі можливості для створення розподілених систем, які можуть бути успішно використані в різних галузях, від бізнесу до наукових досліджень. Розподілені системи на основі хмарних технологій значно підвищують ефективність, надійність та безпеку інформаційних систем, сприяючи інноваційному розвитку та конкурентоспроможності організацій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1) The First 50 Years of Living Online: ARPANET and Internet – Computer History Museum. URL: https://www.computerhistory.org/revolution/networking/4/19 (дата звернення: 01.10.2023).

2) Хван, К., Фокс, Г. К., Донгара, Дж. Дж. Розподілені та хмарні обчислення: Від паралельної обробки до Інтернету речей / Кай Хван, Джеффрі К. Фокс, Джек Дж. Донгара. – М.: Morgan Kaufmann, 2012. – 672 с. 19 (дата звернення: 01.10.2023).

3) Amazon Web Services (AWS). Офіційна документація AWS. URL: https://aws.amazon.com (дата звернення: 01.10.2023).

4) Amazon EC2. Документація EC2. URL: https://aws.amazon.com/ec2/ (дата звернення: 10.10.2023).

5) Microsoft Azure. Офіційна документація Azure. URL: https://azure.microsoft.com (дата звернення: 20.10.2023).

6) Node.js. Офіційна документація Node.js. URL: https://nodejs.org (дата звернення: 01.11.2023).

7) TypeScript. Офіційна документація TypeScript. URL: https://www.typescriptlang.org (дата звернення: 15.11.2023).

8) PostgreSQL. Офіційна документація PostgreSQL. URL: https://www.postgresql.org (дата звернення: 28.11.2023).

9) Fastify. Офіційна документація Fastify. URL: https://www.fastify.io (дата звернення: 10.12.2023).

10) Docker. Офіційна документація Docker. URL: https://www.docker.com (дата звернення: 20.12.2023).

11) Makefile. Керівництво по створенню Makefile. URL: https://www.gnu.org/software/make/ (дата звернення: 30.12.2023).

12) JWT (JSON Web Token). Офіційна документація JWT. URL: https://jwt.io (дата звернення: 10.01.2024).

13) Swagger. Офіційна документація Swagger. URL: https://swagger.io (дата звернення: 20.01.2024).

14) Elysiajs. Документація та ресурси по Elysiajs. URL: https://github.com/elysiajs (дата звернення: 30.01.2024).

15) Drizzle ORM. Документація та ресурси по Drizzle ORM. URL: https://drizzle–orm.dev (дата звернення: 10.02.2024).

16) AWS CLI. Офіційна документація AWS CLI. URL: https://aws.amazon.com/cli/ (дата звернення: 20.02.2024).

17) Newman S. Building Microservices. O'Reilly Media, 2015. 280 с. (дата звернення: 01.03.2024).

18) Kleppmann M. Designing Data–Intensive Applications. O'Reilly Media, 2017. 616 с. (дата звернення: 10.03.2024).

19) Cope M. Architecting the Cloud. John Wiley & Sons, 2014. 300 с. (дата звернення: 20.03.2024).

20) AWS Lambda. Документація Lambda. URL: https://aws.amazon.com/lambda/ (дата звернення: 30.03.2024).

21) Vernon V. Implementing Domain–Driven Design. Addison–Wesley Professional, 2013. 656 с. (дата звернення: 10.04.2024).

22) Sbarski P. Serverless Architectures on AWS. Manning Publications, 2017. 320 с. (дата звернення: 20.04.2024).

23) Amazon ECS. Документація ECS. URL: https://aws.amazon.com/ecs/ (дата звернення: 01.05.2024).

24) Amazon RDS. Документація RDS. URL: https://aws.amazon.com/rds/ (дата звернення: 10.05.2024).

25) Amazon ECR. Документація ECR. URL: https://aws.amazon.com/ecr/ (дата звернення: 20.05.2024).

26) Amazon IAM. Документація IAM. URL: https://aws.amazon.com/iam/ (дата звернення: 30.05.2024).

27) Edge Computing. Статті та ресурси по Edge Computing. URL: https://ieee.org (дата звернення: 10.06.2024).

28) Річардсон, К. Мікросервісні патерни: з прикладами на Java / К. Річардсон. – Київ: Manning Publications, 2019. – 520 с. (дата звернення: 10.10.2023).

29) Bun. Офіційна документація Bun. UR: https://bun.sh/ (дата звернення: 01.11.2023).

30) Hurl. Офіційна документація Hurl. URL: https://hurl.dev/ (дата звернення: 01.11.2023).

31) Мартін, Р. С. Чиста архітектура: Посібник майстра з структури та дизайну програмного забезпечення / Р. С. Мартін. – Pearson, 2017. – 432 с. (дата звернення: 10.12.2023).

32) Amazon Simple Queue Service Documentation. URL: https://docs.aws.amazon.com/sqs/ (дата звернення: 13.06.2024).

ДОДАТОК А
Код сервісу авторизації

```
infra
config.ts
import { z } from "zod";

export interface Config {
	APP_PORT: number;

	POSTGRES_USER: string;
	POSTGRES_PASSWORD: string;
	POSTGRES_DB: string;
	POSTGRES_HOST: string;
	POSTGRES_PORT: number;

	JWT_SECRET: string;
	JWT_ACCESS_LIFETIME: number;

	REFRESH_TOKEN_LIFETIME: number;
}

export class ConfigValidationError extends Error {
	private constructor(message: string) {
		super(message);
		this.name = "ConfigValidationError";
	}

	static from(errors: Record<string, string[]>) {
		const message = Object.entries(errors)
			.map(([field, errors]) => {
				return `${field}: ${errors.map((msg) =>
msg.toLowerCase()).join(", ")}`;
			})
			.join("\n");

		return new ConfigValidationError(message);
	}
}

export const configSchema = z.object({
	APP_PORT: z.coerce.number().int().positive(),

	POSTGRES_USER: z.string(),
	POSTGRES_PASSWORD: z.string(),
	POSTGRES_DB: z.string(),
	POSTGRES_HOST: z.string(),
	POSTGRES_PORT: z.coerce.number().int().positive(),

	JWT_SECRET: z.string(),
	JWT_ACCESS_LIFETIME: z.coerce.number().int().positive(),

	REFRESH_TOKEN_LIFETIME: z.coerce.number().int().positive(),
});

export function load() {
	try {
```

```
            return configSchema.parse(process.env);
      } catch (error) {
            const formattedError = error.format() as Record<
                  string,
                  { _errors?: string[] }
            >;
            const transformedErrors: Record<string, string[]> = {};

            for (const [field, errorDetail] of Object.entries(formattedError)) {
                  if (Array.isArray(errorDetail._errors)) {
                        transformedErrors[field] = errorDetail._errors;
                  }
            }

            throw ConfigValidationError.from(transformedErrors);
      }
}
```

```
schema.ts
import { json, pgTable, serial, varchar } from "drizzle-orm/pg-core";

export const accesses = pgTable("accesses", {
      id: serial("id").primaryKey(),
      login: varchar("login", { length: 128 }).notNull().unique(),
      password: varchar("password", { length: 64 }).notNull(),
      jwtPayload: json("jwt_payload").notNull(),
      refreshTokens: json("refresh_tokens").notNull(),
});
```

```
migration.ts
import * as path from "node:path";

import { migrate } from "drizzle-orm/postgres-js/migrator";

import { getConnection } from "./db";
import type { ConnectionParams } from "./utils";

export async function migrateDb({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): Promise<void> {
      const connection = await getConnection({
            user,
            password,
            db,
            host,
            port,
      });

      return migrate(connection, {
            migrationsFolder: path.join(process.cwd(), "migrations"),
      });
}
```

```
utils.ts
export type ConnectionParams = {
      user: string;
      password: string;
      db: string;
      host: string;
      port: number;
};

export function buildConnectionString({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): string {
      return `postgres://${user}:${password}@${host}:${port}/${db}`;
}

db.ts
import { type PostgresJsDatabase, drizzle } from "drizzle-orm/postgres-js";
import postgres from "postgres";

import * as schema from "./schema";
import { type ConnectionParams, buildConnectionString } from "./utils";

export type Connection = PostgresJsDatabase<typeof schema>;

let _connection: Connection;

export async function getConnection({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): Promise<Connection> {
      const connectionString = buildConnectionString({
            user,
            password,
            db,
            host,
            port,
      });

      if (!_connection)
            _connection = drizzle(postgres(connectionString, { max: 5 }), { schema
});

      return _connection;
}

repositories
base.ts
import type { Connection } from "../infra/data-src/pg/db";

export abstract class BaseRepository {
```

```
      constructor(protected _connection: Connection) {}
}

access.ts
import { and, eq } from "drizzle-orm";
import { Access, type AccessPayload } from "../entities/access";
import { accesses } from "../infra/data-src/pg/schema";
import { BaseRepository } from "./base";

export type SelectAccess = typeof accesses.$inferSelect;

export class AccessRepository extends BaseRepository {
      async createFromEntity(access: Access) {
            const result = await this._connection
                  .insert(accesses)
                  .values({
                        login: access.login,
                        password: access.password,
                        jwtPayload: access.jwtPayload,
                        refreshTokens:
Object.fromEntries(access.refreshTokens.entries()),
                  })
                  .returning({ id: accesses.id })
                  .execute();

            access.id = result[0].id;

            return access;
      }

      async getAccess(selectBy: Partial<SelectAccess>) {
            const result = await this._connection
                  .select({
                        id: accesses.id,
                        login: accesses.login,
                        password: accesses.password,
                        jwtPayload: accesses.jwtPayload,
                        refreshTokens: accesses.refreshTokens,
                  })
                  .from(accesses)
                  .where(
                        and(
                              ...Object.keys(selectBy).map((k: keyof typeof selectBy)
=>
                                    eq(accesses[k], selectBy[k]),
                              ),
                        ),
                  )
                  .limit(1)
                  .execute();

            const raw = result?.[0];

            if (!raw?.id) return null;

            return Access.from(raw as AccessPayload);
      }
```

```
	async updateJwtPayloadFromEntity(access: Access) {
		await this._connection
			.update(accesses)
			.set({ jwtPayload: access.jwtPayload });
	}

	async updateFromEntity(access: Access) {
		if (!access.id) return;

		await this._connection
			.update(accesses)
			.set({
				login: access.login,
				password: access.password,
				refreshTokens:
Object.fromEntries(access.refreshTokens.entries()),
			})
			.where(eq(accesses.id, access.id))
			.execute();
	}
}

http
entry.ts
import path from "node:path";

import auto from "@fastify/autoload";
import type { FastifyInstance } from "fastify";
import fp from "fastify-plugin";
import {
	serializerCompiler,
	validatorCompiler,
} from "fastify-type-provider-zod";

import { getConnection } from "../infra/data-src/pg/db";
import { migrateDb } from "../infra/data-src/pg/migration";

const serverEntryPointPlugin = fp(async function serverEntryPoint(
	fastify: FastifyInstance,
) {
	await fastify.register(auto, {
		dir: path.join(__dirname, "plugins"),
		options: {},
	});

	fastify.setValidatorCompiler(validatorCompiler);
	fastify.setSerializerCompiler(serializerCompiler);

	const dbConnCredentials = {
		user: fastify.config.POSTGRES_USER,
		password: fastify.config.POSTGRES_PASSWORD,
		db: fastify.config.POSTGRES_DB,
		host: fastify.config.POSTGRES_HOST,
		port: fastify.config.POSTGRES_PORT,
	};
```

```
      fastify.decorate("db", {
            connection: await getConnection(dbConnCredentials),
      });

      await migrateDb(dbConnCredentials);

      await fastify.register(auto, {
            dir: path.join(__dirname, "routes"),
            dirNameRoutePrefix: true,
            options: {},
      });
});

export default serverEntryPointPlugin;

fastify.d.ts
import "fastify";
import type { Config } from "../infra/config";
import type { Connection } from "../infra/data-src/pg/db";
import type { Context } from "../services/context";

declare module "fastify" {
      interface FastifyInstance {
            db: { connection: Connection };
            config: Config;
      }

      interface FastifyRequest {
            serviceContext: Context;
      }
}

server.ts
import fastify from "fastify";
import serverEntryPointPlugin from "./entry";

const server = fastify({ logger: true });

server
      .register(serverEntryPointPlugin)
      .then(() => server.listen({ port: server.config.APP_PORT, host: "0.0.0.0" }));

error-handler.ts
import type { FastifyInstance, FastifyReply, FastifyRequest } from "fastify";
import fp from "fastify-plugin";
import { match } from "ts-pattern";
import { ZodError } from "zod";
import { AccessDuplicationError } from "../../services/accesses/errors/access-
duplication-error";
import { AccessLoginError } from "../../services/accesses/errors/access-login-error";
import { AccessRefreshError } from "../../services/accesses/errors/access-refresh-
error";
import { JwtVerifyError } from "../../services/accesses/errors/jwt-verify-error";

const serviceLayerErrors = [
      AccessLoginError,
      AccessDuplicationError,
```

```
	AccessRefreshError,
	JwtVerifyError,
] as const;

const allErrors = [...serviceLayerErrors, ZodError, Error] as const;

type ExtractEnumTypeFromArray<T extends readonly any[]> = T extends readonly [
	infer Head,
	...infer Tail,
]
	? Head extends { new (...args: any[]): infer O }
		? O | ExtractEnumTypeFromArray<Tail>
		: never
	: never;

function sendErrorResponse(
	reply: FastifyReply,
	statusCode: number,
	errorType: string,
	message: string,
	data?: any,
) {
	reply.status(statusCode).send({
		statusCode,
		error: errorType,
		message,
		data,
	});
}

const errorHandlerPlugin = fp(async function errorHandler(
	fastify: FastifyInstance,
) {
	fastify.setErrorHandler(
		(
			error: ExtractEnumTypeFromArray<typeof allErrors>,
			_req: FastifyRequest,
			reply: FastifyReply,
		) => {
			match(error)
				.when(
					(e) => e instanceof ZodError,
					(zodError: ZodError) => {
						sendErrorResponse(
							reply,
							422,
							"VALIDATION_ERROR",
							"Request validation error",
							zodError.errors.map((ve) => ({
								path: ve.path,
								message: ve.message,
							})),
						);
					},
				)
				.when(
```

```
                              (e) => serviceLayerErrors.some((ec) => e instanceof
ec),
                              (
                                    matchedError: ExtractEnumTypeFromArray<typeof
serviceLayerErrors>,
                              ) => {
                                    sendErrorResponse(
                                          reply,
                                          422,
                                          "UNPROCESSABLE_ENTITY",
                                          matchedError.message,
                                    );
                              },
                        )
                        .otherwise((e) => {
                              sendErrorResponse(
                                    reply,
                                    500,
                                    "INTERNAL_SERVER_ERROR",
                                    "Service is unavailable due to internal reasons",
                              );
                        });
            },
      );
});

export default errorHandlerPlugin;
```

context.ts

```
import type { FastifyInstance } from "fastify";
import fp from "fastify-plugin";
import { AccessRepository } from "../../repositories/access";

const contextPlugin = fp(async function context(fastify: FastifyInstance) {
      fastify.decorateRequest("serviceContext", null);

      fastify.addHook("preHandler", (req, _reply, done) => {
            req.serviceContext = {
                  accessRepository: new AccessRepository(fastify.db.connection),
                  config: fastify.config,
            };
            done();
      });
});

export default contextPlugin;
```

config.ts

```
import type { FastifyInstance } from "fastify";
import fp from "fastify-plugin";
import { type Config, load } from "../../infra/config";

const configPlugin = fp(async function config(fastify: FastifyInstance) {
      fastify.decorate<Config>("config", load());
});

export default configPlugin;
```

```
ping.ts
import type { FastifyInstance } from "fastify";
import { factory } from "../../services/ping/get";

export default async function ping(fastify: FastifyInstance) {
      const getPing = factory();

      fastify.route({
            method: "GET",
            url: "/ping",
            handler: (req) => getPing({ context: req.serviceContext }),
      });
}

accesses.ts
import crypto from "node:crypto";
import type { FastifyInstance, FastifyRequest } from "fastify";

import type { RefreshDtoIn } from "src/services/accesses/refresh/dto-in";
import type { UpdateDtoIn } from "src/services/accesses/update/dto-in";
import { factory as createFactory } from "../../services/accesses/create/create";
import type { CreateAccessDtoIn } from "../../services/accesses/create/dto-in";
import type { LoginDtoIn } from "../../services/accesses/login/dto-in";
import { factory as loginFactory } from "../../services/accesses/login/login";
import { factory as refreshFactory } from "../../services/accesses/refresh/refresh";
import { factory as updateFactory } from "../../services/accesses/update/update";
import type { VerifyDtoIn } from "../../services/accesses/verify/dto-in";
import { factory as verifyFactory } from "../../services/accesses/verify/verify";

function generateUserAgentHash(userAgent?: string) {
      return crypto
            .createHash("sha256")
            .update(userAgent ?? "default")
            .digest("base64");
}

export default async function accesses(fastify: FastifyInstance) {
      const createAccess = createFactory();
      const login = loginFactory();
      const verify = verifyFactory();
      const refresh = refreshFactory();
      const update = updateFactory();

      fastify.route({
            method: "POST",
            url: "/accesses",
            handler: async (
                  req: FastifyRequest<{ Body: Omit<CreateAccessDtoIn, "deviceId"> }>,
            ) =>
                  createAccess({
                        dto: {
                              ...req.body,
                              deviceId: generateUserAgentHash(req.headers["user-
agent"]),
                        },
                        context: req.serviceContext,
```

```
                  }),
      });

      fastify.route({
            method: "POST",
            url: "/accesses/login",
            handler: async (
                  req: FastifyRequest<{ Body: Omit<LoginDtoIn, "deviceId"> }>,
            ) =>
                  login({
                        dto: {
                              ...req.body,
                              deviceId: generateUserAgentHash(req.headers["user-
agent"]),
                        },
                        context: req.serviceContext,
                  }),
      });

      fastify.route({
            method: "POST",
            url: "/accesses/verify",
            handler: async (req: FastifyRequest<{ Body: VerifyDtoIn }>) =>
                  verify({
                        dto: req.body,
                        context: req.serviceContext,
                  }),
      });

      fastify.route({
            method: "POST",
            url: "/accesses/:accessId/refresh",
            handler: async (
                  req: FastifyRequest<{
                        Body: Omit<RefreshDtoIn, "accessId" | "deviceId">;
                        Params: { accessId: InstanceType<typeof
RefreshDtoIn>["accessId"] };
                  }>,
            ) =>
                  refresh({
                        dto: {
                              ...req.body,
                              ...req.params,
                              deviceId: generateUserAgentHash(req.headers["user-
agent"]),
                        },
                        context: req.serviceContext,
                  }),
      });

      fastify.route({
            method: "POST",
            url: "/accesses/:accessId",
            handler: async (
                  req: FastifyRequest<{
                        Body: Omit<UpdateDtoIn, "accessId">;
```

```
                        Params: { accessId: InstanceType<typeof
UpdateDtoIn>["accessId"] };
                  }>,
            ) =>
                  update({
                        dto: { ...req.body, ...req.params },
                        context: req.serviceContext,
                  }),
      });
}

run.ts
import "./http/server";

services
make.ts
import type { ZodSchema } from "zod";
import type { Context } from "./context";

export type ActionDto<T extends Record<string, any> = Record<string, any>> = T;
export type ActionArg<T extends Record<string, any>> = {
      dto?: ActionDto<T>;
      context: Context;
};
export type TAction<T extends Record<string, any>, R> = (
      arg: ActionArg<T>,
) => Promise<R>;

export function makeService<T extends Record<string, any>, R>(
      action: TAction<T, R>,
      validationSchema?: ZodSchema<T>,
) {
      const proxyHandler = {
            async apply(target: TAction<T, R>, thisArg: unknown, args:
[ActionArg<T>]) {
                  if (validationSchema)
                        args[0].dto = await validationSchema.parseAsync(args[0].dto);

                  return target.apply(thisArg, args);
            },
      };

      return new Proxy(action, proxyHandler);
}

context.ts
import type { Config } from "../infra/config";
import type { AccessRepository } from "../repositories/access";

export interface Context {
      readonly config: Config;
      readonly accessRepository: AccessRepository;
}

get.ts
import { makeService } from "../make";
```

```
async function get() {
	return { ping: "pong" } as const;
}

export function factory() {
	return makeService(get);
}
```

```
dto-out.ts
export class UpdateDtoOut {
	constructor(
		public readonly id: number,
		public readonly login: string,
	) {}
}
```

```
dto-in.ts
export class UpdateDtoIn {
	constructor(
		public readonly accessId: number,
		public readonly login?: string,
		public readonly password?: string,
	) {}
}
```

```
update.ts
import z from "zod";

import type { Context } from "../../context";
import { makeService } from "../../make";
import { AccessUpdateError } from "../errors/access-update-error";
import type { UpdateDtoIn } from "./dto-in";
import { UpdateDtoOut } from "./dto-out";

async function update({
	dto,
	context,
}: { dto: UpdateDtoIn; context: Context }) {
	const access = await context.accessRepository.getAccess({
		id: dto.accessId,
	});

	if (!access) throw new AccessUpdateError();

	if (dto.login) access.login = dto.login;
	if (dto.password) await access.setPassword(dto.password);

	await context.accessRepository.updateFromEntity(access);

	return new UpdateDtoOut(access.id!, access.login);
}

export function factory() {
	return makeService(
		update,
		z.object({
			accessId: z.coerce.number().min(1).max(999_999_999_999_999),
```

```
                  login: z.string().max(64).optional(),
                  password: z.string().min(8).max(64).optional(),
            }),
      );
}

verify.ts
import z from "zod";

import { Access } from "../../../entities/access";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { JwtVerifyError } from "../errors/jwt-verify-error";
import type { VerifyDtoIn } from "./dto-in";

async function verify({
      dto,
      context,
}: {
      dto: VerifyDtoIn;
      context: Context;
}) {
      const decodedJwtAccess = await Access.verifyAndDecodeJwt(
            dto.jwtAccess,
            context.config.JWT_SECRET,
            dto.ignoreExpiration,
      );

      if (!decodedJwtAccess?.accessId) throw new JwtVerifyError();

      const access = await context.accessRepository.getAccess({
            id: decodedJwtAccess.accessId,
      });

      if (!access) throw new JwtVerifyError();
}

export function factory() {
      return makeService(
            verify,
            z.object({
                  jwtAccess: z.string().max(256),
                  ignoreExpiration: z.boolean().optional(),
            }),
      );
}

dto-in.ts
export class VerifyDtoIn {
      constructor(
            public readonly jwtAccess: string,
            public readonly ignoreExpiration?: boolean,
      ) {}
}

refresh.ts
import z from "zod";
```

```
import type { Context } from "../../context";
import { makeService } from "../../make";
import { AccessRefreshError } from "../errors/access-refresh-error";
import type { RefreshDtoIn } from "./dto-in";
import { RefreshDtoOut } from "./dto-out";

async function refresh({
      dto,
      context,
}: { dto: RefreshDtoIn; context: Context }) {
      const access = await context.accessRepository.getAccess({
            id: dto.accessId,
      });

      if (!access) throw new AccessRefreshError();

      const refreshTokenVerifyResult = await access
            .verifyRefreshToken(dto.deviceId, dto.refreshToken)
            .catch(() => {
                  throw new AccessRefreshError();
            });

      if (!refreshTokenVerifyResult) throw new AccessRefreshError();

      access.deleteRefreshToken(dto.deviceId);

      const refreshToken = await access.addOrReplaceRefreshToken(
            dto.deviceId,
            context.config.JWT_SECRET,
            context.config.REFRESH_TOKEN_LIFETIME,
      );

      await context.accessRepository.updateFromEntity(access);

      const jwtAccess = await access.generateJwtAccess(
            context.config.JWT_SECRET,
            context.config.JWT_ACCESS_LIFETIME,
      );

      return new RefreshDtoOut(access.id!, access.login, refreshToken, jwtAccess);
}

export function factory() {
      return makeService(
            refresh,
            z.object({
                  refreshToken: z.string().min(1).max(256),
                  deviceId: z.string().min(8).max(256),
                  accessId: z.coerce.number().min(1).max(999_999_999_999_999),
            }),
      );
}

dto-out.ts
export class RefreshDtoOut {
      constructor(
```

```
            public readonly id: number,
            public readonly login: string,
            public readonly refreshToken: string,
            public readonly jwtAccess: string,
      ) {}
}

dto-in.ts
export class RefreshDtoIn {
      constructor(
            public readonly accessId: number,
            public readonly deviceId: string,
            public readonly refreshToken: string,
      ) {}
}

access-login-error.ts
export class AccessLoginError extends Error {
      constructor(message = "Invalid credentials") {
            super(message);
            this.name = "AccessLoginError";
      }
}

jwt-verify-error.ts
export class JwtVerifyError extends Error {
      constructor(message = "Invalid JWT") {
            super(message);
            this.name = "JwtVerifyError";
      }
}

access-duplication-error.ts
export class AccessDuplicationError extends Error {
      constructor(message = "Access already exists") {
            super(message);
            this.name = "AccessDuplicationError";
      }
}

access-refresh-error.ts
export class AccessRefreshError extends Error {
      constructor(message = "Refresh is failed") {
            super(message);
            this.name = "AccessRefreshError";
      }
}

access-update-error.ts
export class AccessUpdateError extends Error {
      constructor(message = "Update is failed") {
            super(message);
            this.name = "AccessUpdateError";
      }
}

dto-out.ts
```

```
export class CreateAccessDtoOut {
	constructor(
		public readonly id: number,
		public readonly login: string,
		public readonly refreshToken: string,
		public readonly jwtAccess: string,
	) {}
}

dto-in.ts
export class CreateAccessDtoIn {
	constructor(
		public readonly login: string,
		public readonly password: string,
		public readonly deviceId: string,
		public readonly jwtPayload: Record<string, any>,
	) {}
}

create.ts
import z from "zod";

import { Access } from "../../../entities/access";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { AccessDuplicationError } from "../errors/access-duplication-error";
import type { CreateAccessDtoIn } from "./dto-in";
import { CreateAccessDtoOut } from "./dto-out";

async function create({
	dto,
	context,
}: {
	dto: CreateAccessDtoIn;
	context: Context;
}) {
	const existingAccess = await context.accessRepository.getAccess({
		login: dto.login,
	});

	if (existingAccess) throw new AccessDuplicationError();

	const access = await Access.from({
		id: null,
		login: dto.login,
		password: dto.password,
		jwtPayload: dto.jwtPayload,
		refreshTokens: {},
	});

	await access.setPassword(dto.password);

	const refreshToken = await access.addOrReplaceRefreshToken(
		dto.deviceId,
		context.config.JWT_SECRET,
		context.config.REFRESH_TOKEN_LIFETIME,
	);
```

```
	await context.accessRepository.createFromEntity(access);

	const jwtAccess = await access.generateJwtAccess(
		context.config.JWT_SECRET,
		context.config.JWT_ACCESS_LIFETIME,
	);

	return new CreateAccessDtoOut(
		access.id!,
		access.login,
		refreshToken,
		jwtAccess,
	);
}

export function factory() {
	return makeService(
		create,
		z.object({
			login: z.string().max(64),
			password: z.string().min(8).max(64),
			deviceId: z.string().min(8).max(256),
			jwtPayload: z.record(z.string(), z.any()),
		}),
	);
}
```

login.ts

```
import z from "zod";

import type { Context } from "../../context";
import { makeService } from "../../make";
import { AccessLoginError } from "../errors/access-login-error";
import type { LoginDtoIn } from "./dto-in";
import { LoginDtoOut } from "./dto-out";

async function login({ dto, context }: { dto: LoginDtoIn; context: Context }) {
	const access = await context.accessRepository.getAccess({
		login: dto.login,
	});

	if (!access) throw new AccessLoginError();

	const passwordVerifyResult = await access
		.verifyPassword(dto.password)
		.catch(() => {
			throw new AccessLoginError();
		});

	if (!passwordVerifyResult) throw new AccessLoginError();

	if (dto.jwtPayload) {
		access.jwtPayload = dto.jwtPayload;

		await context.accessRepository.updateJwtPayloadFromEntity(access);
	}
```

```
	access.deleteRefreshToken(dto.deviceId);

	const refreshToken = await access.addOrReplaceRefreshToken(
		dto.deviceId,
		context.config.JWT_SECRET,
		context.config.REFRESH_TOKEN_LIFETIME,
	);

	const jwtAccess = await access.generateJwtAccess(
		context.config.JWT_SECRET,
		context.config.JWT_ACCESS_LIFETIME,
	);

	return new LoginDtoOut(access.id!, access.login, refreshToken, jwtAccess);
}

export function factory() {
	return makeService(
		login,
		z.object({
			login: z.string().max(64),
			password: z.string().min(8).max(64),
			deviceId: z.string().min(8).max(256),
			jwtPayload: z.record(z.string(), z.any()).optional(),
		}),
	);
}

dto-out.ts
export class LoginDtoOut {
	constructor(
		public readonly id: number,
		public readonly login: string,
		public readonly refreshToken: string,
		public readonly jwtAccess: string,
	) {}
}

dto-in.ts
export class LoginDtoIn {
	constructor(
		public readonly login: string,
		public readonly password: string,
		public readonly deviceId: string,
		public readonly jwtPayload: Record<string, any>,
	) {}
}

entities
access.ts
import bcrypt from "bcrypt";
import { TokenError, createSigner, createVerifier } from "fast-jwt";

import type { MaybeNumberId } from "./types/id";

const SALT_ROUNDS = 10;
```

```
export interface CompleteJwtPayload {
	[k: string]: any;

	accessId: MaybeNumberId;
}

export type AccessPayload = {
	id: MaybeNumberId;
	login: string;
	password: string;
	jwtPayload: Record<string, any>;
	refreshTokens: Record<string, string>;
};

export class Access {
	private _id: MaybeNumberId;
	private _login: string;
	private _password: string;
	private _jwtPayload: Record<string, any>;
	private _refreshTokens: Map<string, string>;

	private constructor(payload: AccessPayload) {
		this._id = payload.id ?? null;
		this._login = payload.login;
		this._password = payload.password;
		this._jwtPayload = payload.jwtPayload;
		this._refreshTokens = new Map(Object.entries(payload.refreshTokens));
	}

	static async from(payload: AccessPayload): Promise<Access> {
		return new Access({ ...payload });
	}

	get id(): MaybeNumberId {
		return this._id;
	}

	get login(): string {
		return this._login;
	}

	get password(): string {
		return this._password;
	}

	set login(login: string) {
		this._login = login;
	}

	set id(id: MaybeNumberId) {
		this._id = id;
	}

	get jwtPayload() {
		return this._jwtPayload;
	}
```

```
	set jwtPayload(jwtPayload: Record<string, any>) {
		this._jwtPayload = jwtPayload;
	}

	get refreshTokens(): Map<string, string> {
		return this._refreshTokens;
	}

	async setPassword(password: string): Promise<void> {
		this._password = await bcrypt.hash(password, SALT_ROUNDS);
	}

	async addOrReplaceRefreshToken(
		id: string,
		secret: string,
		lifetime: number,
	): Promise<string> {
		const refreshToken = await this.generateRefreshToken(secret, lifetime);

		this._refreshTokens.set(id, await bcrypt.hash(refreshToken,
SALT_ROUNDS));

		return refreshToken;
	}

	deleteRefreshToken(id: string): void {
		this._refreshTokens.delete(id);
	}

	async verifyRefreshToken(id: string, refreshToken: string) {
		if (!this._refreshTokens.has(id)) return false;

		return await bcrypt.compare(
			refreshToken,
			this._refreshTokens.get(id) as string,
		);
	}

	async verifyPassword(password: string) {
		return await bcrypt.compare(password, this._password);
	}

	async generateJwtAccess(secret: string, lifetime: number) {
		return await this.#generateJwt(secret, lifetime, {
			...this._jwtPayload,
			accessId: this._id,
		});
	}

	async generateRefreshToken(secret: string, lifetime: number) {
		return await this.#generateJwt(secret, lifetime, { accessId: this._id });
	}

	async #generateJwt(
		secret: string,
		lifetime: number,
```

```
            payload: Record<string, unknown>,
      ) {
            const signer = createSigner({
                  key: async () => secret,
                  expiresIn: lifetime,
            });

            return await signer({ ...payload });
      }

      static async verifyAndDecodeJwt(
            token: string,
            secret: string,
            ignoreExpiration = false,
      ): Promise<CompleteJwtPayload | null> {
            try {
                  const verifier = createVerifier({
                        key: async () => secret,
                        ignoreExpiration,
                  });

                  return await verifier(token);
            } catch (error) {
                  if (error instanceof TokenError) return null;
                  throw error;
            }
      }

      toPlainObject(): Readonly<Omit<AccessPayload, "password" | "refreshTokens">> {
            return {
                  id: this._id,
                  login: this._login,
                  jwtPayload: this._jwtPayload,
            };
      }
}
```

id.ts

```
import type { Optional } from "./optional";

type ToOptional<T> = Optional<T>;

export type MaybeNumberId = ToOptional<number>;
```

optional.ts

```
export type Optional<T> = T | null | undefined;
```

ДОДАТОК Б
Код сервісу управління користувачами

```
infra
config.ts
import { z } from "zod";

export interface Config {
	APP_PORT: number;

	POSTGRES_USER: string;
	POSTGRES_PASSWORD: string;
	POSTGRES_DB: string;
	POSTGRES_HOST: string;
	POSTGRES_PORT: number;

	AWS_ACCESS_KEY_ID: string;
	AWS_SECRET_ACCESS_KEY: string;
	AWS_REGION: string;
	AWS_SQS_QUEUE_URL: string;

	PRELUDE_BASE_URL: string;
}

export class ConfigValidationError extends Error {
	private constructor(message: string) {
		super(message);
		this.name = "ConfigValidationError";
	}

	static from(errors: Record<string, string[]>) {
		const message = Object.entries(errors)
			.map(([field, errors]) => {
				return `${field}: ${errors.map((msg) =>
msg.toLowerCase()).join(", ")}`;
			})
			.join("\n");

		return new ConfigValidationError(message);
	}
}

export const configSchema = z.object({
	APP_PORT: z.coerce.number().int().positive(),

	POSTGRES_USER: z.string(),
	POSTGRES_PASSWORD: z.string(),
	POSTGRES_DB: z.string(),
	POSTGRES_HOST: z.string(),
	POSTGRES_PORT: z.coerce.number().int().positive(),

	AWS_ACCESS_KEY_ID: z.string(),
	AWS_SECRET_ACCESS_KEY: z.string(),
	AWS_REGION: z.string(),
	AWS_SQS_QUEUE_URL: z.string(),

	PRELUDE_BASE_URL: z.string(),
```

```
});

export function load(): Config {
	try {
		return configSchema.parse(Bun.env);
	} catch (error: any) {
		const formattedError = error.format() as Record<
			string,
			{ _errors?: string[] }
		>;
		const transformedErrors: Record<string, string[]> = {};

		for (const [field, errorDetail] of Object.entries(formattedError)) {
			if (Array.isArray(errorDetail._errors)) {
				transformedErrors[field] = errorDetail._errors;
			}
		}

		throw ConfigValidationError.from(transformedErrors);
	}
}
```

```
schema.ts
import { bigint, date, pgTable, serial, varchar } from "drizzle-orm/pg-core";

export const users = pgTable("users", {
	id: serial("id").primaryKey(),
	name: varchar("name", { length: 128 }),
	birthday: date("birthday", { mode: "date" }),
	username: varchar("username", { length: 128 }).notNull().unique(),
	externalAccessId: bigint("external_access_id", { mode: "number" })
		.notNull()
		.unique(),
});
```

```
migration.ts
import * as path from "node:path";

import { migrate } from "drizzle-orm/postgres-js/migrator";

import { getConnection } from "./db";
import type { ConnectionParams } from "./utils";

export async function migrateDb({
	user,
	password,
	db,
	host,
	port,
}: ConnectionParams): Promise<void> {
	const connection = await getConnection({
		user,
		password,
		db,
		host,
		port,
	});
```

```
      return migrate(connection, {
            migrationsFolder: path.join(process.cwd(), "migrations"),
      });
}

utils.ts
export type ConnectionParams = {
      user: string;
      password: string;
      db: string;
      host: string;
      port: number;
};

export function buildConnectionString({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): string {
      return `postgres://${user}:${password}@${host}:${port}/${db}`;
}

db.ts
import { type PostgresJsDatabase, drizzle } from "drizzle-orm/postgres-js";
import postgres from "postgres";

import * as schema from "./schema";
import { type ConnectionParams, buildConnectionString } from "./utils";

export type Connection = PostgresJsDatabase<typeof schema>;

let _connection: Connection;

export async function getConnection({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): Promise<Connection> {
      const connectionString = buildConnectionString({
            user,
            password,
            db,
            host,
            port,
      });

      if (!_connection)
            _connection = drizzle(postgres(connectionString, { max: 5 }), {
                  schema,
                  logger: false,
            });
```

```
      return _connection;
}

repositories
base.ts
import type { Connection } from "../infra/data-src/pg/db";

export abstract class BaseRepository {
      constructor(protected _connection: Connection) {}
}

user.ts
import { and, eq } from "drizzle-orm";
import { User, type UserPayload } from "../entities/user";
import { users } from "../infra/data-src/pg/schema";
import { BaseRepository } from "./base";

export type SelectUser = typeof users.$inferSelect;

export class UserRepository extends BaseRepository {
      async createFromEntity(user: User) {
            const result = await this._connection
                  .insert(users)
                  .values({
                        name: user.name,
                        username: user.username,
                        birthday: user.birthday,
                        externalAccessId: user.externalAccessId,
                  })
                  .returning({ id: users.id })
                  .execute();

            user.id = result[0].id;

            return user;
      }

      async getUser(selectBy: Partial<SelectUser>) {
            const eqArray = [];

            for (const k of Object.keys(selectBy) as Array<keyof typeof selectBy>) {
                  if (selectBy[k] === undefined) continue;

                  eqArray.push(eq(users[k], selectBy[k]!));
            }

            const result = await this._connection
                  .select({
                        id: users.id,
                        name: users.name,
                        username: users.username,
                        birthday: users.birthday,
                        externalAccessId: users.externalAccessId,
                  })
                  .from(users)
                  .where(and(...eqArray))
                  .limit(1)
```

```
                .execute();

            const raw = result?.[0];

            if (!raw?.id) return null;

            return User.from(raw as UserPayload);
      }

      async updateFromEntity(user: User) {
            if (!user.id) return;

            await this._connection
                  .update(users)
                  .set({
                        name: user.name,
                        username: user.username,
                        birthday: user.birthday,
                  })
                  .where(eq(users.id, user.id))
                  .execute();
      }
}
```

sqs
handler.ts

```
import { match } from "ts-pattern";
import { type ZodSchema, z } from "zod";
import type { Context } from "../services/context";
import { GetDtoIn } from "../services/users/get/dto-in";
import { factory as getFactory } from "../services/users/get/get";

const getUserArgsSchema = z.object({
      userId: z.number().optional(),
      accessId: z.number().optional(),
});

const messageSchema = z.object({
      action: z.string(),
      args: z.object({}).passthrough(),
});

type Actions = "getUser";

const actionsSchema: Record<Actions, ZodSchema<any>> = {
      getUser: getUserArgsSchema,
};

export async function handler(message: Record<string, any>, context: Context) {
      const parseResult = messageSchema.safeParse(message);

      if (!parseResult.success) return false;

      const action = parseResult.data.action as Actions;
      const actionSchema = actionsSchema[action];

      if (!actionSchema) return false;
```

```
	const parsedActionArgs = actionSchema.parse(parseResult.data.args);

	return match(action)
		.with("getUser", async () => {
			const { userId, accessId } = parsedActionArgs;

			const dto = new GetDtoIn(userId, accessId);

			let message = {};

			try {
				const result = await getFactory()({ dto, context });

				message = result.toJSON();
			} catch (error: any) {
				message = { error: error.message };
			}

			await context.aws.sqs.sendMessage(
				context.config.AWS_SQS_QUEUE_URL,
				message,
			);

			return true;
		})
		.otherwise(() => {
			return false;
		});
}

http
server.ts
import { swagger } from "@elysiajs/swagger";
import { Elysia } from "elysia";
import { onError } from "./hooks/on-error";
import { awsPlugin } from "./plugins/aws";
import { configPlugin } from "./plugins/config";
import { contextPlugin } from "./plugins/context";
import { createLoggerFactory } from "./plugins/logger";
import { usersRoute } from "./routes/users";

export const app = new Elysia()
	.use(swagger())
	.use(configPlugin)
	.use(awsPlugin)
	.use(contextPlugin)
	.use(createLoggerFactory())
	.onError(({ error, set }) => {
		onError(error, set);
	})
	.get("/ping", () => ({ ping: "pong" }))
	.use(usersRoute)
	.listen({ port: Bun.env.APP_PORT }, () =>
		console.log(`🦊 Elysia is running at :${Bun.env.APP_PORT}`),
	);
```

context.ts

```
import assert from "node:assert/strict";
import { PreludeHttpTransport } from "@litlinx/lib/infra/prelude/http";
import { Elysia } from "elysia";
import { getConnection } from "../../infra/data-src/pg/db";
import { migrateDb } from "../../infra/data-src/pg/migration";
import { UserRepository } from "../../repositories/user";
import type { Context } from "../../services/context";
import { handler } from "../../sqs/handler";
import { awsPlugin } from "./aws";
import { configPlugin } from "./config";

let isQueuePollingEnabled = false;

export const contextPlugin = new Elysia({ name: "contextPlugin" })
      .use(configPlugin)
      .use(awsPlugin)
      .derive({ as: "global" }, async ({ aws, config }) => {
            assert(aws && config);

            const dbCredentials = {
                  user: config.POSTGRES_USER,
                  port: config.POSTGRES_PORT,
                  db: config.POSTGRES_DB,
                  host: config.POSTGRES_HOST,
                  password: config.POSTGRES_PASSWORD,
            };

            const dbConnection = await getConnection(dbCredentials);

            await migrateDb(dbCredentials);

            const context = {
                  config,
                  userRepository: new UserRepository(dbConnection),
                  preludeTransport: new
PreludeHttpTransport(config.PRELUDE_BASE_URL),
                  aws,
            } satisfies Context;

            if (!isQueuePollingEnabled) {
                  await aws.sqs.pollQueue(config.AWS_SQS_QUEUE_URL, (message) =>
                        handler(JSON.parse(message), context),
                  );

                  isQueuePollingEnabled = true;
            }

            return { context };
      });
```

aws.ts

```
import assert from "node:assert";
import { setup } from "@litlinx/lib/infra/aws/setup";
import { pollQueue, sendMessage } from "@litlinx/lib/infra/aws/sqs";
import { Elysia } from "elysia";
import { configPlugin } from "./config";
```

```
export const awsPlugin = new Elysia({ name: "awsPlugin" })
      .use(configPlugin)
      .derive({ as: "scoped" }, ({ config }) => {
            assert(config);

            setup({ awsRegion: config.AWS_REGION });

            return {
                  aws: {
                        sqs: {
                              pollQueue,
                              sendMessage,
                        },
                  },
            };
      });
```

logger.ts

```
import { randomUUID } from "node:crypto";
import { logger, serializers } from "@bogeychan/elysia-logger";

export function createLoggerFactory() {
      return logger({
            serializers: {
                  ...serializers,
                  request: (request: Request) => {
                        const url = new URL(request.url);
                        const headers =
Object.fromEntries(request.headers.entries());

                        const requestData = {
                              id: request.headers.get("X-Request-ID") ??
randomUUID(),
                              path: url.pathname,
                              headers,
                              requestDate: new Date().toISOString(),
                              userAgent: headers["user-agent"] ?? "unknown",
                              ip:
                                    headers["x-forwarded-for"] ??
                                    request.headers.get("X-Real-IP") ??
                                    "unknown",
                        };

                        return {
                              ...requestData,
                              path: url.pathname,
                        };
                  },
            },
      });
}
```

config.ts

```
import { Elysia } from "elysia";
import { load } from "../../infra/config";
```

```
export const configPlugin = new Elysia({ name: "configPlugin" }).derive(
      { as: "scoped" },
      () => {
            return {
                  config: load(),
            };
      },
);

on-error.ts
import { TypeBoxError } from "@sinclair/typebox";
import type { Context } from "elysia";
import { match } from "ts-pattern";
import { ZodError } from "zod";
import { UserAlreadyExistsError } from "../../services/users/errors/user-already-
exists-error";
import { UserLoginError } from "../../services/users/errors/user-login-error";
import { UserRefreshError } from "../../services/users/errors/user-refresh-error";
import { UserUpdateError } from "../../services/users/errors/user-update-error";

const serviceLayerErrors = [
      UserAlreadyExistsError,
      UserRefreshError,
      UserUpdateError,
      UserLoginError,
] as const;

type ExtractEnumTypeFromArray<T extends readonly any[]> = T extends readonly [
      infer Head,
      ...infer Tail,
]
      ? Head extends { new (...args: any[]): infer O }
            ? O | ExtractEnumTypeFromArray<Tail>
            : never
      : never;

function getErrorResponse(
      statusCode: number,
      errorType: string,
      message: string,
      data?: any,
) {
      return {
            statusCode,
            error: errorType,
            message,
            data,
      };
}

export function onError(error: any, set: Context["set"]) {
      console.trace(error);
      const result = match(error)
            .when(
                  (e: any) =>
                        e instanceof ZodError ||
                        e instanceof TypeBoxError ||
```

```
                        e.code === "VALIDATION",
                  (e: any) => {
                        set.status = 422;

                        let data = {};

                        if (e.errors) {
                              data = e.errors.map((ve: { path: string; message:
string }) => ({
                                    path: ve.path,
                                    message: ve.message,
                              }));
                        }

                        return getErrorResponse(
                              422,
                              "VALIDATION_ERROR",
                              "Request validation error",
                              data,
                        );
                  },
            )
            .when(
                  (e) => serviceLayerErrors.some((ec) => e instanceof ec),
                  (matchedError: ExtractEnumTypeFromArray<typeof serviceLayerErrors>)
=> {
                        set.status = 422;
                        return getErrorResponse(
                              422,
                              "UNPROCESSABLE_ENTITY",
                              matchedError.message,
                        );
                  },
            )
            .otherwise(() => {
                  set.status = 500;
                  return getErrorResponse(
                        500,
                        "INTERNAL_SERVER_ERROR",
                        "Service is unavailable due to internal reasons",
                  );
            });

      return result;
}
```

users.ts

```
import Elysia, { t } from "elysia";
import { factory as createUserServiceFactory } from
"../../services/users/create/create";
import { CreateUserDtoIn } from "../../services/users/create/dto-in";
import { GetDtoIn } from "../../services/users/get/dto-in";
import { factory as getServiceFactory } from "../../services/users/get/get";
import { LoginDtoIn } from "../../services/users/login/dto-in";
import { factory as loginServiceFactory } from "../../services/users/login/login";
import { RefreshDtoIn } from "../../services/users/refresh/dto-in";
import { factory as refreshServiceFactory } from
```

```
"../../services/users/refresh/refresh";
import { UpdateUserDtoIn } from "../../services/users/update/dto-in";
import { factory as updateServiceFactory } from "../../services/users/update/update";
import { createJwtAuthGuard, decodeJwt } from "../guards/jwt-auth";
import { contextPlugin } from "../plugins/context";

export const usersRoute = new Elysia({ name: "usersRoute" })
      .use(contextPlugin)
      .decorate("createUserService", createUserServiceFactory())
      .decorate("loginService", loginServiceFactory())
      .decorate("updateService", updateServiceFactory())
      .decorate("refreshService", refreshServiceFactory())
      .decorate("getService", getServiceFactory())
      .group("users", (app) =>
            app
                  .post(
                        "/",
                        async ({ body, context, createUserService }) => {
                              const result = await createUserService({
                                    dto: new CreateUserDtoIn(
                                          body.username,
                                          body.password,
                                          body.name,
                                          body.birthday,
                                    ),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              body: t.Object({
                                    username: t.String({
                                          description: "Username for access",
                                          examples: "username",
                                    }),
                                    password: t.String({
                                          description: "Password for access",
                                          examples: "password",
                                    }),
                                    birthday: t.Optional(
                                          t.Date({
                                                description: "Birthday of user",
                                                examples: "2024-05-25T22:23:33.807Z",
                                          }),
                                    ),
                                    name: t.Optional(
                                          t.String({ description: "Name of user",
examples: "name" }),
                                    ),
                              }),
                        },
                  )
                  .post(
                        "/login",
                        async ({ body, context, loginService }) => {
                              const result = await loginService({
```

```
                                    dto: new LoginDtoIn(body.username,
body.password),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              body: t.Object({
                                    username: t.String({
                                          description: "Username for access",
                                          examples: "username",
                                    }),
                                    password: t.String({
                                          description: "Password for access",
                                          examples: "password",
                                    }),
                              }),
                        },
                  )
                  .guard((app) =>
                        app.use(createJwtAuthGuard(true)).post(
                              "/refresh",
                              async ({ body, context, accessId, refreshService }) =>
{
                                    const result = await refreshService({
                                          dto: new RefreshDtoIn(accessId as number,
body.refreshToken),
                                          context,
                                    });

                                    return result.toJSON();
                              },
                              {
                                    body: t.Object({
                                          refreshToken: t.String({
                                                description: "Refresh token",
                                                examples: "token",
                                          }),
                                    }),
                              },
                        ),
                  )
                  .guard((app) =>
                        app
                              .use(createJwtAuthGuard())
                              .patch(
                                    "/:userId",
                                    async ({ params, body, context, updateService })
=> {
                                          const result = await updateService({
                                          dto: new UpdateUserDtoIn(
                                                params.userId,
                                                body.username,
                                                body.name,
                                                body.birthday,
                                          ),
```

```
                                        context,
                                  });

                                  return result.toJSON();
                            },
                            {
                                  params: t.Object({
                                        userId: t.Numeric({
                                              description: "Id of user",
                                              examples: 1,
                                        }),
                                  }),
                                  body: t.Object({
                                        username: t.Optional(
                                              t.String({
                                                    description: "Username
for access",
                                                    examples: "username",
                                              }),
                                        ),
                                        birthday: t.Optional(
                                              t.Date({
                                                    description: "Birthday of
user",
                                                    examples: "2024-05-
25T22:23:33.807Z",
                                              }),
                                        ),
                                        name: t.Optional(
                                              t.String({ description: "Name
of user", examples: "name" }),
                                        ),
                                  }),
                            },
                      )
                      .get(
                            "/:userId",
                            async ({ params, context, getService, bearer })
=> {
                                  const result = await getService({
                                        dto: new GetDtoIn(params.userId,
decodeJwt(bearer!).accessId),
                                        context,
                                  });

                                  return result.toJSON();
                            },
                            {
                                  params: t.Object({
                                        userId: t.Numeric({
                                              description: "Id of user",
                                              examples: 1,
                                        }),
                                  }),
                            },
                      ),
                ),
```

```
	);

jwt-auth.ts
import { bearer } from "@elysiajs/bearer";
import { type Context, Elysia } from "elysia";

import { contextPlugin } from "../plugins/context";

function createErrorResponse(set: Context["set"]) {
	set.status = 401;
	set.headers["WWW-Authenticate"] =
		`Bearer realm='sign', error="invalid_request"`;

	return "Unauthorized";
}

function base64UrlDecode(base64Url: string): string {
	const sanitizedBase64Url = base64Url.replace(/-/g, "+").replace(/_/g, "/");

	const padding = "=".repeat((4 - (sanitizedBase64Url.length % 4)) % 4);

	const base64 = sanitizedBase64Url + padding;

	return Buffer.from(base64, "base64").toString("utf-8");
}

export function decodeJwt(token: string): { accessId: number } {
	const parts = token.split(".");

	const payload = parts[1];

	const decodedPayload = base64UrlDecode(payload);

	return JSON.parse(decodedPayload);
}

export function createJwtAuthGuard(ignoreExpiration = false) {
	return new Elysia({ name: "jwtAuthGuard" })
		.use(contextPlugin)
		.use(bearer())
		.derive({ as: "scoped" }, ({ bearer }) => {
			if (!bearer) return {};

			const { accessId } = decodeJwt(bearer);

			return { accessId };
		})
		.onBeforeHandle(
			{ as: "global" },
			async ({ set, bearer, accessId, context }) => {
				if (!(bearer && accessId && context)) return
createErrorResponse(set);

				try {
					await context.preludeTransport.verify(bearer,
ignoreExpiration);
				} catch {
```

```
                        return createErrorResponse(set);
                    }

                    const user = await context.userRepository.getUser({
                        externalAccessId: accessId,
                    });

                    if (!user) return createErrorResponse(set);
                },
            );
}

run.ts
import "./http/server";

services
make.ts
import type { ZodSchema } from "zod";
import type { Context } from "./context";

export type ActionDto<T extends Record<string, any> = Record<string, any>> = T;
export type ActionArg<
    T extends Record<string, any>,
    D extends ActionDto<T> = ActionDto<T>,
> = {
    dto: D;
    context: Context;
};
export type TAction<T extends Record<string, any>, R> = (
    arg: ActionArg<T>,
) => Promise<R>;

export function makeService<T extends Record<string, any>, R>(
    action: TAction<T, R>,
    validationSchema?: ZodSchema<T>,
) {
    const proxyHandler = {
        async apply(target: TAction<T, R>, thisArg: unknown, args:
[ActionArg<T>]) {
            if (validationSchema)
                args[0].dto = await validationSchema.parseAsync(args[0].dto);

            return target.apply(thisArg, args);
        },
    };

    return new Proxy(action, proxyHandler);
}

context.ts
import type { PollQueueFn, SendMessageFn } from "@litlinx/lib/infra/aws/sqs";
import type { PreludeHttpTransport } from "@litlinx/lib/infra/prelude/http";
import type { Config } from "../infra/config";
import type { UserRepository } from "../repositories/user";

export interface Context {
    readonly config: Config;
```

```
      readonly userRepository: UserRepository;
      readonly preludeTransport: PreludeHttpTransport;
      readonly aws: {
            sqs: {
                  pollQueue: PollQueueFn;
                  sendMessage: SendMessageFn;
            };
      };
}

dto-out.ts
export class UpdateUserDtoOut {
      constructor(
            public readonly id: number,
            public readonly username: string,
            public readonly name: string | null = null,
            public readonly birthday: string | null = null,
      ) {}

      toJSON() {
            return Object.assign({}, this);
      }
}

dto-in.ts
export class UpdateUserDtoIn {
      constructor(
            public readonly userId: number,
            public readonly username?: string,
            public readonly name?: string,
            public readonly birthday?: Date,
      ) {}
}

update.ts
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { UserUpdateError } from "../errors/user-update-error";
import type { UpdateUserDtoIn } from "./dto-in";
import { UpdateUserDtoOut } from "./dto-out";

async function update({
      dto,
      context,
}: {
      dto: UpdateUserDtoIn;
      context: Context;
}) {
      const user = await context.userRepository.getUser({
            id: dto.userId,
      });

      if (!user) throw new UserUpdateError("User not found");

      if (dto.username) {
            await context.preludeTransport.updateAccess(
```

```
                  user.externalAccessId,
                  dto.username,
            );

            user.username = dto.username;
      }

      if (dto.name) user.name = dto.name;
      if (dto.birthday) user.birthday = dto.birthday;

      await context.userRepository.updateFromEntity(user);

      return new UpdateUserDtoOut(
            user.id!,
            user.username,
            user.name,
            user.getFormattedBirthday(),
      );
}

export function factory() {
      return makeService(
            update,
            z.object({
                  userId:
z.coerce.number().int().positive().max(999_999_999_999_999),
                  name: z.string().max(64).optional(),
                  username: z.string().max(64).optional(),
                  birthday: z.coerce.date().optional(),
            }),
      );
}
```

refresh.ts

```
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { UserRefreshError } from "../errors/user-refresh-error";
import type { RefreshDtoIn } from "./dto-in";
import { RefreshDtoOut } from "./dto-out";

async function refresh({
      dto,
      context,
}: {
      dto: RefreshDtoIn;
      context: Context;
}) {
      const user = await context.userRepository.getUser({
            externalAccessId: dto.accessId,
      });

      if (!user) throw new UserRefreshError();

      const result = await context.preludeTransport.refresh(
            user.externalAccessId,
            dto.refreshToken,
```

```
	);

	return new RefreshDtoOut(result.jwtAccess, result.refreshToken);
}

export function factory() {
	return makeService(
		refresh,
		z.object({
			accessId:
z.coerce.number().int().positive().max(999_999_999_999_999),
			refreshToken: z.string().max(256),
		}),
	);
}

dto-out.ts
export class RefreshDtoOut {
	constructor(
		public readonly jwtAccess: string,
		public readonly refreshToken: string,
	) {}

	toJSON() {
		return Object.assign({}, this);
	}
}

dto-in.ts
export class RefreshDtoIn {
	constructor(
		public readonly accessId: number,
		public readonly refreshToken: string,
	) {}
}

get.ts
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { UserGetError } from "../errors/user-get-error";
import type { GetDtoIn } from "./dto-in";
import { GetDtoOut } from "./dto-out";

async function get({
	dto,
	context,
}: {
	dto: GetDtoIn;
	context: Context;
}) {
	const user = await context.userRepository.getUser({
		externalAccessId: dto.accessId,
		id: dto.userId,
	});

	if (!user) throw new UserGetError();
```

```
	return new GetDtoOut(
		user.id!,
		user.username,
		user.name,
		user.getFormattedBirthday(),
	);
}

export function factory() {
	return makeService(
		get,
		z.object({
			accessId: z.coerce
				.number()
				.int()
				.positive()
				.max(999_999_999_999_999)
				.optional(),
			userId: z.coerce
				.number()
				.int()
				.positive()
				.max(999_999_999_999_999)
				.optional(),
		}),
	);
}
```

dto-out.ts

```
export class GetDtoOut {
	constructor(
		public readonly id: number,
		public readonly username: string,
		public readonly name: string | null = null,
		public readonly birthday: string | null = null,
	) {}

	toJSON() {
		return Object.assign({}, this);
	}
}
```

dto-in.ts

```
export class GetDtoIn {
	constructor(
		public readonly userId?: number,
		public readonly accessId?: number,
	) {}
}
```

user-login-error.ts

```
export class UserLoginError extends Error {
	constructor(message = "User login error") {
		super(message);
		this.name = "UserLoginError";
	}
```

```
}

user-already-exists-error.ts
export class UserAlreadyExistsError extends Error {
	constructor(message = "User already exists") {
		super(message);
		this.name = "UserAlreadyExistsError";
	}
}

user-update-error.ts
export class UserUpdateError extends Error {
	constructor(message = "User update error") {
		super(message);
		this.name = "UserUpdateError";
	}
}

user-refresh-error.ts
export class UserRefreshError extends Error {
	constructor(message = "User refresh error") {
		super(message);
		this.name = "UserRefreshError";
	}
}

user-get-error.ts
export class UserGetError extends Error {
	constructor(message = "User not found") {
		super(message);
		this.name = "UserGetError";
	}
}

dto-out.ts
export class CreateUserDtoOut {
	constructor(
		public readonly id: number,
		public readonly username: string,
		public readonly jwtAccess: string,
		public readonly refreshToken: string,
		public readonly name: string | null = null,
		public readonly birthday: string | null = null,
	) {}

	toJSON() {
		return Object.assign({}, this);
	}
}

dto-in.ts
export class CreateUserDtoIn {
	constructor(
		public readonly username: string,
		public readonly password: string,
		public readonly name?: string,
		public readonly birthday?: Date,
```

```
	) {}
}

create.ts
import z from "zod";
import { User } from "../../../entities/user";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { UserAlreadyExistsError } from "../errors/user-already-exists-error";
import type { CreateUserDtoIn } from "./dto-in";
import { CreateUserDtoOut } from "./dto-out";

async function create({
	dto,
	context,
}: {
	dto: CreateUserDtoIn;
	context: Context;
}) {
	const existingUser = await context.userRepository.getUser({
		username: dto.username,
	});

	if (existingUser) throw new UserAlreadyExistsError();

	const result = await context.preludeTransport.createAccess(
		dto.username,
		dto.password,
	);

	const user = await User.from({
		id: null,
		name: dto.name,
		username: dto.username,
		birthday: dto.birthday,
		externalAccessId: result.id,
	});

	await context.userRepository.createFromEntity(user);

	return new CreateUserDtoOut(
		user.id!,
		user.username,
		result.jwtAccess,
		result.refreshToken,
		user.name,
		user.getFormattedBirthday(),
	);
}

export function factory() {
	return makeService(
		create,
		z.object({
			name: z.string().max(64).optional(),
			username: z.string().max(64),
			password: z.string().min(8).max(64),
```

```
                  birthday: z.coerce.date().optional(),
            }),
      );
}

login.ts
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { UserLoginError } from "../errors/user-login-error";
import type { LoginDtoIn } from "./dto-in";
import { LoginDtoOut } from "./dto-out";

async function login({
      dto,
      context,
}: {
      dto: LoginDtoIn;
      context: Context;
}) {
      const result = await context.preludeTransport.login(
            dto.username,
            dto.password,
      );

      const user = await context.userRepository.getUser({
            externalAccessId: result.id,
      });

      if (!user) throw new UserLoginError();

      return new LoginDtoOut(
            user.id!,
            user.username,
            result.jwtAccess,
            result.refreshToken,
            user.name,
            user.getFormattedBirthday(),
      );
}

export function factory() {
      return makeService(
            login,
            z.object({
                  username: z.string().max(64),
                  password: z.string().min(8).max(64),
            }),
      );
}

dto-out.ts
export class LoginDtoOut {
      constructor(
            public readonly id: number,
            public readonly username: string,
            public readonly jwtAccess: string,
```

```
            public readonly refreshToken: string,
            public readonly name: string | null = null,
            public readonly birthday: string | null = null,
      ) {}

      toJSON() {
            return Object.assign({}, this);
      }
}

dto-in.ts
export class LoginDtoIn {
      constructor(
            public readonly username: string,
            public readonly password: string,
      ) {}
}

entities
user.ts
import type { MaybeNumberId } from "./types/id";
import type { Optional } from "./types/optional";

export type Name = Optional<string>;
export type Birthday = Optional<Date>;

export type UserPayload = {
      id: MaybeNumberId;
      name: Name;
      username: string;
      birthday: Birthday;
      externalAccessId: number;
};

export class User {
      private _id: MaybeNumberId;
      private _name: Name;
      private _username: string;
      private _birthday: Birthday;
      private _externalAccessId: number;

      private constructor(payload: UserPayload) {
            this._id = payload.id ?? null;
            this._name = payload.name;
            this._username = payload.username;
            this._birthday = payload.birthday;
            this._externalAccessId = payload.externalAccessId;
      }

      static async from(payload: UserPayload): Promise<User> {
            return new User(payload);
      }

      get id() {
            return this._id;
      }
```

```
	set id(id: MaybeNumberId) {
		this._id = id;
	}

	get name(): Name {
		return this._name;
	}

	set name(name: string) {
		this._name = name;
	}

	get username() {
		return this._username;
	}

	set username(username: string) {
		this._username = username;
	}

	get birthday(): Birthday {
		return this._birthday;
	}

	getFormattedBirthday() {
		return this._birthday?.toUTCString() ?? null;
	}

	set birthday(birthday: Date) {
		this._birthday = birthday;
	}

	get externalAccessId() {
		return this._externalAccessId;
	}

	toPlainObject(): Readonly<Omit<UserPayload, "externalAccessId">> {
		return {
			id: this._id,
			name: this._name,
			username: this._username,
			birthday: this._birthday,
		};
	}
}
```

id.ts

```
import type { Optional } from "./optional";

type ToOptional<T> = Optional<T>;

export type MaybeNumberId = ToOptional<number>;
```

optional.ts

```
export type Optional<T> = T | null | undefined;
```

ДОДАТОК В
Код сервісу управління книжками

```
infra
config.ts
import { z } from "zod";

export interface Config {
	APP_PORT: number;

	POSTGRES_USER: string;
	POSTGRES_PASSWORD: string;
	POSTGRES_DB: string;
	POSTGRES_HOST: string;
	POSTGRES_PORT: number;

	AWS_ACCESS_KEY_ID: string;
	AWS_SECRET_ACCESS_KEY: string;
	AWS_REGION: string;
	AWS_SQS_QUEUE_URL: string;

	PRELUDE_BASE_URL: string;
	COMEDY_BASE_URL: string;
}

export class ConfigValidationError extends Error {
	private constructor(message: string) {
		super(message);
		this.name = "ConfigValidationError";
	}

	static from(errors: Record<string, string[]>) {
		const message = Object.entries(errors)
			.map(([field, errors]) => {
				return `${field}: ${errors.map((msg) =>
msg.toLowerCase()).join(", ")}`;
			})
			.join("\n");

		return new ConfigValidationError(message);
	}
}

export const configSchema = z.object({
	APP_PORT: z.coerce.number().int().positive(),

	POSTGRES_USER: z.string(),
	POSTGRES_PASSWORD: z.string(),
	POSTGRES_DB: z.string(),
	POSTGRES_HOST: z.string(),
	POSTGRES_PORT: z.coerce.number().int().positive(),

	AWS_ACCESS_KEY_ID: z.string(),
	AWS_SECRET_ACCESS_KEY: z.string(),
	AWS_REGION: z.string(),
	AWS_SQS_QUEUE_URL: z.string(),
```

```
      PRELUDE_BASE_URL: z.string(),
      COMEDY_BASE_URL: z.string(),
});

export function load(): Config {
      try {
            return configSchema.parse(Bun.env);
      } catch (error: any) {
            const formattedError = error.format() as Record<
                  string,
                  { _errors?: string[] }
            >;
            const transformedErrors: Record<string, string[]> = {};

            for (const [field, errorDetail] of Object.entries(formattedError)) {
                  if (Array.isArray(errorDetail._errors)) {
                        transformedErrors[field] = errorDetail._errors;
                  }
            }

            throw ConfigValidationError.from(transformedErrors);
      }
}
```

schema.ts

```
import { bigint, integer, pgTable, serial, varchar } from "drizzle-orm/pg-core";

export const books = pgTable("books", {
      id: serial("id").primaryKey(),
      name: varchar("name", { length: 128 }).notNull(),
      pages: integer("pages").notNull(),
      externalUserId: bigint("external_user_id", { mode: "number" }).notNull(),
});
```

migration.ts

```
import * as path from "node:path";

import { migrate } from "drizzle-orm/postgres-js/migrator";

import { getConnection } from "./db";
import type { ConnectionParams } from "./utils";

export async function migrateDb({
      user,
      password,
      db,
      host,
      port,
}: ConnectionParams): Promise<void> {
      const connection = await getConnection({
            user,
            password,
            db,
            host,
            port,
      });
```

```
	return migrate(connection, {
		migrationsFolder: path.join(process.cwd(), "migrations"),
	});
}

utils.ts
export type ConnectionParams = {
	user: string;
	password: string;
	db: string;
	host: string;
	port: number;
};

export function buildConnectionString({
	user,
	password,
	db,
	host,
	port,
}: ConnectionParams): string {
	return `postgres://${user}:${password}@${host}:${port}/${db}`;
}

db.ts
import { type PostgresJsDatabase, drizzle } from "drizzle-orm/postgres-js";
import postgres from "postgres";

import * as schema from "./schema";
import { type ConnectionParams, buildConnectionString } from "./utils";

export type Connection = PostgresJsDatabase<typeof schema>;

let _connection: Connection;

export async function getConnection({
	user,
	password,
	db,
	host,
	port,
}: ConnectionParams): Promise<Connection> {
	const connectionString = buildConnectionString({
		user,
		password,
		db,
		host,
		port,
	});

	if (!_connection)
		_connection = drizzle(postgres(connectionString, { max: 5 }), {
			schema,
			logger: false,
		});

	return _connection;
```

```
}

repositories
base.ts
import type { Connection } from "../infra/data-src/pg/db";

export abstract class BaseRepository {
      constructor(protected _connection: Connection) {}
}

book.ts
import { and, eq } from "drizzle-orm";
import { Book, type BookPayload } from "../entities/book";
import { books } from "../infra/data-src/pg/schema";
import { BaseRepository } from "./base";

export type SelectBook = typeof books.$inferSelect;

export class BookRepository extends BaseRepository {
      async createFromEntity(book: Book) {
            const result = await this._connection
                  .insert(books)
                  .values({
                        name: book.name,
                        pages: book.pages,
                        externalUserId: book.externalUserId,
                  })
                  .returning({ id: books.id })
                  .execute();

            book.id = result[0].id;

            return book;
      }

      async getBook(selectBy: Partial<SelectBook>) {
            const eqArray = [];
            for (const k of Object.keys(selectBy) as Array<keyof typeof selectBy>) {
                  eqArray.push(eq(books[k], selectBy[k]!));
            }

            const result = await this._connection
                  .select({
                        id: books.id,
                        name: books.name,
                        pages: books.pages,
                        externalUserId: books.externalUserId,
                  })
                  .from(books)
                  .where(and(...eqArray))
                  .limit(1)
                  .execute();

            const raw = result?.[0];

            if (!raw?.id) return null;
```

```
            return Book.from(raw as BookPayload);
      }

      async updateFromEntity(book: Book) {
            if (!book.id) return;

            await this._connection
                  .update(books)
                  .set({
                        name: book.name,
                        pages: book.pages,
                        externalUserId: book.externalUserId,
                  })
                  .where(eq(books.id, book.id))
                  .execute();
      }

      async deleteBookById(bookId: SelectBook["id"]) {
            await this._connection.delete(books).where(eq(books.id,
bookId)).execute();
      }
}

http
server.ts
import { swagger } from "@elysiajs/swagger";
import { Elysia } from "elysia";
import { onError } from "./hooks/on-error";
import { awsPlugin } from "./plugins/aws";
import { configPlugin } from "./plugins/config";
import { contextPlugin } from "./plugins/context";
import { createLoggerFactory } from "./plugins/logger";
import { booksRoute } from "./routes/books";

export const app = new Elysia()
      .use(swagger())
      .use(configPlugin)
      .use(awsPlugin)
      .use(contextPlugin)
      .use(createLoggerFactory())
      .onError(({ error, set }) => {
            onError(error, set);
      })
      .get("/ping", () => ({ ping: "pong" }))
      .use(booksRoute)
      .listen({ port: Bun.env.APP_PORT }, () =>
            console.log(`🦊 Elysia is running at :${Bun.env.APP_PORT}`),
      );

context.ts
import assert from "node:assert/strict";
import { PreludeHttpTransport } from "@litlinx/lib/infra/prelude/http";
import { Elysia } from "elysia";
import { getConnection } from "../../infra/data-src/pg/db";
import { migrateDb } from "../../infra/data-src/pg/migration";
import { BookRepository } from "../../repositories/book";
import type { Context } from "../../services/context";
```

```
import { awsPlugin } from "./aws";
import { configPlugin } from "./config";

export const contextPlugin = new Elysia({ name: "contextPlugin" })
      .use(configPlugin)
      .use(awsPlugin)
      .derive({ as: "scoped" }, async ({ aws, config }) => {
            assert(aws && config);

            const dbCredentials = {
                  user: config.POSTGRES_USER,
                  port: config.POSTGRES_PORT,
                  db: config.POSTGRES_DB,
                  host: config.POSTGRES_HOST,
                  password: config.POSTGRES_PASSWORD,
            };

            const dbConnection = await getConnection(dbCredentials);

            await migrateDb(dbCredentials);

            return {
                  context: {
                        config,
                        bookRepository: new BookRepository(dbConnection),
                        preludeHttpTransport: new
PreludeHttpTransport(config.PRELUDE_BASE_URL),
                        aws,
                  } satisfies Context,
            };
      });

aws.ts
import assert from "node:assert";
import { setup } from "@litlinx/lib/infra/aws/setup";
import {
      pollQueue,
      receiveMessage,
      sendMessage,
} from "@litlinx/lib/infra/aws/sqs";
import { Elysia } from "elysia";
import { configPlugin } from "./config";

export const awsPlugin = new Elysia({ name: "awsPlugin" })
      .use(configPlugin)
      .derive({ as: "scoped" }, ({ config }) => {
            assert(config);

            setup({ awsRegion: config.AWS_REGION });

            return {
                  aws: {
                        sqs: {
                              pollQueue,
                              sendMessage,
                              receiveMessage,
                        },
```

```
                  },
            };
      });

logger.ts
import { randomUUID } from "node:crypto";
import { logger, serializers } from "@bogeychan/elysia-logger";

export function createLoggerFactory() {
      return logger({
            serializers: {
                  ...serializers,
                  request: (request: Request) => {
                        const url = new URL(request.url);
                        const headers =
Object.fromEntries(request.headers.entries());

                        const requestData = {
                              id: request.headers.get("X-Request-ID") ??
randomUUID(),
                              path: url.pathname,
                              headers,
                              requestDate: new Date().toISOString(),
                              userAgent: headers["user-agent"] ?? "unknown",
                              ip:
                                    headers["x-forwarded-for"] ??
                                    request.headers.get("X-Real-IP") ??
                                    "unknown",
                        };

                        return {
                              ...requestData,
                              path: url.pathname,
                        };
                  },
            },
      });
}

config.ts
import { Elysia } from "elysia";
import { load } from "../../infra/config";

export const configPlugin = new Elysia({ name: "configPlugin" }).derive(
      { as: "scoped" },
      () => {
            return {
                  config: load(),
            };
      },
);

on-error.ts
import { TypeBoxError } from "@sinclair/typebox";
import type { Context } from "elysia";
import { match } from "ts-pattern";
import { ZodError } from "zod";
```

```
import { BookCreateError } from "../../services/books/errors/book-create-error";
import { BookGetError } from "../../services/books/errors/book-get-error";
import { BookRemoveError } from "../../services/books/errors/book-remove-error";
import { BookUpdateError } from "../../services/books/errors/book-update-error";

const serviceLayerErrors = [
      BookCreateError,
      BookGetError,
      BookRemoveError,
      BookUpdateError,
] as const;

type ExtractEnumTypeFromArray<T extends readonly any[]> = T extends readonly [
      infer Head,
      ...infer Tail,
]
      ? Head extends { new (...args: any[]): infer O }
            ? O | ExtractEnumTypeFromArray<Tail>
            : never
      : never;

function getErrorResponse(
      statusCode: number,
      errorType: string,
      message: string,
      data?: any,
) {
      return {
            statusCode,
            error: errorType,
            message,
            data,
      };
}

export function onError(error: any, set: Context["set"]) {
      const result = match(error)
            .when(
                  (e: any) =>
                        e instanceof ZodError ||
                        e instanceof TypeBoxError ||
                        e.code === "VALIDATION",
                  (e: any) => {
                        set.status = 422;

                        let data = {};

                        if (e.errors) {
                              data = e.errors.map((ve: { path: string; message:
string }) => ({
                                    path: ve.path,
                                    message: ve.message,
                              }));
                        }

                        return getErrorResponse(
                              422,
```

```
                              "VALIDATION_ERROR",
                              "Request validation error",
                              data,
                        );
                  },
            )
            .when(
                  (e) => serviceLayerErrors.some((ec) => e instanceof ec),
                  (matchedError: ExtractEnumTypeFromArray<typeof serviceLayerErrors>)
=> {
                        set.status = 422;
                        return getErrorResponse(
                              422,
                              "UNPROCESSABLE_ENTITY",
                              matchedError.message,
                        );
                  },
            )
            .otherwise(() => {
                  set.status = 500;
                  return getErrorResponse(
                        500,
                        "INTERNAL_SERVER_ERROR",
                        "Service is unavailable due to internal reasons",
                  );
            });

      return result;
}
```

books.ts

```
import Elysia, { t } from "elysia";
import { factory as createBookServiceFactory } from
"../../services/books/create/create";
import { CreateBookDtoIn } from "../../services/books/create/dto-in";
import { GetDtoIn } from "../../services/books/get/dto-in";
import { factory as getServiceFactory } from "../../services/books/get/get";
import { RemoveBookDtoIn } from "../../services/books/remove/dto-in";
import { factory as removeBookServiceFactory } from
"../../services/books/remove/remove";
import { UpdateBookDtoIn } from "../../services/books/update/dto-in";
import { factory as updateBookServiceFactory } from
"../../services/books/update/update";
import { createJwtAuthGuard, decodeJwt } from "../guards/jwt-auth";
import { contextPlugin } from "../plugins/context";

export const booksRoute = new Elysia({ name: "booksRoute" })
      .use(contextPlugin)
      .decorate("createBookService", createBookServiceFactory())
      .decorate("getService", getServiceFactory())
      .decorate("updateBookService", updateBookServiceFactory())
      .decorate("removeBookService", removeBookServiceFactory())
      .group("/books", (app) =>
            app.guard((app) =>
                  app
                        .use(createJwtAuthGuard())
                        .post(
```

```
                        "/",
                        async ({ body, context, createBookService, bearer }) =>
{
                              const result = await createBookService({
                                    dto: new CreateBookDtoIn(
                                          body.name,
                                          body.pages,
                                          body.userId,
                                          decodeJwt(bearer!).accessId,
                                    ),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              body: t.Object({
                                    name: t.String({
                                          description: "Name of book",
                                          examples: "Book Name",
                                    }),
                                    pages: t.Numeric({
                                          description: "Number of pages",
                                          examples: 100,
                                    }),
                                    userId: t.Numeric({
                                          description: "User id",
                                          examples: 1,
                                    }),
                              }),
                        },
                  )
                  .get(
                        "/:bookId",
                        async ({ params, context, getService, bearer }) => {
                              const result = await getService({
                                    dto: new GetDtoIn(params.bookId,
decodeJwt(bearer!).accessId),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              params: t.Object({
                                    bookId: t.Numeric({
                                          description: "Book id",
                                          examples: 100,
                                    }),
                              }),
                        },
                  )
                  .patch(
                        "/:bookId",
                        async ({ params, body, context, updateBookService,
bearer }) => {
                              const result = await updateBookService({
```

```
                                    dto: new UpdateBookDtoIn(
                                          params.bookId,
                                          decodeJwt(bearer!).accessId,
                                          body.name,
                                          body.pages,
                                    ),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              params: t.Object({
                                    bookId: t.Numeric({
                                          description: "Book id",
                                          examples: 100,
                                    }),
                              }),
                              body: t.Object({
                                    name: t.Optional(
                                          t.String({
                                                description: "Name of book",
                                                examples: "Book Name",
                                          }),
                                    ),
                                    pages: t.Optional(
                                          t.Numeric({
                                                description: "Number of pages",
                                                examples: 100,
                                          }),
                                    ),
                              }),
                        },
                  )
                  .delete(
                        "/:bookId",
                        async ({ params, context, removeBookService, bearer })
=> {
                              const result = await removeBookService({
                                    dto: new RemoveBookDtoIn(
                                          params.bookId,
                                          decodeJwt(bearer!).accessId,
                                    ),
                                    context,
                              });

                              return result.toJSON();
                        },
                        {
                              params: t.Object({
                                    bookId: t.Numeric({
                                          description: "Book id",
                                          examples: 100,
                                    }),
                              }),
                        },
                  ),
```

```
            ),
      );

jwt-auth.ts
import { bearer } from "@elysiajs/bearer";
import { type Context, Elysia } from "elysia";

import { contextPlugin } from "../plugins/context";

function createErrorResponse(set: Context["set"]) {
      set.status = 401;
      set.headers["WWW-Authenticate"] =
            `Bearer realm='sign', error="invalid_request"`;

      return "Unauthorized";
}

function base64UrlDecode(base64Url: string): string {
      const sanitizedBase64Url = base64Url.replace(/-/g, "+").replace(/_/g, "/");

      const padding = "=".repeat((4 - (sanitizedBase64Url.length % 4)) % 4);

      const base64 = sanitizedBase64Url + padding;

      return Buffer.from(base64, "base64").toString("utf-8");
}

export function decodeJwt(token: string): { accessId: number } {
      const parts = token.split(".");

      const payload = parts[1];

      const decodedPayload = base64UrlDecode(payload);

      return JSON.parse(decodedPayload);
}

export function createJwtAuthGuard(ignoreExpiration = false) {
      return new Elysia({ name: "jwtAuthGuard" })
            .use(contextPlugin)
            .use(bearer())
            .derive({ as: "scoped" }, ({ bearer }) => {
                  if (!bearer) return {};

                  const { accessId } = decodeJwt(bearer);

                  return { accessId };
            })
            .onBeforeHandle(
                  { as: "global" },
                  async ({ set, bearer, accessId, context }) => {
                        if (!(bearer && accessId && context)) return
createErrorResponse(set);

                        try {
                              await context.preludeHttpTransport.verify(bearer,
ignoreExpiration);
```

```
                        } catch {
                              return createErrorResponse(set);
                        }
                  },
            );
}

run.ts
import "./http/server";

services
make.ts
import type { ZodSchema } from "zod";
import type { Context } from "./context";

export type ActionDto<T extends Record<string, any> = Record<string, any>> = T;
export type ActionArg<
      T extends Record<string, any>,
      D extends ActionDto<T> = ActionDto<T>,
> = {
      dto: D;
      context: Context;
};
export type TAction<T extends Record<string, any>, R> = (
      arg: ActionArg<T>,
) => Promise<R>;

export function makeService<T extends Record<string, any>, R>(
      action: TAction<T, R>,
      validationSchema?: ZodSchema<T>,
) {
      const proxyHandler = {
            async apply(target: TAction<T, R>, thisArg: unknown, args:
[ActionArg<T>]) {
                  if (validationSchema)
                        args[0].dto = await validationSchema.parseAsync(args[0].dto);

                  return target.apply(thisArg, args);
            },
      };

      return new Proxy(action, proxyHandler);
}

context.ts
import type {
      PollQueueFn,
      ReceiveMessageFn,
      SendMessageFn,
} from "@litlinx/lib/infra/aws/sqs";
import type { PreludeHttpTransport } from "@litlinx/lib/infra/prelude/http";
import type { Config } from "../infra/config";
import type { BookRepository } from "../repositories/book";

export interface Context {
      readonly config: Config;
      readonly bookRepository: BookRepository;
```

```
      readonly preludeHttpTransport: PreludeHttpTransport;
      readonly aws: {
            sqs: {
                  pollQueue: PollQueueFn;
                  sendMessage: SendMessageFn;
                  receiveMessage: ReceiveMessageFn;
            };
      };
}

dto-out.ts
export class RemoveBookDtoOut {
      constructor(
            public readonly id: number,
            public readonly name: string,
            public readonly pages: number,
            public readonly userId: number,
      ) {}

      toJSON() {
            return Object.assign({}, this);
      }
}

remove.ts
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { BookRemoveError } from "../errors/book-remove-error";
import type { RemoveBookDtoIn } from "./dto-in";
import { RemoveBookDtoOut } from "./dto-out";

async function remove({
      dto,
      context,
}: {
      dto: RemoveBookDtoIn;
      context: Context;
}) {
      await context.aws.sqs.sendMessage(context.config.AWS_SQS_QUEUE_URL, {
            action: "getUser",
            args: { accessId: dto.accessId },
      });

      const [message] = await context.aws.sqs.receiveMessage(
            context.config.AWS_SQS_QUEUE_URL,
      );

      if (!message?.id) throw new BookRemoveError("User not found");

      const book = await context.bookRepository.getBook({
            id: dto.bookId,
            externalUserId: message.id,
      });

      if (!book) throw new BookRemoveError("Book not found");
```

```
	await context.bookRepository.deleteBookById(book.id!);

	return new RemoveBookDtoOut(
		book.id!,
		book.name,
		book.pages,
		book.externalUserId,
	);
}

export function factory() {
	return makeService(
		remove,
		z.object({
			bookId:
z.coerce.number().int().positive().max(999_999_999_999_999),
			accessId:
z.coerce.number().int().positive().max(999_999_999_999_999),
		}),
	);
}
```

dto-in.ts

```
export class RemoveBookDtoIn {
	constructor(
		public readonly bookId: number,
		public readonly accessId: number,
	) {}
}
```

dto-out.ts

```
export class UpdateBookDtoOut {
	constructor(
		public readonly id: number,
		public readonly name: string,
		public readonly pages: number,
		public readonly userId: number,
	) {}

	toJSON() {
		return Object.assign({}, this);
	}
}
```

dto-in.ts

```
export class UpdateBookDtoIn {
	constructor(
		public readonly bookId: number,
		public readonly accessId: number,
		public readonly name?: string,
		public readonly pages?: number,
	) {}
}
```

update.ts

```
import z from "zod";
import type { Context } from "../../context";
```

```
import { makeService } from "../../make";
import { BookUpdateError } from "../errors/book-update-error";
import type { UpdateBookDtoIn } from "./dto-in";
import { UpdateBookDtoOut } from "./dto-out";

async function update({
      dto,
      context,
}: {
      dto: UpdateBookDtoIn;
      context: Context;
}) {
      await context.aws.sqs.sendMessage(context.config.AWS_SQS_QUEUE_URL, {
            action: "getUser",
            args: { accessId: dto.accessId },
      });

      const [message] = await context.aws.sqs.receiveMessage(
            context.config.AWS_SQS_QUEUE_URL,
      );

      if (!message?.id) throw new BookUpdateError("User not found");

      const book = await context.bookRepository.getBook({
            id: dto.bookId,
            externalUserId: message.id,
      });

      if (!book) throw new BookUpdateError("Book not found");

      if (dto.name) book.name = dto.name;
      if (dto.pages) book.pages = dto.pages;

      await context.bookRepository.updateFromEntity(book);

      return new UpdateBookDtoOut(
            book.id!,
            book.name,
            book.pages,
            book.externalUserId,
      );
}

export function factory() {
      return makeService(
            update,
            z.object({
                  name: z.string().max(128).optional(),
                  pages: z.coerce.number().int().positive().max(100_000).optional(),
                  bookId:
z.coerce.number().int().positive().max(999_999_999_999_999),
                  accessId:
z.coerce.number().int().positive().max(999_999_999_999_999),
            }),
      );
}
```

```
get.ts
import z from "zod";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { BookCreateError } from "../errors/book-create-error";
import { BookGetError } from "../errors/book-get-error";
import type { GetDtoIn } from "./dto-in";
import { GetDtoOut } from "./dto-out";

async function get({
      dto,
      context,
}: {
      dto: GetDtoIn;
      context: Context;
}) {
      await context.aws.sqs.sendMessage(context.config.AWS_SQS_QUEUE_URL, {
            action: "getUser",
            args: { accessId: dto.accessId },
      });

      const [message] = await context.aws.sqs.receiveMessage(
            context.config.AWS_SQS_QUEUE_URL,
      );

      if (!message?.id) throw new BookCreateError("User not found");

      const book = await context.bookRepository.getBook({
            id: dto.bookId,
            externalUserId: message.id,
      });

      if (!book) throw new BookGetError("Book not found");

      return new GetDtoOut(book.id!, book.name, book.pages, book.externalUserId);
}

export function factory() {
      return makeService(
            get,
            z.object({
                  bookId:
z.coerce.number().int().positive().max(999_999_999_999_999),
                  accessId:
z.coerce.number().int().positive().max(999_999_999_999_999),
            }),
      );
}

dto-out.ts
export class GetDtoOut {
      constructor(
            public readonly id: number,
            public readonly name: string,
            public readonly pages: number,
            public readonly userId: number,
      ) {}
```

```
	toJSON() {
		return Object.assign({}, this);
	}
}

dto-in.ts
export class GetDtoIn {
	constructor(
		public readonly bookId: number,
		public readonly accessId: number,
	) {}
}

book-create-error.ts
export class BookCreateError extends Error {
	constructor(message = "Book create error") {
		super(message);
		this.name = "BookCreateError";
	}
}

book-update-error.ts
export class BookUpdateError extends Error {
	constructor(message = "Book update error") {
		super(message);
		this.name = "BookUpdateError";
	}
}

book-remove-error.ts
export class BookRemoveError extends Error {
	constructor(message = "Book remove error") {
		super(message);
		this.name = "BookRemoveError";
	}
}

book-get-error.ts
export class BookGetError extends Error {
	constructor(message = "Book get error") {
		super(message);
		this.name = "BookGetError";
	}
}

dto-out.ts
export class CreateBookDtoOut {
	constructor(
		public readonly id: number,
		public readonly name: string,
		public readonly pages: number,
		public readonly userId: number,
	) {}

	toJSON() {
		return Object.assign({}, this);
```

```
	}
}

dto-in.ts
export class CreateBookDtoIn {
	constructor(
		public readonly name: string,
		public readonly pages: number,
		public readonly userId: number,
		public readonly accessId: number,
	) {}
}

create.ts
import z from "zod";
import { Book } from "../../../entities/book";
import type { Context } from "../../context";
import { makeService } from "../../make";
import { BookCreateError } from "../errors/book-create-error";
import type { CreateBookDtoIn } from "./dto-in";
import { CreateBookDtoOut } from "./dto-out";

async function create({
	dto,
	context,
}: {
	dto: CreateBookDtoIn;
	context: Context;
}) {
	await context.aws.sqs.sendMessage(context.config.AWS_SQS_QUEUE_URL, {
		action: "getUser",
		args: { userId: dto.userId, accessId: dto.accessId },
	});

	const [message] = await context.aws.sqs.receiveMessage(
		context.config.AWS_SQS_QUEUE_URL,
	);

	if (!message?.id) throw new BookCreateError("User not found");

	const book = await Book.from({
		id: null,
		name: dto.name,
		pages: dto.pages,
		externalUserId: dto.userId,
	});

	await context.bookRepository.createFromEntity(book);

	return new CreateBookDtoOut(
		book.id!,
		book.name,
		book.pages,
		book.externalUserId,
	);
}
```

```
export function factory() {
	return makeService(
		create,
		z.object({
			name: z.string().max(128),
			pages: z.coerce.number().int().positive().max(100_000),
			userId:
z.coerce.number().int().positive().max(999_999_999_999_999),
			accessId:
z.coerce.number().int().positive().max(999_999_999_999_999),
		}),
	);
}

entities
book.ts
import type { MaybeNumberId } from "./types/id";

export type BookPayload = {
	id: MaybeNumberId;
	name: string;
	pages: number;
	externalUserId: number;
};

export class Book {
	private _id: MaybeNumberId;
	private _name: string;
	private _pages: number;
	private _externalUserId: number;

	private constructor(payload: BookPayload) {
		this._id = payload.id ?? null;
		this._name = payload.name;
		this._pages = payload.pages;
		this._externalUserId = payload.externalUserId;
	}

	static async from(payload: BookPayload): Promise<Book> {
		return new Book(payload);
	}

	get id() {
		return this._id;
	}

	set id(id: MaybeNumberId) {
		this._id = id;
	}

	get name(): string {
		return this._name;
	}

	set name(name: string) {
		this._name = name;
	}
```

```
	get pages(): number {
		return this._pages;
	}

	set pages(pages: number) {
		this._pages = pages;
	}

	get externalUserId(): number {
		return this._externalUserId;
	}

	set externalUserId(externalUserId: number) {
		this._externalUserId = externalUserId;
	}

	toPlainObject(): Readonly<Omit<BookPayload, "externalUserId">> {
		return {
			id: this._id,
			name: this._name,
			pages: this._pages,
		};
	}
}
```

id.ts

```
import type { Optional } from "./optional";

type ToOptional<T> = Optional<T>;

export type MaybeNumberId = ToOptional<number>;
```

optional.ts

```
export type Optional<T> = T | null | undefined;
```

ДОДАТОК Г
Приклад Makefile

```
DOCKER_REGISTRY ?= public.ecr.aws/qfsdf3413
DOCKER_USERNAME ?= oxb4f
APP_NAME ?= plot
GIT_BRANCH ?= $(shell git rev-parse --abbrev-ref HEAD)
GIT_COMMIT ?= $(shell git rev-parse --short HEAD)
DOCKER_TAG ?= ${GIT_BRANCH}-${GIT_COMMIT}
APP_IMAGE ?= ${DOCKER_REGISTRY}/${DOCKER_USERNAME}/${APP_NAME}:${DOCKER_TAG}
DOCKER_FILE ?= build/Dockerfile
DOCKER_COMPOSE ?= deployment/docker-compose.dev.yml
ENV_FILE ?= .env.dev

RN := $(shell echo $$(( RANDOM % 9 + 1 ))`seq -w 0 8 | shuf | tr -d '\n' | head -c
9`)
RS := $(shell openssl rand -base64 10 | tr -dc 'a-zA-Z0-9' | head -c 10)

# @target build
PULL ?= false ## Pull the latest base image before building. Default is false.
NO_CACHE ?= false ## Build the image without using cache. Default is false.
TARGET ?= dev ## Build target for docker: dev, prod or test

# @target run
DETACH ?= false ## Run docker-compose in detached mode. Default is false.

BLUE := \033[1;34m
GREEN := \033[0;32m
WHITE := \033[0;97m
RESET := \033[0m

.PHONY: build run stop help hurl test-unit

build: ## Build the Docker image for the application
ifeq ($(PULL),true)
      PULL_FLAG=--pull
else
      PULL_FLAG=
endif
ifeq ($(NO_CACHE),true)
      NO_CACHE_FLAG=--no-cache
else
      NO_CACHE_FLAG=
endif
      DOCKER_BUILDKIT=1 docker build $(PULL_FLAG) $(NO_CACHE_FLAG) --tag ${APP_IMAGE}
--file ${DOCKER_FILE} --target ${TARGET} ..

run: stop ## Start the application using Docker Compose
ifeq ($(DETACH),true)
      DETACH_FLAG=-d
else
      DETACH_FLAG=
endif
      APP_IMAGE=${APP_IMAGE} ENV_FILE=${ENV_FILE} docker-compose --file
${DOCKER_COMPOSE} --project-name ${APP_NAME} up $(DETACH_FLAG)
```

```
stop: ## Stop the application using Docker Compose
	APP_IMAGE=${APP_IMAGE} ENV_FILE=${ENV_FILE} docker-compose --file
${DOCKER_COMPOSE} down --remove-orphans --volumes

hurl: ## Run hurl tests against the application
	@hurl \
		--variables-file ./hurl/vars \
		--variable rn=$(RN) \
		--variable rs=$(RS) \
		--file-root ./hurl \
		--insecure \
		--error-format long \
		--test \
		--glob "./hurl/*.hurl"

test-unit: stop ## Run unit tests against the application
	APP_IMAGE=${APP_IMAGE} ENV_FILE=${ENV_FILE} docker-compose --file
${DOCKER_COMPOSE} --project-name ${APP_NAME} up --detach

	docker-compose --file ${DOCKER_COMPOSE} --project-name ${APP_NAME} exec --tty -
-interactive app sh -c "pnpm test:unit"

help: ## Display this help message
	@echo "Usage: make [target] [FLAGS]"
	@echo ""
	@echo "$(BLUE)Targets:$(RESET)"
	@awk 'BEGIN {FS = ":.*?## "; print "$(GREEN)"} /^[a-zA-Z_-]+:.*?## / {printf "
%-10s $(WHITE)%s$(RESET)\n", $$1, $$2}' $(MAKEFILE_LIST)
	@echo ""
	@echo "$(BLUE)Flags:$(RESET)"
	@awk 'BEGIN {target=""} \
		/^# @target/ {target=$$3; next} \
		/##/ && target!="" { \
			split($$1, arr, "?"); var=arr[1]; \
			desc=substr($$0, index($$0, "##") + 3); \
			if (!printed[target]++) { \
				printf "\n${GREEN}Flags for '" target "' target:${RESET}\n";
\
			} \
			printf "  ${BLUE}%s=true${RESET}   - ${WHITE}%s${RESET}\n", var,
desc \
		} \
		/^$$/ {target=""}' $(MAKEFILE_LIST)
```

ДОДАТОК Д
Приклад Dockerfile

```
### base
FROM oven/bun:alpine AS base

RUN mkdir -p /app
WORKDIR /app

RUN mkdir -p plot

COPY plot/package.json ./plot/package.json

COPY plot/src/ plot/src

COPY lib lib
COPY comedy comedy
COPY package.json ./
COPY bun.lockb ./
COPY tsconfig.json ./

RUN bun install


### run
FROM base AS run

RUN mkdir -p /app
WORKDIR /app

COPY  --from=base /app/. .
COPY  --from=base package*.json ./
COPY plot/tsconfig*.json ./
COPY plot/drizzle.config.ts ./
COPY plot/migrations ./

ENTRYPOINT []


### test
FROM run AS test

RUN mkdir -p /app
WORKDIR /app

COPY --from=run /app/. .
COPY --from=base /app/package*.json ./

RUN bun install

ENTRYPOINT []
```

ДОДАТОК Е
Приклад docker-compose файлу

```
services:
  app:
    image: ${APP_IMAGE:-}
    command: sh -c 'cd plot && bun dev'
    restart: always
    container_name: plot
    env_file:
      - ${ENV_FILE:-.env}
    ports:
      - "${APP_PORT:-8082}:${APP_PORT:-8082}"
    volumes:
      - ../src:/app/plot/src
      - ../migrations:/app/plot/migrations
    depends_on:
      - db
    expose:
      - "${APP_PORT:-8082}"

  db:
    image: postgres:15-alpine
    env_file:
      - ${ENV_FILE:-.env}
    ports:
      - "${POSTGRES_PORT:-5434}:${POSTGRES_PORT:-5434}"
    expose:
      - "${POSTGRES_PORT:-5434}"
    command: -p "${POSTGRES_PORT:-5434}"
```

ДОДАТОК Є
Приклад Hurl тесту

```
POST {{API_USERS_BASE}}/users
Content-Type: application/json
{
    "username": "05{{rn}}{{rs}}book1",
    "password": "05{{rn}}{{rs}}pass1",
    "name": "Test name",
    "birthday": "2024-02-23"
}

HTTP 200
[Asserts]
jsonpath "$.id" isInteger
jsonpath "$.username" == "05{{rn}}{{rs}}book1"
jsonpath "$.jwtAccess" isString
jsonpath "$.refreshToken" isString
jsonpath "$.name" isString
jsonpath "$.birthday" exists
[Captures]
userId: jsonpath "$.id"
jwt: jsonpath "$.jwtAccess"

POST {{API_BASE}}/books
Authorization: Bearer {{jwt}}
Content-Type: application/json
{
    "name": "Test book",
    "pages": 100,
    "userId": {{userId}}
}

HTTP 200
[Asserts]
jsonpath "$.id" isInteger
jsonpath "$.name" == "Test book"
jsonpath "$.pages" == 100

POST {{API_BASE}}/books
Authorization: Bearer {{jwt}}
Content-Type: application/json
{
    "name": "Test book 1",
    "pages": 100,
    "userId": {{userId}}
}

HTTP 200
[Asserts]
jsonpath "$.id" isInteger
jsonpath "$.name" == "Test book 1"
jsonpath "$.pages" == 100

POST {{API_BASE}}/books
Authorization: Bearer {{jwt}}
```

```
Content-Type: application/json
{
    "name": "Test book 2",
    "pages": 100,
    "userId": {{userId}}
}

HTTP 200
[Asserts]
jsonpath "$.id" isInteger
jsonpath "$.name" == "Test book 2"
jsonpath "$.pages" == 100

GET {{API_BASE}}/books
Authorization: Bearer {{jwt}}
Content-Type: application/json
{
    "name": "Test book 2",
    "pages": 100,
    "userId": {{userId}}
}

HTTP 200
[Asserts]
jsonpath "$.count" == 3
jsonpath "$.data[*].name" includes "Test book"
jsonpath "$.data[*].name" includes "Test book 1"
jsonpath "$.data[*].name" includes "Test book 2"
```

ДОДАТОК Ж
Приклад налаштування деплою застосунку до Amazon ECS

```
AWS_REGION = us-east-1
DOCKER_CONTEXT = testecscontext
DEFAULT_CONTEXT = desktop-linux

ECR_REPO_URL = public.ecr.aws/111111

GIT_BRANCH = $(shell git rev-parse --abbrev-ref HEAD)
GIT_COMMIT = $(shell git rev-parse --short HEAD)

PRELUDE_IMAGE = $(ECR_REPO_URL)/oxb4f/prelude:$(GIT_BRANCH)-$(GIT_COMMIT)
COMEDY_IMAGE = $(ECR_REPO_URL)/oxb4f/comedy:$(GIT_BRANCH)-$(GIT_COMMIT)
PLOT_IMAGE = $(ECR_REPO_URL)/oxb4f/plot:$(GIT_BRANCH)-$(GIT_COMMIT)

ECS_CLUSTER_NAME = TestCluster

DIRS = prelude comedy plot

.PHONY: all
all: push deploy

.PHONY: create-context
create-context:
	@docker context create ecs $(DOCKER_CONTEXT) --from-env || echo "Context
$(DOCKER_CONTEXT) already exists"

.PHONY: use-context
use-context: create-context
	@docker context use $(DOCKER_CONTEXT)

.PHONY: use-default-context
use-default-context:
	@docker context use $(DEFAULT_CONTEXT)

.PHONY: login
login:
	@aws ecr-public get-login-password --region $(AWS_REGION) | docker login --
username AWS --password-stdin $(ECR_REPO_URL)

.PHONY: create-repos
create-repos:
	@echo "Checking and creating repositories if they do not exist..."
	@aws ecr-public describe-repositories --region $(AWS_REGION) --repository-names
oxb4f/prelude || \
	(aws ecr-public create-repository --region $(AWS_REGION) --repository-name
oxb4f/prelude || true)
	@aws ecr-public describe-repositories --region $(AWS_REGION) --repository-names
oxb4f/comedy || \
	(aws ecr-public create-repository --region $(AWS_REGION) --repository-name
oxb4f/comedy || true)
	@aws ecr-public describe-repositories --region $(AWS_REGION) --repository-names
oxb4f/plot || \
	(aws ecr-public create-repository --region $(AWS_REGION) --repository-name
oxb4f/plot || true)
```

```
.PHONY: build-all
build-all:
	@for dir in $(DIRS); do \
		$(MAKE) -C $$dir build TARGET=run; \
	done

.PHONY: push
push: login create-repos build-all
	@docker push $(PRELUDE_IMAGE)
	@docker push $(COMEDY_IMAGE)
	@docker push $(PLOT_IMAGE)

.PHONY: deploy
deploy: use-context
	@PRELUDE_IMAGE=$(PRELUDE_IMAGE) COMEDY_IMAGE=$(COMEDY_IMAGE)
PLOT_IMAGE=$(PLOT_IMAGE) docker compose -f docker-compose.yml up -d
```