

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
ДЕРЖАВНИЙ ЗАКЛАД  
«ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Навчально-науковий інститут математики  
та інформаційних технологій

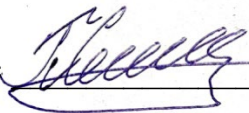
Кафедра інформаційних технологій та систем

**Попков Євген Васильович**

**Архітектурні підходи до реалізації мікросервісів у serverless-середовищах**

**кваліфікаційна робота  
здобувача вищої освіти другого (магістерського) рівня  
освітньої програми «Мультимедійні системи»  
за спеціальністю 121 „Інженерія програмного забезпечення”**

Особистий підпис



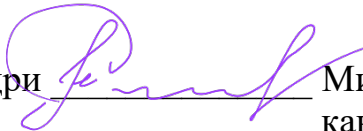
Євген ПОПКОВ

Науковий керівник



Світлана ПЕРЕЯСЛАВСЬКА,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій та  
систем

Завідувач кафедри



Микола СЕМЕНОВ,  
кандидат педагогічних наук,  
доцент кафедри інформаційних  
технологій та систем

## АНОТАЦІЯ

**Попков Є.В.**

**Тема:** Архітектурні підходи до реалізації мікросервісів у serverless-середовищах

**Спеціальність:** 121 «Інженерія програмного забезпечення».

**Установа:** ДЗ ЛНУ імені Тараса Шевченка, 2025р.

**Магістерська робота містить:** 161 с., 17 рис., 7 табл., 1 додат., 32 джерел.

**Об'єкт дослідження** – мікросервісна архітектура розподілених програмних систем.

**Предмет дослідження** – вплив serverless-середовищ на архітектуру мікросервісних систем.

**Мета роботи** – аналіз архітектурних підходів до реалізації мікросервісів у serverless-середовищах.

**Методи дослідження.** *Загальнонаукові:* аналіз еволюції архітектур програмних систем і serverless-підходу, аналіз архітектурних рішень мікросервісних систем, синтез теоретичних положень щодо поєднання мікросервісної архітектури з serverless-середовищами, порівняння традиційних і serverless-підходів до реалізації розподілених програмних систем, узагальнення результатів теоретичного та практичного аналізу; *методи інформаційного пошуку:* бібліографічний пошук, аналіз наукових і спеціалізованих джерел, реферування та цитування інформаційних матеріалів з тематики дослідження.

**Результати роботи.** Проаналізовано архітектурні підходи до реалізації мікросервісів у serverless-середовищах. Розглянуто еволюцію архітектур програмних систем та узагальнено особливості впливу serverless-середовищ на архітектурні рішення мікросервісних систем. Досліджено актуальний стан наукових підходів до використання serverless-технологій у контексті мікросервісної архітектури та виокремлено ключові особливості і обмеження цього підходу. На прикладі демонстраційної системи проілюстровано можливість застосування мікросервісної моделі у serverless-середовищі та

проаналізовано відповідність обраних архітектурних рішень базовим принципам побудови розподілених систем.

**Ключові слова:** мікросервісна архітектура, serverless, хмарні обчислення, розподілені програмні системи.

## ABSTRACT

**Popkov Y.P.**

**Topic:** Architectural Approaches to the Implementation of Microservices in Serverless Environments

**Specialty:** 121 “Software Engineering”.

**Institution:** Taras Shevchenko Luhansk National University, 2025.

**Master’s thesis contains:** 161 pages, 17 figures, 7 tables, 1 appendix, 32 sources.

**Object of research:** microservices architecture of distributed software systems.

**Subject of research:** the impact of serverless environments on the architecture of microservices systems.

**Purpose of the work:** analysis of architectural approaches to the implementation of microservices in serverless environments.

**Research methods.** General scientific methods: analysis of the evolution of software system architectures and the serverless approach, analysis of architectural solutions of microservice-based systems, synthesis of theoretical principles regarding the integration of microservice architecture with serverless environments, comparison of traditional and serverless approaches to the implementation of distributed software systems, generalization of the results of theoretical and practical analysis; information retrieval methods: bibliographic search, analysis of scientific and specialized sources, abstracting and citation of information materials related to the research topic.

**Results of the work.** Architectural approaches to the implementation of microservices in serverless environments have been analyzed. The evolution of software system architectures has been examined, and the features of the impact of serverless environments on architectural decisions of microservice-based systems have been generalized. The current state of scientific approaches to the use of serverless technologies in the context of microservice architecture has been investigated, and the key characteristics and limitations of this approach have been identified. Using a demonstration system as an example, the applicability of the microservice model in a serverless environment has been illustrated, and the

compliance of the chosen architectural solutions with the fundamental principles of building distributed systems has been analyzed.

**Keywords:** microservices architecture, serverless, cloud computing, distributed software systems.

## ЗМІСТ

ВСТУП .....	9
РОЗДІЛ 1 ЕВОЛЮЦІЯ АРХІТЕКТУР ПРОГРАМНИХ СИСТЕМ .....	14
1.1. Монолітна архітектура .....	15
1.2. Модульні та компонентні підходи .....	16
1.3. Перехід до розподілених систем: клієнт–серверна та багаторівнева моделі .....	17
1.4. Сервісно-орієнтована архітектура (SOA) .....	18
1.5. Перехід до мікросервісної архітектури .....	20
Висновки до розділу 1 .....	24
РОЗДІЛ 2 SERVERLESS ЯК СЕРЕДОВИЩЕ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ ВЗАЄМОДІЇ .....	25
2.1. Актуальний стан досліджень .....	26
2.2. Serverless як середовище реалізації мікросервісів .....	28
2.3. Реалізація принципів мікросервісної архітектури у serverless-моделі ..	29
2.3.1. Організація коду та інфраструктурне управління .....	29
2.3.2. Організація та гранулярність функцій у межах мікросервісу .....	30
2.3.3. Інтеграційна інфраструктура serverless-середовища .....	32
Висновки до розділу 2 .....	34
РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ У SERVERLESS-СЕРЕДОВИЩІ .....	35
3.1. Постановка задачі та підхід до реалізації .....	35
3.2. Доменна модель та межі мікросервісів .....	36
3.3. Архітектурна організація мікросервісів у serverless-середовищі .....	40
3.3.1. Infrastructure as Code як основа організації інфраструктури .....	40
3.3.2. Поділ інфраструктури на спільний та сервісні шари .....	42
3.3.3. Управління конфігурацією та сервісною адресацією .....	45
3.4. Загальна архітектура взаємодії мікросервісів .....	47
3.5. Синхронна взаємодія між мікросервісами .....	48
3.6. Асинхронна модель взаємодії .....	50
3.6.1. Технічна модель асинхронної взаємодії .....	50
3.6.2. Організація подієвих каналів у хмарному середовищі .....	51

3.6.3. Зв'язування подієвих каналів та функцій обробки .....	53
3.7. Узгодженість даних та надійність обробки.....	55
Висновки до розділу 3 .....	59
ВИСНОВКИ.....	60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	62
ДОДАТКИ.....	66
Додаток А. Вихідний код додатку.....	66

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

API	–	Application Programming Interface
AWS	–	Amazon Web Services
HTTP	–	HyperText Transfer Protocol
IaC	–	Infrastructure as Code
IAM	–	Identity and Access Management
REST	–	Representational State Transfer
SNS	–	Simple Notification Service
SQS	–	Simple Queue Service
SOA	–	Service-Oriented Architecture
SOAP	–	Simple Object Access Protocol
XML	–	eXtensible Markup Language
RPC	–	Remote Procedure Call
gRPC	–	Google Remote Procedure Call
DDD	–	Domain-Driven Design
FaaS	–	Function as a Service



## ВСТУП

**Актуальність дослідження.** Сучасний етап розвитку програмної інженерії характеризується зростанням масштабів програмних систем, а також підвищенням вимог до їх гнучкості, масштабованості та надійності. Із зростанням навантаження, кількості користувачів і функціональних можливостей систем дедалі ширше застосовуються підходи до побудови програмних систем, за яких прикладна логіка та обчислювальні ресурси розосереджуються між окремими компонентами. Водночас така розосередженість актуалізує низку архітектурних викликів, пов'язаних із координацією компонентів, керуванням станом та забезпеченням узгодженості поведінки системи в цілому.

Паралельно з цими процесами набули поширення хмарні технології, які запропонували нові моделі розгортання та експлуатації програмних рішень. Подальшим етапом цієї еволюції стало формування serverless-підходу, за якого виконання прикладної логіки відбувається з мінімальним залученням розробника до керування інфраструктурними ресурсами. Така модель змінює традиційні уявлення про життєвий цикл програмних компонентів і впливає на підходи до організації архітектури розподілених систем.

На сьогодні питання архітектури розподілених програмних систем, мікросервісного підходу та serverless-середовищ привертають значну увагу як наукової спільноти, так і практиків галузі. Вагомий внесок у розвиток підходів до мікросервісної архітектури та систематизацію архітектурних рішень зроблено у працях відомих дослідників і практиків, зокрема М. Richards, N. Ford та S. Newman. Паралельно, у вітчизняному науковому просторі окремі аспекти мікросервісного підходу, хмарних архітектур і serverless-моделі розглядалися у роботах Т. Науменко, А. Петренка, Т. Кондратюка, В. Дюльгера, А. Сорокіна, Є. Боровскової, А. Баштового, А. Фечана, В. Лесного, С. Антоненка, О. Улічева, А. Загорулька, В. Павленка, О. Коломійцева, Б. Булах, К. Харченка та інших авторів.

Водночас наявні дослідження, як правило, зосереджуються на окремих аспектах функціонування таких систем – зокрема продуктивності, економічній ефективності, питаннях зберігання та обробки даних у serverless-моделі, а також особливостях застосування конкретних технологічних платформ або прикладних сценаріїв (наприклад, побудови окремих serverless-рішень). Це формує важливу базу результатів, однак не завжди дає узгоджене архітектурне бачення того, як саме мікросервісна модель має інтерпретуватися у serverless-середовищі з урахуванням зміни життєвого циклу компонентів, механізмів розгортання та інтеграційної інфраструктури.

Разом із тим, питання архітектурного поєднання мікросервісної моделі з serverless-середовищами та впливу такого поєднання на характер архітектурних рішень залишається відкритим і потребує подальшого наукового осмислення. Зокрема, недостатньо висвітленими залишаються підходи до інтерпретації мікросервісної моделі в умовах безсерверного середовища виконання, у межах якого змінюються традиційні уявлення про межі компонентів, життєвий цикл та механізми розгортання.

Таким чином, актуальною є необхідність дослідження serverless-середовищ як одного з можливих середовищ реалізації мікросервісної архітектури з точки зору їх впливу на архітектурні рішення та властивості мікросервісних систем.

Актуальність обраної теми зумовлена поєднанням зростаючої складності сучасних програмних систем, практичного інтересу до serverless-технологій та потреби в архітектурному осмисленні їх застосування у контексті мікросервісної моделі.

**Об’єктом дослідження** є мікросервісна архітектура розподілених програмних систем

**Предмет дослідження** – вплив serverless-середовищ на архітектуру мікросервісних систем.

**Мета** – аналіз архітектурних підходів до реалізації мікросервісів у serverless-середовищах.

Поставлена мета передбачає вирішення наступних завдань:

- проаналізувати еволюцію архітектурних підходів до побудови програмних систем та передумови формування мікросервісної моделі
- дослідити особливості serverless-середовищ як моделей виконання розподілених програмних систем
- проаналізувати сучасний стан наукових і прикладних досліджень у сфері поєднання мікросервісної архітектури та serverless-технологій
- розглянути архітектурні підходи до реалізації мікросервісів у serverless-середовищах з урахуванням їх впливу на межі компонентів і життєвий цикл сервісів
- спроектувати та реалізувати демонстраційну систему для ілюстрації практичного застосування мікросервісної архітектури у serverless-середовищі
- проаналізувати прийняті архітектурні рішення на прикладі демонстраційної системи з точки зору відповідності принципам мікросервісної архітектури.

**Методи дослідження.** Наукові положення магістерської роботи базуються на сукупності загальнонаукових та спеціальних методів і підходів, необхідних для досягнення поставленої мети та розв'язання визначених завдань дослідження, зокрема:

- аналіз наукових і прикладних джерел, присвячених архітектурі розподілених програмних систем, мікросервісному підходу та serverless-середовищам, а також аналіз сучасних архітектурних практик їх використання;
- синтез теоретичних положень і практичних підходів до реалізації мікросервісної архітектури у serverless-середовищах для формування цілісного уявлення про предмет дослідження;

- порівняння різних архітектурних підходів до побудови розподілених систем з метою виявлення їх особливостей, переваг та обмежень у контексті використання serverless-технологій;
- узагальнення результатів аналізу наукових джерел і практичної реалізації демонстраційної системи для формулювання висновків щодо архітектурних особливостей поєднання мікросервісної моделі та serverless-середовищ;
- моделювання, що полягає у проєктуванні архітектури демонстраційної системи як спрощеного відображення реальної мікросервісної системи, реалізованої у serverless-середовищі;
- проєктування та практична реалізація програмних компонентів демонстраційної системи з використанням сучасних хмарних сервісів і підходів до автоматизації інфраструктури;
- архітектурний аналіз реалізованої демонстраційної системи з точки зору відповідності базовим принципам мікросервісної архітектури;
- бібліографічний пошук, реферування та цитування наукових, технічних і нормативних джерел інформації, що стосуються теми дослідження.

Наукова новизна магістерської роботи полягає в архітектурному осмисленні поєднання мікросервісної моделі та serverless-середовищ з урахуванням особливостей безсерверної моделі виконання. У роботі запропоновано узагальнений підхід до розгляду serverless-середовищ як одного з можливих середовищ реалізації мікросервісної архітектури, що дозволяє проаналізувати вплив таких середовищ на організацію мікросервісів, їх межі та життєвий цикл.

У межах дослідження сформовано архітектурний погляд на реалізацію мікросервісів у serverless-середовищах на основі практичного прикладу, що дало змогу конкретизувати особливості застосування мікросервісних принципів в умовах керованої хмарної інфраструктури без надмірного узагальнення або абсолютизації окремих технічних рішень.

Практичним результатом дослідження є наочна демонстрація можливості реалізації базових принципів мікросервісної архітектури в умовах serverless-середовища. Розроблена в межах роботи демонстраційна система ілюструє підходи до організації мікросервісів та їх взаємодії з урахуванням особливостей безсерверної моделі виконання.

Отримані результати можуть бути корисними при розгляді архітектурних рішень, пов'язаних із використанням serverless-технологій у контексті мікросервісної архітектури, зокрема в частині інтерпретації базових принципів мікросервісної моделі та їх адаптації до специфіки безсерверного середовища виконання.

Окремі результати магістерського дослідження обговорювалися та були представлені під час міжнародної науково-практичної конференції з публікацією матеріалів у збірнику конференції [22].

## РОЗДІЛ 1

### ЕВОЛЮЦІЯ АРХІТЕКТУР ПРОГРАМНИХ СИСТЕМ

Розвиток архітектур програмного забезпечення є природною реакцією на постійне вдосконалення технологій та зростання масштабів їхнього застосування. Зі збільшенням обчислювальних потужностей, появою нових способів зберігання й обміну даними, а також із підвищенням навантаження на системи, постала потреба у більш гнучких і масштабованих підходах до організації програмних рішень.

Питання еволюції архітектур програмного забезпечення викликає сталий інтерес як серед практиків, так і серед науковців. У сучасних дослідженнях і працях провідних архітекторів, зокрема Марка Річардса та Ніла Форда [1], здійснюється систематизація архітектурних підходів і класифікація моделей проектування, що відображає прагнення галузі до впорядкування накопиченого досвіду.

Формування архітектурних підходів відбувалося поступово, через взаємний вплив і трансформацію існуючих рішень. Водночас простежується спільна тенденція – перехід від централізованих структур до децентралізованих і розподілених моделей, спрямованих на підвищення гнучкості, масштабованості та ефективності програмних систем.

У подальших підрозділах розглянуто основні етапи цієї еволюції – ключові архітектурні підходи, що передували формуванню мікросервісної моделі та заклали підґрунтя для її появи

## 1.1. Монолітна архітектура

На початкових етапах розвитку програмного забезпечення переважали монолітні архітектури, у яких уся функціональність застосунку – від користувацького інтерфейсу до бізнес-логіки та доступу до даних – була зосереджена в єдиному програмному компоненті. Такі системи зазвичай створювалися як цілісні застосунки з тісно пов'язаними компонентами, спільним середовищем виконання та єдиним життєвим циклом розробки.

Попри свою простоту, монолітна архітектура не позбавлена переваг і не втратила актуальності навіть сьогодні. Її цілісність спрощує керування кодовою базою, тестування та відлагодження, а також зменшує інфраструктурні витрати. У випадках, коли система не потребує горизонтального масштабування або має стабільні вимоги до навантаження, моноліт залишається цілком виправданим і ефективним вибором. Такий підхід особливо доречний на ранніх етапах розробки, коли важливими є швидкість створення прототипів і передбачуваність поведінки системи.

Водночас із розширенням функціональності та збільшенням масштабів проєкту монолітна структура поступово виявляє свої обмеження. Високий рівень зв'язності між компонентами ускладнює локалізацію змін і підвищує ризик побічних ефектів під час оновлень. Будь-яка модифікація потребує повторного збирання та розгортання всієї системи, що знижує швидкість впровадження нових функцій. Масштабування в межах моноліту можливе лише шляхом повного дублювання всієї системи, навіть якщо підвищене навантаження спостерігається лише на окремому функціональному сегменті.

Ці обмеження стали поштовхом до подальшої еволюції – пошуку способів структурного впорядкування внутрішньої логіки системи та зменшення її зв'язності без втрати цілісності. Наступним етапом цього розвитку стали модульні та компонентні підходи, які заклали основу для поділу програмних систем на логічно ізольовані частини з чіткими межами відповідальності.

## 1.2. Модульні та компонентні підходи

Наступним етапом еволюції архітектур програмного забезпечення стають модульні та компонентні підходи, орієнтовані на впорядкування внутрішньої структури системи без втрати її цілісності. Вони передбачають поділ застосунку на частини з чітко визначеними межами відповідальності та контрольованими взаємозалежностями.

Концептуальні основи такого поділу закладені у роботах Д. Парнаса [14], який формулює принцип приховування реалізації (information hiding). Він визначає, що зміни в одному модулі не повинні впливати на інші, якщо між ними дотримано чітко визначені інтерфейси. Цей підхід становить фундамент подальшої модульності у програмній інженерії.

Структурований поділ системи може здійснюватися за різними ознаками. Багатошарова архітектура (Layered Architecture) передбачає впорядкування компонентів за технічними шарами – інтерфейс користувача, бізнес-логіка та доступ до даних (рис 1.1). Модульна архітектура (Modular Architecture), натомість, ґрунтується на поділі за предметними областями, де кожен модуль відповідає окремій бізнес-функції або домену (рис 1.2).

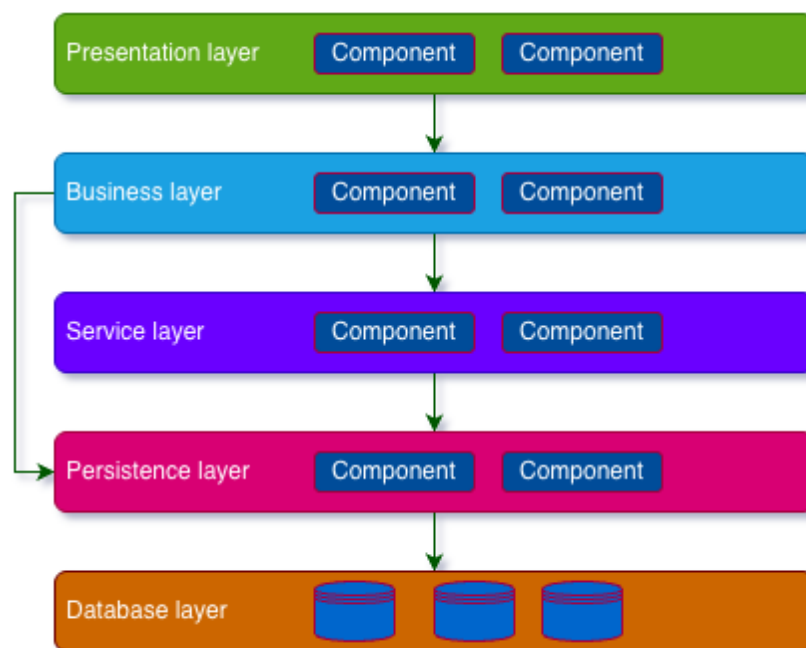


Рис. 1.1. Багатошарова архітектура



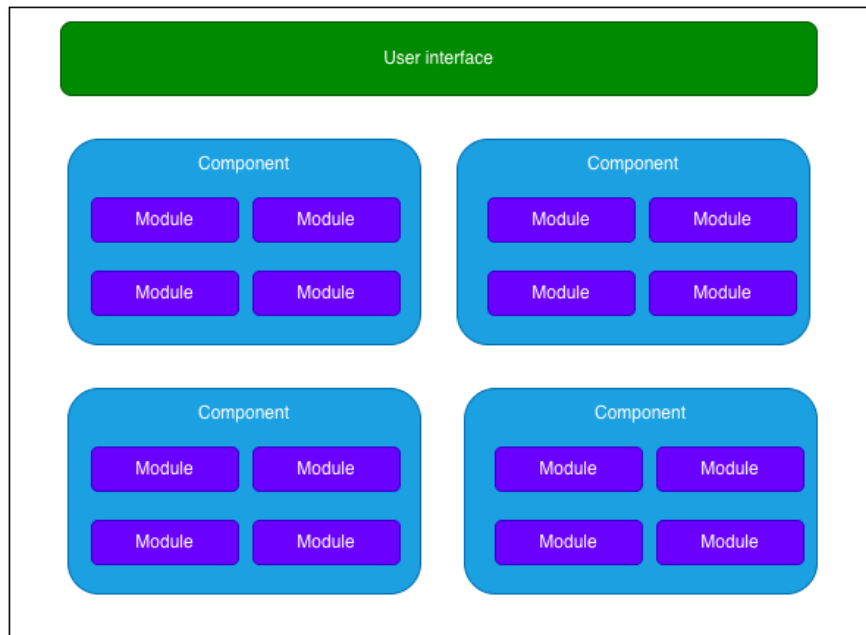


Рис. 1.2. Модульна архітектура

Застосування таких принципів структуризації поклало основу для розвитку розподілених архітектурних підходів. Вони сприяють формуванню концепцій, у яких ізольованість компонентів і чіткість меж відповідальності стають ключовими чинниками масштабованості та гнучкості програмних систем.

### **1.3. Перехід до розподілених систем: клієнт–серверна та багаторівнева моделі**

Наступним етапом розвитку архітектур програмного забезпечення стає перехід від централізованих застосунків до розподілених систем, у яких окремі компоненти можуть виконуватися на різних вузлах і взаємодіяти через мережеві протоколи. Такий підхід відкриває можливість відокремлення інтерфейсу користувача від серверної логіки та даних, закладаючи основу для подальших архітектурних моделей.

Одним із найпоширеніших підходів постає клієнт–серверна модель (рис 1.3), що передбачає розділення системи на клієнтську та серверну частини. Клієнт відповідає за взаємодію з користувачем і частину обробки (від локальної валідації та кешування до окремих обчислень) і ініціює звернення

до серверних ресурсів. Сервер зосереджує бізнес-логіку спільного користування та доступ до даних, забезпечуючи узгодженість, безпеку та централізовані сервіси. Такий розподіл дозволяє гнучко балансувати навантаження між сторонами та спрощує еволюцію системи [27].

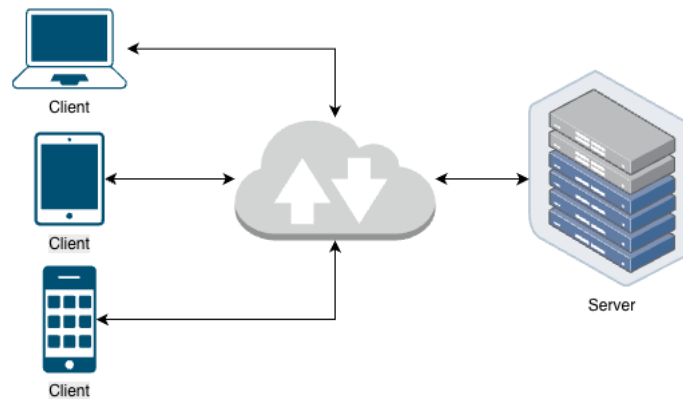


Рис. 1.3. Клієнт–серверна модель

Багаторівнева модель (N-Tier) розвиває цю ідею, розділяючи логічні частини системи на кілька рівнів – інтерфейс користувача, бізнес-логіку, доступ до даних тощо – і за потреби розгортаючи їх на різних середовищах. Це підвищує масштабованість і керованість, уможлиблює незалежне оновлення окремих рівнів і точкове балансування навантаження.

Водночас мережева взаємодія привносить додаткові виклики: затримки й нестабільність з'єднань, потребу в стандартизованих протоколах і механізмах узгодженості даних. Попри це, клієнт–серверна та багаторівнева моделі створюють технічне підґрунтя для подальшого розвитку розподілених систем, у яких взаємодія між компонентами набуває більш формалізованого, стандартизованого та незалежного характеру.

#### 1.4. Сервісно-орієнтована архітектура (SOA)

Подальший розвиток розподілених систем призводить до появи підходів, що забезпечують формалізовану взаємодію між автономними компонентами. Одним із найвпливовіших рішень цього етапу стає сервісно-орієнтована

архітектура (Service-Oriented Architecture, SOA) (рис 1.4). Її мета полягає у створенні системи з незалежних сервісів, кожен з яких виконує окрему бізнес-функцію та взаємодіє з іншими через чітко визначені інтерфейси.

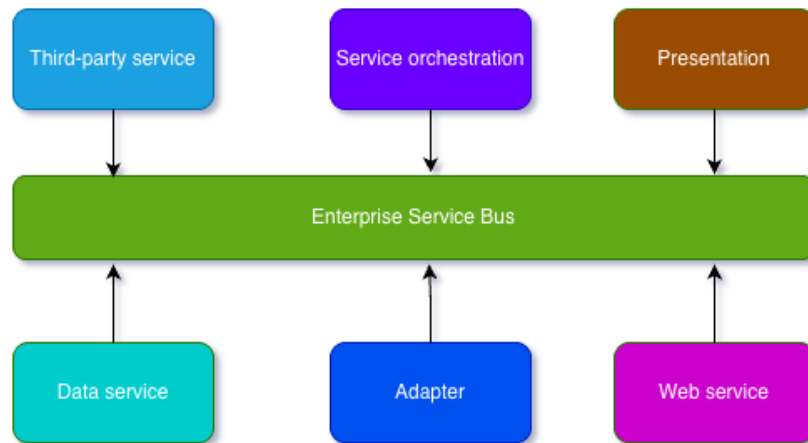


Рис. 1.4. Сервіс-орієнтована архітектура

Концепція SOA активно формується у 2000-х роках, коли великі корпоративні системи потребують єдиних принципів інтеграції та повторного використання бізнес-функцій. Важливий внесок у формування теоретичних засад цього підходу зробив Томас Ерл, чия праця *Service-Oriented Architecture: Concepts, Technology, and Design* (2005) стала фундаментальною у визначенні принципів побудови сервісів, повторного використання, слабкої зв'язності та інкапсуляції бізнес-логіки [**Error! Reference source not found.**].

В основі SOA лежать ключові принципи: незалежність сервісів, чітко визначені контракти взаємодії, повторне використання та посередництво через спільну інфраструктуру. Для реалізації комунікації між сервісами часто застосовуються стандартизовані протоколи, зокрема SOAP або XML-RPC, а також шини даних (Enterprise Service Bus, ESB), що виконують роль централізованого посередника. Такий підхід дає змогу уніфікувати обмін даними між різнорідними компонентами, незалежно від платформи чи технологічного стеку.

Водночас централізація комунікації через шину створює нові обмеження – поява “єдиної точки відмови”, складність масштабування та підвищена залежність від інфраструктури. Ці фактори стимулюють подальшу еволюцію архітектур у напрямку децентралізації, де кожен сервіс стає повністю самостійним елементом. Саме ця ідея знаходить своє втілення у мікросервісній архітектурі, що розглядається на наступному етапі.

### 1.5. Перехід до мікросервісної архітектури

Подальша еволюція архітектур програмного забезпечення спрямована на подолання обмежень, пов’язаних із надмірною централізацією та складністю координації численних сервісів у розподілених системах. Це призводить до формування мікросервісної архітектури (Microservices Architecture) – стилю проєктування, що передбачає побудову системи як набору невеликих, автономних сервісів, кожен з яких виконує окрему бізнес-функцію, має власний життєвий цикл і може розвиватись, масштабуватися та впроваджуватись незалежно від інших (рис 1.5). Взаємодія між сервісами здійснюється через стандартизовані, легковагі інтерфейси – зазвичай HTTP API або асинхронні механізми обміну повідомленнями. Концептуальне визначення цього підходу сформульоване Мартіном Фаулером та Джеймсом Льюїсом у праці *Microservices* (2014) [2].

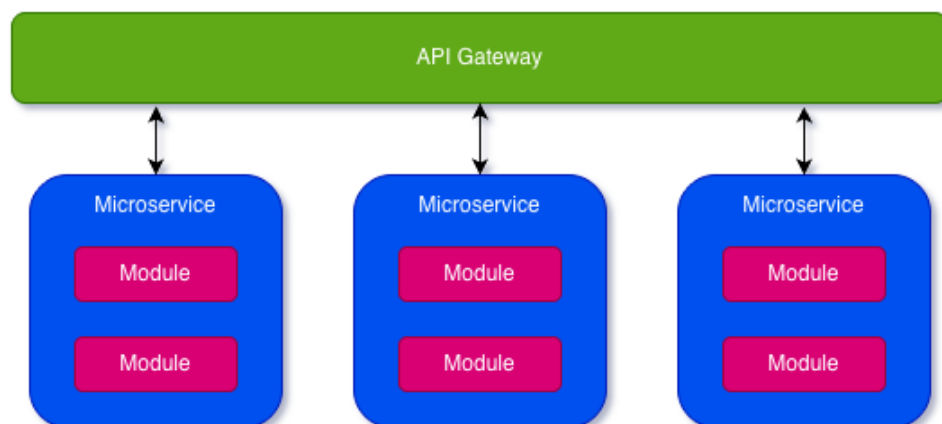


Рис. 1.5. Мікросервісна архітектура

Мікросервіси продовжують ідеї сервісно-орієнтованої архітектури, але роблять акцент на децентралізації управління, мінімізації спільної інфраструктури та ізоляції середовищ виконання. Кожен сервіс є самостійним компонентом, який може мати власну базу даних, технологічний стек і команду розробки. Такий підхід забезпечує високу гнучкість, прискорює розгортання змін і підвищує масштабованість системи [32].

Практичне поширення мікросервісної архітектури починається у великих технологічних компаніях. Amazon ще у 1998 році формулює принципи розподіленої архітектури у документі Distributed Computing Manifesto [5], що пізніше лягли в основу сервісної моделі AWS. Netflix у 2012 році завершує перехід від моноліту до мікросервісів для підтримки масштабування своєї платформи потокового відео [6]. Ці приклади стають знаковими для індустрії, продемонструвавши переваги децентралізованого підходу у високонавантажених середовищах.

У стислому формулюванні Фаулер і Льюїс описують мікросервіси як стиль побудови програмної системи, у якій кожен сервіс:

- виконується у власному процесі;
- взаємодіє з іншими через легкі механізми комунікації;
- сфокусований навколо чітко окресленої області відповідальності;
- може бути змінений і розгорнутий незалежно;
- керується з мінімальним ступенем централізації;
- потенційно реалізований на різних мовах програмування чи з різними базами даних.

Ці характеристики не зводяться лише до технічних аспектів, а відображають фундаментальні архітектурні принципи, що відрізняють мікросервісний підхід від традиційних монолітних систем. Межі між сервісами зазвичай визначаються за доменними ознаками, відповідно до структури предметної області. Такий підхід узгоджується з концепцією bounded context у доменно-

орієнтованому проектуванні (DDD) [**Error! Reference source not found.**], яка передбачає розподіл системи на ізольовані області відповідальності.

Ізольованість процесів виконання дозволяє кожному сервісу мати власне середовище запуску – процес, контейнер або функцію [25], що забезпечує:

- незалежне масштабування;
- підвищену відмовостійкість;
- технологічну автономність.

Водночас такий рівень розподілення створює нові виклики, зокрема пов'язані з міжпроцесною взаємодією, нестабільністю мережі, варіативністю затримок, втратами повідомлень та потребою в реалізації стратегій повторних викликів з урахуванням ідемпотентності. Для забезпечення надійності використовуються шаблони стійкості, такі як circuit breaker, bulkhead isolation або retry with backoff, що дозволяють запобігати каскадним відмовам у системі.

Важливе значення має вибір механізмів комунікації між сервісами. Залежно від характеру бізнес-процесів застосовуються синхронні HTTP-виклики або асинхронні рішення на основі черг повідомлень і подій. Обраний стиль комунікації впливає на узгодженість даних, продуктивність і надійність системи.

Однією з ключових переваг мікросервісної архітектури є можливість незалежного впровадження та еволюції окремих компонентів [26]. Це можливо лише за умови дотримання стабільних контрактів, сумісності з поточним навантаженням і коректного версіонування API. Такі властивості суттєво підвищують гнучкість розробки й спрощують супровід у масштабі великих організацій.

Поліглотність у мікросервісній архітектурі є наслідком незалежності реалізації окремих сервісів та наявності формалізованих контрактів взаємодії. Вона передбачає можливість використовувати різні мови програмування чи технологічні платформи для реалізації окремих компонентів, що дозволяє командам обирати інструменти, найкраще пристосовані до конкретних

завдань, зберігаючи при цьому узгодженість системи завдяки стандартизованим інтерфейсам.

Отже, мікросервісна архітектура відображає подальший етап еволюції програмних систем у напрямку підвищення гнучкості, масштабованості та технологічної автономності.

*Таблиця 1.1*

Порівняльна таблиця основних архітектур програмних систем

Ознака / Архітектура	Монолітна	Сервісно-орієнтована (SOA)	Мікросервісна
<b>Основна ідея</b>	Вся система – єдиний застосунок	Незалежні сервіси, інтегровані через спільну шину	Автономні сервіси з мінімальною централізацією
<b>Тип взаємодії компонентів</b>	Внутрішні виклики функцій	Через ESB	HTTP API, події, асинхронні черги
<b>Ступінь зв'язності</b>	Висока	Помірна (через спільну інфраструктуру)	Низька, слабка зв'язність
<b>Масштабованість</b>	Лише вертикальна	Висока, але потребує централізованого керування	Висока, гнучке незалежне масштабування
<b>Відмовостійкість</b>	Низька – збій зупиняє систему	Залежить від шини (ESB)	Висока, завдяки ізоляції сервісів
<b>Тип даних / сховище</b>	Спільна база даних	Може бути спільна або інтегрована через ESB	Незалежні бази даних на сервіс
<b>Переваги</b>	Простота, цілісність, легке тестування	Повторне використання, інтеграція	Гнучкість, незалежне розгортання, масштабованість
<b>Недоліки</b>	Важко масштабувати, змінювати	Централізована шина, висока складність	Координація, DevOps-складність, спостережуваність

## **Висновки до розділу 1**

Еволюція архітектур програмного забезпечення від монолітних до мікросервісних моделей демонструє послідовний рух галузі від централізованих, тісно пов'язаних структур до більш гнучких, модульних і розподілених систем. Кожен етап цієї трансформації – від впровадження принципів модульності та розмежування відповідальності до формалізації взаємодії сервісів у межах SOA – закладав основу для підвищення масштабованості, надійності та незалежності компонентів.

Мікросервісна архітектура стала логічним результатом цього розвитку, поєднавши автономність, чітке розмежування доменів і стандартизовану комунікацію. Водночас її децентралізована природа привнесла нові виклики – насамперед щодо керування інфраструктурою та складності координації великої кількості сервісів. Подальший розвиток архітектурних підходів зосереджується на пошуку рішень, здатних зберегти переваги мікросервісності, мінімізуючи супровідні витрати, що стало передумовою появи serverless-моделі, розглянутої у наступному розділі.



## РОЗДІЛ 2

### SERVERLESS ЯК СЕРЕДОВИЩЕ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ ВЗАЄМОДІЇ

Упродовж останнього десятиліття зростання масштабів розподілених систем та ускладнення управління інфраструктурою стимулювали появу нових моделей виконання програмних компонентів. Однією з таких моделей стала так звана serverless-модель, яка, попри назву, не означає відсутність серверів. Як влучно зазначає Кен Фромм, автор одного з ранніх описів концепції: “Термін serverless не означає, що сервери більше не використовуються. Він лише означає, що розробникам більше не доводиться про них особливо замислюватися” [9].

Термін serverless набув широкої популярності після анонсу AWS Lambda на конференції AWS re:Invent у 2014 році [8]. Відтоді ним позначають підхід, за якого хмарний провайдер повністю бере на себе розгортання, масштабування, моніторинг та обслуговування інфраструктури. Розробник, натомість, зосереджується на реалізації прикладної логіки та визначенні сценаріїв її активації. Цей підхід охоплює не лише виконання прикладної логіки, а й інші компоненти системи – бази даних, брокери повідомлень, сервіси аутентифікації та маршрутизації – які також можуть бути реалізовані у безсерверному форматі.

Виконання прикладного коду в serverless-середовищах зазвичай реалізується у форматі Function-as-a-Service (FaaS) – моделі, у якій код запускається у відповідь на події, такі як HTTP-запити, повідомлення з брокерів, зміни у базах даних або таймери. Такі функції мають короткий життєвий цикл, автоматично масштабуються та оплачуються за фактичний час виконання [24]. Подібні сервіси реалізовані у провідних хмарних платформах – AWS Lambda (Amazon Web Services), Azure Functions (Microsoft), Google Cloud Functions і IBM Cloud Functions [10, 11, 12, 30].

Попри те, що serverless не є архітектурним стилем у класичному розумінні, його дедалі частіше розглядають як ефективну модель виконання для мікросервісних рішень. Її характерні властивості – ізольованість, незалежність розгортання, масштабованість – добре узгоджуються з принципами мікросервісної архітектури. Водночас це поєднання породжує і нові практичні питання, які потребують окремого аналізу в контексті конкретних реалізацій.

## **2.1. Актуальний стан досліджень**

У сучасній науковій літературі спостерігається стійкий інтерес до теми поєднання мікросервісної архітектури та serverless-моделі виконання. Роботи останніх років охоплюють як концептуальні огляди, так і експериментальні дослідження, що аналізують продуктивність і вартість таких систем.

Прикладом таких досліджень є робота Kundavaram [17], у якій систематизовано базові концепції serverless-моделі. Автор визначає чотири ключові складові, що формують її архітектурну основу: інфраструктурна абстракція, динамічне управління ресурсами, модель оплати за фактичне використання та фокус на розробці прикладної логіки.

Інфраструктурна абстракція знімає з розробника необхідність безпосереднього адміністрування серверів, покладаючи ці завдання на автоматизований рівень платформи, який відповідає за оркестрацію контейнерів, мережеві налаштування, оновлення та безпеку. Динамічне управління ресурсами дає змогу автоматично масштабувати систему в реальному часі залежно від поточного навантаження. Платформа відстежує показники виконання та адаптивно регулює обсяг ресурсів, підтримуючи стабільну продуктивність при мінімальному споживанні. Модель оплати за фактичне використання реалізує принцип відповідності витрат реальному часу виконання та обсягу спожитих ресурсів. Вартість обчислень формується пропорційно до тривалості роботи функцій, використаної пам'яті, кількості запитів і обсягу переданих даних, що дозволяє уникнути витрат на

невикористані ресурси. Фокус на розробці прикладної логіки зміщує увагу розробника з керування інфраструктурою на оптимізацію застосунку та реалізацію бізнес-функцій. Платформа надає вбудовані засоби розгортання, тестування та моніторингу, що спрощує життєвий цикл розробки та підвищує продуктивність команд.

Окремий напрям досліджень присвячений аналізу serverless-архітектур у контексті еволюції традиційних моделей зберігання та обробки даних. У цих роботах розглядаються фактори, що впливають на вибір безсерверних рішень, серед яких автоматичне масштабування, гнучке керування ресурсами, подієва модель взаємодії та економічна модель оплати за використання. Разом із перевагами відзначаються й виклики, пов'язані з узгодженістю даних, обмеженою гнучкістю конфігурації та адаптацією класичних підходів до роботи з транзакціями в умовах serverless-середовищ [22, 23, 28].

Також уваги заслуговують експериментальні дослідження, присвячені вивченню практичних аспектів використання serverless-моделі у контексті мікросервісних архітектур. У низці робіт [18,19] проведено емпіричне порівняння різних варіантів реалізації мікросервісних систем – від традиційних контейнеризованих розгортань до функційної моделі AWS Lambda. Автори аналізували такі параметри, як масштабованість, час відгуку, вартість та особливості перенесення застосунків між різними середовищами виконання. Результати демонструють, що serverless-рішення забезпечують гнучке автоматичне масштабування та зниження операційних витрат, але потребують урахування відмінностей у структурі застосунків і можуть поступатися за стабільністю часу відгуку.

Проведений аналіз літературних джерел підтверджує актуальність теми та наявність стійкого інтересу до поєднання мікросервісної архітектури з serverless-підходами. Водночас більшість досліджень зосереджуються переважно на технічних або економічних аспектах, залишаючи поза увагою архітектурний вимір – питання збереження меж сервісів, ізольованості та керованості при перенесенні мікросервісної моделі до безсерверного

середовища. Цей аспект визначає напрям подальшого аналізу, представлений у наступних підрозділах.

## 2.2. Serverless як середовище реалізації мікросервісів

Значний внесок у формування архітектурних концепцій застосування serverless-підходів роблять провідні практики галузі. Зокрема, Сем Ньюмен у другому виданні Building Microservices [3] описує можливі варіанти адаптації мікросервісної моделі до середовищ виконання на основі Function-as-a-Service (FaaS).

Автор виділяє два базові підходи до організації мікросервісів у таких середовищах:

- **Function per microservice** (одна функція на мікросервіс).

Цей варіант передбачає, що мікросервіс розгортається як єдина функція у FaaS-середовищі. Усі запити або події потрапляють до спільної точки входу функції, яка виконує маршрутизацію до відповідних частин логіки сервісу. Такий підхід зберігає усталене уявлення про мікросервіс як єдину одиницю розгортання й може бути раціональним вибором на початкових етапах.

- **Function per aggregate** (одна функція на агрегат).

У цьому підході мікросервіс поділяється на кілька функцій, кожна з яких реалізує бізнес-логіку одного агрегату – цілісної групи сутностей предметної області відповідно до принципів Domain-Driven Design [**Error! Reference source not found.**]. Така структура забезпечує ізоляцію життєвого циклу кожного агрегату та зменшує ризик взаємного впливу логіки. Мікросервіс при цьому розглядається як логічне об'єднання кількох функцій.

Розглянуті підходи описують базові моделі перенесення мікросервісної логіки до функційної парадигми виконання. У подальшому аналізі ці моделі розглядаються детальніше з практичної точки зору – з урахуванням способів збереження архітектурної цілісності сервісу, організації коду та керування функціями.

## **2.3. Реалізація принципів мікросервісної архітектури у serverless-моделі**

Розглянуті вище підходи окреслюють загальну логіку перенесення мікросервісної архітектури у середовище з функційною моделлю виконання. На практиці ж постає низка додаткових аспектів, що визначають ефективність і цілісність такого перенесення.

У цьому підрозділі увагу зосереджено на деталізації архітектурних рішень, пов'язаних із організацією функцій у межах мікросервісу, підтримкою узгодженості коду, засобами інфраструктурного опису та взаємодією між компонентами serverless-середовища.

### **2.3.1. Організація коду та інфраструктурне управління**

На відміну від класичної моделі, де мікросервіс є єдиною одиницею виконання, у serverless-середовищі мікросервіс може бути представлений набором незалежних функцій. Це розширює можливості масштабування й оптимізації ресурсів, але водночас ускладнює підтримання архітектурної цілісності. Для збереження узгодженості необхідно забезпечити єдність доменної моделі, конфігурації середовища та політик доступу між усіма функціями сервісу.

Доцільним способом збереження цілісності є застосування підходу, за якого кожен мікросервіс підтримується в межах окремого репозиторію. Така організація дозволяє об'єднати всі функції, спільні моделі даних, бібліотеки та конфігураційні файли у спільну кодову базу. У ході підготовки до впровадження з неї можуть створюватися кілька незалежних функцій, кожна з яких відповідає за окремий бізнес-процес або агрегат, але всі вони підтримуються спільними механізмами тестування, збірки та впровадження.

Для узгодженого керування інфраструктурою застосовуються підходи Infrastructure as Code (IaC), які дозволяють описувати ресурси системи як частину кодової бази. Такі описи охоплюють визначення функцій, API-шлюзів, черг повідомлень, таблиць баз даних і політик доступу, що забезпечує

цілісне відтворення середовища та контроль його версій. У результаті мікросервіс зберігає архітектурну єдність, навіть якщо фізично складається з багатьох окремих функцій.

### **2.3.2. Організація та гранулярність функцій у межах мікросервісу**

Раніше було розглянуто моделі, запропоновані Семом Ньюменом, які описують базові варіанти відображення мікросервісів у функційній парадигмі виконання – Function Per Microservice та Function Per Aggregate. Ці моделі задають загальний напрям для побудови архітектури, проте на практиці постає ширший спектр принципів групування функцій, що визначають гнучкість, масштабованість і керованість системи.

Одним із ключових питань є вибір рівня гранулярності – тобто визначення, який обсяг функціональності доцільно реалізовувати в межах однієї функції. У мікросервісній архітектурі хорошою практикою вважається узгодження меж сервісів із bounded context відповідно до підходу Domain-Driven Design, проте фактичний рівень поділу залежить від глибини моделювання предметної області та специфіки бізнес-процесів. Аналогічно, у serverless-середовищі оптимальний рівень поділу функцій визначається доменом, характером навантаження та типами взаємодії між компонентами. Функції з великою кількістю залежностей або значним обсягом коду стають ресурсоемними, що збільшує час запуску (cold start) [20]. Менші за розміром функції завантажуються швидше й забезпечують вищу реактивність системи.

Окрім продуктивності, гранулярність також впливає на ефективність використання ресурсів і можливість тонкого налаштування середовища виконання. Водночас надмірне дроблення функцій призводить до збільшення кількості артефактів, складнішого процесу збірки та впровадження, а також ускладнює тестування й моніторинг. Оптимальний баланс між розміром і керованістю функцій досягається шляхом архітектурного аналізу з урахуванням продуктивності, вартості виконання та складності супроводу.

Окремою перевагою більш деталізованого поділу функцій є можливість гнучкого керування ресурсами. Serverless-платформи, зокрема AWS Lambda, дозволяють налаштовувати обсяг пам'яті, тайм-аути та рівень паралельності для кожної функції окремо [21]. Це відкриває можливість точного розподілу ресурсів: виділення додаткових потужностей для критичних операцій або обмеження паралельності для менш пріоритетних. Така ізолюваність допомагає уникнути ситуацій, коли одна перевантажена функція блокує обробку важливих запитів у системі.

Структурування функцій також може здійснюватися за рівнем доступу або типом користувачів. У межах одного мікросервісу доцільно відокремлювати функції, призначені для клієнтів, від адміністративних функцій, оскільки вони можуть відрізнятися логікою авторизації, моделями даних і навіть джерелами інформації. Такий поділ спрощує контроль доступу й підвищує безпеку, особливо в системах із різними сценаріями взаємодії.

Ще одним критерієм групування є тип взаємодії. Частина функцій може обробляти синхронні запити через REST або GraphQL API, тоді як інші – реагувати на події з черг повідомлень, потоків баз даних чи таймерів. Розділення функцій за типом інтеграції полегшує налаштування тригерів, логування та масштабування окремих каналів комунікації. Крім того, це сприяє більшій стабільності системи: помилки або перевантаження в асинхронній гілці не впливають безпосередньо на користувацькі запити.

Незалежно від обраної стратегії групування функцій, архітектура сервісу має забезпечувати ізолюваність доменної логіки від технічних аспектів реалізації. Чіткий поділ на рівні даних, бізнес-логіки та інтерфейсів дозволяє змінювати структуру або кількість функцій без втручання в основні бізнес-процеси. Такий підхід узгоджується з принципами гексагональної архітектури (Hexagonal Architecture), у межах якої зовнішні адаптери можуть еволюціонувати незалежно від ядра. У контексті serverless це забезпечує стабільність доменної моделі та спрощує повторне використання компонентів.

### 2.3.3. Інтеграційна інфраструктура serverless-середовища

Традиційна мікросервісна архітектура спирається на ефективні мережеві механізми взаємодії між сервісами. Найпоширенішими з них є синхронні – засновані на викликах через HTTP/REST, RPC або gRPC – та асинхронні, що використовують брокери повідомлень на кшталт Kafka, RabbitMQ чи ActiveMQ. Незалежно від конкретних технологій, метою цих механізмів є забезпечення зв'язності між сервісами при збереженні їхньої автономності.

Разом із тим, у традиційних середовищах побудова такої інтеграційної інфраструктури потребує значних зусиль [29]. Для забезпечення надійної синхронної взаємодії необхідно налаштовувати маршрути мережевого трафіку, балансувальники навантаження, системи виявлення сервісів та механізми автоматичного масштабування. Усі ці елементи мають бути узгоджені між собою, підтримувати високу доступність і відповідати вимогам безпеки.

Асинхронна взаємодія також не є простою у реалізації: платформи на зразок Kafka або RabbitMQ потребують ретельної конфігурації, що охоплює керування кластером, реплікацію даних, контроль доступу, підтримку високої доступності та розподіл навантаження між розділами. До цього додаються питання моніторингу, відновлення після збоїв та сумісності клієнтських бібліотек. У результаті значна частина зусиль команди зосереджується не на реалізації бізнес-логіки, а на підтримці комунікаційної інфраструктури.

У serverless-моделі більшість цих завдань вирішується на рівні платформи. Обравши надійного хмарного провайдера, розробник отримує інтеграційну інфраструктуру як готовий набір сервісів, що не потребують складного розгортання чи обслуговування. Наприклад, поєднання API Gateway та AWS Lambda забезпечує повноцінну синхронну взаємодію з автоматичним масштабуванням, керуванням навантаженням і вбудованими механізмами безпеки. Аналогічно, сервіси SQS або SNS надають готовий механізм обміну повідомленнями, здатний обробляти значні обсяги трафіку без втручання розробника в інфраструктурні деталі.



Разом із тим, такий рівень абстракції має і свої обмеження. У serverless-середовищах розробник позбавлений глибокого контролю над конфігурацією мережових компонентів, параметрами балансування чи деталями масштабування. Можливість оптимізації продуктивності або поведінки системи часто визначається рамками, встановленими провайдером. На відміну від власноруч розгорнутої інфраструктури, де всі елементи можна гнучко налаштувати, у serverless доводиться адаптувати архітектуру до доступних можливостей платформи.

Таким чином, serverless-середовище забезпечує зрілу інфраструктуру для інтеграції мікросервісів, проте зменшує контроль розробника над її параметрами. Отримуючи готові механізми масштабування та взаємодії, команди звільняються від інфраструктурних турбот, але водночас змушені враховувати межі гнучкості платформи. Це визначає необхідність уважного архітектурного планування та зваженого вибору способів інтеграції відповідно до характеристик і обмежень конкретного середовища виконання.

## **Висновки до розділу 2**

У цьому розділі було простежено еволюцію serverless-підходу та обґрунтовано його придатність як середовища для реалізації мікросервісної архітектури. Особливу увагу приділено питанню організації та гранулярності функцій у межах мікросервісу, що визначає здатність системи зберігати логічну цілісність і межі доменних контекстів навіть за умов поділу на окремі функції. Саме цей аспект є ключовим для ефективного перенесення мікросервісної моделі у функційну парадигму виконання.

Зроблено висновок, що serverless забезпечує технічне підґрунтя для реалізації мікросервісів – із вбудованими засобами масштабування, інтеграції та управління інфраструктурою, – однак вимагає продуманого балансування між автономністю сервісів і обмеженнями платформи.

У наступному розділі наведено практичну демонстрацію цього підходу на прикладі побудови мікросервісу у середовищі AWS, з використанням Pulumi для опису інфраструктури як коду та забезпечення цілісного процесу розгортання.

## РОЗДІЛ 3

### ПРАКТИЧНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ У SERVERLESS-СЕРЕДОВИЩІ

#### 3.1. Постановка задачі та підхід до реалізації

У попередніх розділах було розглянуто еволюцію архітектур програмних систем, принципи мікросервісної архітектури, а також особливості serverless як середовища виконання. Аналіз показав, що serverless-підхід накладає певну специфіку на спосіб реалізації мікросервісів, проте не скасовує базових архітектурних принципів, а радше трансформує їх практичне втілення.

Метою практичної частини даної роботи є розробка демонстраційної системи, яка дозволяє проаналізувати, яким чином при переході до serverless-середовища може бути збережена мікросервісна модель та її ключові архітектурні ознаки. Основний акцент зроблено не на повноті бізнес-функціоналу, а на дослідженні архітектурних рішень і механізмів взаємодії між сервісами, що дозволяють поєднати синхронну та асинхронну взаємодію між мікросервісами без втрати їх архітектурної самостійності у serverless-реалізації.

Для досягнення зазначених цілей було обрано домен резервування ресурсів, який природно передбачає розподіл відповідальностей між сервісами та наявність синхронних та асинхронних бізнес-процесів. Під резервуванням у межах даної роботи розуміється попереднє закріплення обмеженого у часі або кількості ресурсу, наприклад переговорної кімнати в офісі, столика в ресторані, транспортного засобу для прокату або іншого подібного об'єкта. На основі обраного домену побудовано демонстраційну систему, що складається з двох мікросервісів, відповідальних за управління резервуваннями та доступністю ресурсів відповідно.

Далі розглядається доменна модель демонстраційної системи та обґрунтовується розподіл відповідальностей між мікросервісами.

### 3.2. Доменна модель та межі мікросервісів

Домен резервування ресурсів, обраний для демонстраційної системи, природно передбачає розподіл відповідальностей між окремими частинами системи та необхідність координації між ними. У межах такого домену логічно виокремлюються процеси управління резервуваннями та управління доступністю ресурсів, які мають різну бізнес-семантику та різні вимоги до збереження стану.

Для забезпечення зрозумілої структури системи та зменшення зв'язаності між її компонентами зазначені процеси було розділено на окремі логічні частини. Такий поділ дозволяє ізолювати бізнес-логіку, пов'язану з життєвим циклом резервувань, від логіки управління ресурсами та їх станом.

Відповідно до цих міркувань домен було поділено на два мікросервіси – Booking та Inventory, кожен з яких має власну область відповідальності та модель даних (рис. 3.1). Обрана структура слугує основою для подальшого аналізу взаємодії між сервісами та способів їх реалізації в розподіленому середовищі.

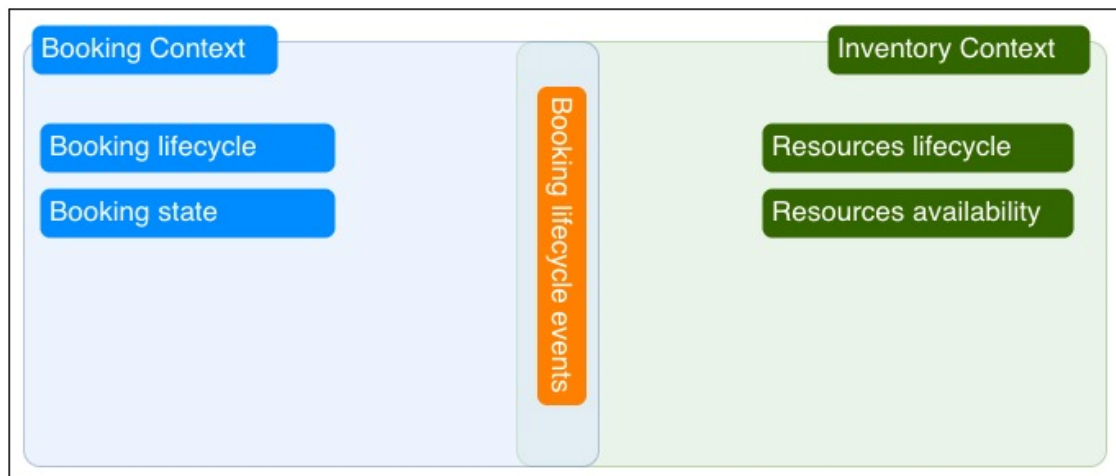


Рис. 3.1. Межі доменних зон

У таблиці 3.1 наведено зовнішні та внутрішні API мікросервісів системи резервування. Перелік інтерфейсів відображає розподіл функцій між

сервісами Booking та Inventory і показує, як користувачі та сервіси взаємодіють між собою.

*Таблиця 3.1*

API мікросервісів системи

Сервіс	Метод	Шлях	Призначення	Споживач
Booking	POST	/bookings	Створення резервування	Клієнт
Booking	GET	/bookings	Отримання списку резервувань	Клієнт
Booking	GET	/bookings/{id}	Отримання стану резервування	Клієнт
Inventory	POST	/items	Створення ресурсу	Менеджер
Inventory	GET	/items	Отримання списку ресурсів	Клієнт/ Менеджер
Inventory	GET	/items/{type}	Отримання списку ресурсів по типу	Клієнт/ Менеджер
Inventory	POST	/items/availability	Перевірка доступності ресурсу	Клієнт/ Менеджер
Inventory	POST	/items:lookup	Отримання списку ресурсів по списку ідентифікаторів	Booking

У межах даної роботи API мікросервісів розглядаються з точки зору їх логічного призначення та типових сценаріїв використання. Хоча на рівні реалізації не вводиться жорстке розмежування доступу до окремих ендпоінтів, кожен API орієнтований на певний клас споживачів, що відповідає його ролі в загальній архітектурі системи. Такий підхід дозволяє зосередитися на архітектурних рішеннях і взаємодії між сервісами, не ускладнюючи модель деталями реалізації.

**Мікросервіс Booking** відповідає за управління процесом резервування ресурсів та супровід його життєвого циклу. Надані ним API зосереджені на роботі з резервуваннями як окремими доменними об'єктами та орієнтовані на взаємодію з клієнтською частиною системи. Через відповідні ендпоінти здійснюється створення нових резервувань, отримання списку резервувань, а також запит поточного стану конкретного резервування.

API мікросервісу Booking не містить операцій, пов'язаних із безпосереднім управлінням ресурсами або перевіркою їх доступності. Таке обмеження є свідомим архітектурним рішенням і зумовлене прагненням уникнути дублювання бізнес-логіки та зберегти чіткий розподіл

відповідальностей між сервісами. Усі операції, пов'язані зі станом ресурсів, винесені за межі даного сервісу.

**Мікросервіс Inventory** відповідає за управління ресурсами системи та зберігання інформації про їхній поточний стан. Його API зосереджені на роботі з переліком ресурсів, а також на наданні даних, необхідних для прийняття рішень у процесі резервування. Через відповідні ендпоінти здійснюється отримання списку доступних ресурсів і виконання допоміжних запитів, що використовуються іншими компонентами системи.

На відміну від мікросервісу Booking, API Inventory не призначені для прямої роботи з процесом резервування. Основною роллю даного сервісу є підтримка актуального стану ресурсів та надання узгодженої інформації про них за запитом. Такий підхід дозволяє зосередити всю логіку, пов'язану з ресурсами, в одному сервісі та уникнути розповсюдження цієї логіки між іншими компонентами системи.

Окремі API мікросервісу Inventory використовуються для взаємодії між сервісами та слугують для отримання детальної інформації про ресурси на основі їх ідентифікаторів. Це дає змогу іншим сервісам, зокрема Booking, працювати з ресурсами опосередковано, не маючи прямого доступу до внутрішніх структур даних Inventory. У результаті зберігається чіткий розподіл відповідальностей і зменшується зв'язність між мікросервісами, що є важливим для подальшого розвитку та масштабування системи.

Події, наведені в таблиці 3.2, відображають ключові етапи процесу резервування та використовуються для асинхронної координації між мікросервісами Booking та Inventory. Кожна подія фіксує завершений бізнес-факт і не передбачає негайної реакції з боку споживача, що дозволяє сервісам залишатися слабо зв'язаними між собою.

Таблиця 3.2

## Асинхронна взаємодія між мікросервісами системи резервування

Подія	Джерело	Опис події	Споживач
BookingRequested	Booking	Створено нове резервування	Inventory
BookingAccepted	Inventory	Підтверджено можливість резервування ресурсу	Booking
BookingRejected	Inventory	Ресурс недоступний для резервування	Booking

Подія **BookingRequested** публікується мікросервісом Booking після створення нового резервування та сигналізує про намір користувача зарезервувати певний ресурс. Мікросервіс Inventory, отримавши цю подію, виконує перевірку можливості резервування ресурсу з урахуванням його поточного стану.

Події **BookingAccepted** та **BookingRejected** публікуються мікросервісом Inventory як результат обробки запиту на резервування. У відповідь на ці події мікросервіс Booking оновлює стан відповідного резервування, відображаючи результат перевірки доступності ресурсу. Такий обмін подіями дозволяє уникнути синхронних викликів між сервісами та зменшує ризик блокування або каскадних відмов.

Запропонована демонстраційна модель домену резервування та взаємодії між мікросервісами дозволяє на практичному прикладі проаналізувати ключові принципи побудови мікросервісної архітектури, зокрема розподіл відповідальностей, ізоляцію бізнес-логіки та використання асинхронної взаємодії. Навмисно спрощена предметна область не знижує цінності моделі, а, навпаки, дає змогу зосередитися на архітектурних аспектах системи без надмірного ускладнення реалізації.

Описані межі сервісів і механізми їх взаємодії формують зручну основу для подальшого дослідження особливостей реалізації мікросервісів у serverless-середовищі. Така модель дозволяє розглянути, яким чином архітектурні принципи зберігаються при перенесенні сервісів у середовище безсерверних обчислень та які переваги і обмеження при цьому виникають.

### **3.3. Архітектурна організація мікросервісів у serverless-середовищі**

У даному підрозділі розглядається практична організація мікросервісів демонстраційної системи у serverless-середовищі. Основна увага приділяється тому, яким чином логічна модель мікросервісів відображається у структурі інфраструктури, наборі функцій та керованих хмарних ресурсів.

Опис зосереджений на архітектурних рішеннях, пов'язаних із організацією інфраструктури, способом її опису у вигляді коду, поділом на спільні та сервісні шари, а також управлінням конфігурацією та адресацією сервісів. Такий підхід дозволяє проаналізувати, як мікросервіси реалізуються в serverless-середовищі з точки зору їх практичного використання та незалежного розвитку.

#### **3.3.1. Infrastructure as Code як основа організації інфраструктури**

У демонстраційній системі організація інфраструктури мікросервісів побудована на основі підходу Infrastructure as Code, за якого всі хмарні ресурси описуються у вигляді програмного коду та створюються автоматизовано. Такий підхід дозволяє розглядати інфраструктуру не як зовнішню конфігурацію, а як невід'ємну частину архітектури системи, що розвивається разом із прикладним кодом.

Використання Infrastructure as Code є особливо доцільним у serverless-середовищі, де кожен мікросервіс складається з набору керованих хмарних ресурсів. Програмний опис інфраструктури дозволяє забезпечити повторюваність середовища, зменшити залежність від ручної конфігурації та спростити контроль змін у процесі розвитку системи.

Кожен мікросервіс у межах демонстраційної системи розглядається як окрема одиниця інфраструктурної організації, що включає всі ресурси, необхідні для його функціонування: функції обробки запитів, HTTP-інтерфейси, сховища стану, ролі доступу та механізми інтеграції з іншими сервісами.



Для узагальнення практичної структури інфраструктури мікросервісів демонстраційної системи в таблиці 3.3 подано перелік ключових компонентів, що використовуються в реалізації.

Таблиця 3.3

#### Складові інфраструктури мікросервісів демонстраційної системи

Компонент	Роль у системі
AWS Lambda	Виконання бізнес-логіки у відповідь на запити та події
Amazon API Gateway	Надання HTTP-інтерфейсу мікросервісу
Amazon DynamoDB	Зберігання стану та доменних даних
IAM Roles	Обмеження доступу функцій до хмарних ресурсів
Amazon SNS / SQS	Асинхронна взаємодія між мікросервісами
AWS Systems Manager Parameter Store	Зберігання конфігураційних параметрів

Опис інфраструктури у вигляді коду дозволяє формалізувати архітектурні рішення та зберігати їх у репозиторії разом із прикладним кодом мікросервісу. Для реалізації підходу Infrastructure as Code у даній роботі використано інструмент Pulumi, який дає змогу описувати хмарні ресурси з використанням мови програмування та явно задавати залежності між ними.

```
func newLambda(ctx *pulumi.Context, name, path string) (*lambda.Function, error) {
    functionName := resourceName(ctx, name)
    return lambda.NewFunction(ctx, functionName, &lambda.FunctionArgs{
        Role:          lambdaRole.Arn,
        Handler:       pulumi.String("bootstrap"),
        Name:          pulumi.String(functionName),
        Timeout:       pulumi.Int(defaultLambdaTimeout),
        MemorySize:    pulumi.Int(defaultLambdaMemory),
        Runtime:       pulumi.String(lambda.RuntimeCustomAL2023),
        Architectures: pulumi.StringArray{pulumi.String("arm64")},
        Environment:   &lambda.FunctionEnvironmentArgs{Variables: envVariables},
        LoggingConfig: lambda.FunctionLoggingConfigArgs{LogFormat: pulumi.String("JSON")},
        Code:          pulumi.NewAssetArchive(map[string]any{"bootstrap": pulumi.NewFileAsset(path)}),
    })
}
```

Рис. 3.2. Приклад програмного опису інфраструктурного ресурсу

Наведений фрагмент коду (рис. 3.2). ілюструє програмний опис інфраструктурного ресурсу, у якому всі параметри задаються декларативно та зберігаються разом з іншими компонентами системи. Такий підхід забезпечує узгодженість між фактичним станом хмарних ресурсів і їх описом у коді, а також спрощує повторне створення середовища.

Застосування Infrastructure as Code створює передумови для структурованої організації інфраструктури мікросервісів та її подальшого поділу на логічні частини. У демонстраційній системі це дозволяє виокремити незалежні інфраструктурні компоненти з власною відповідальністю та життєвим циклом, що детально розглядається у наступному підрозділі.

### **3.3.2. Поділ інфраструктури на спільний та сервісні шари**

У межах демонстраційної системи інфраструктура мікросервісів організована з урахуванням принципу незалежності сервісів та мінімізації зв'язків між ними. З цією метою всі хмарні ресурси було поділено на окремі логічні групи, кожна з яких має власну відповідальність і життєвий цикл. Такий підхід дозволяє уникнути ситуацій, коли зміни в одному мікросервісі потребують модифікації або повторного застосування інфраструктури інших компонентів системи.

Практична реалізація цього підходу здійснюється шляхом використання кількох незалежних інфраструктурних стеків, описаних за допомогою Infrastructure as Code. Кожен стек відповідає за створення та управління певним набором ресурсів і може розгортатися або змінюватися незалежно від інших. У демонстраційній системі виділено один спільний інфраструктурний стек та окремі стеки для кожного мікросервісу, що узагальнено в таблиці 3.4.

Таблиця 3.4

## Стеки демонстраційної системи та їх призначення

Стек	Призначення
Infrastructure	Створення спільних ресурсів для міжсервісної взаємодії
Booking	Інфраструктура мікросервісу Booking
Inventory	Інфраструктура мікросервісу Inventory

Виділення спільного інфраструктурного стеку зумовлене наявністю ресурсів, які використовуються кількома мікросервісами одночасно. До таких ресурсів належать, зокрема, механізми асинхронної взаємодії, що забезпечують обмін подіями між сервісами. Розміщення цих компонентів у спільному стеку дозволяє уникнути дублювання ресурсів і спрощує їх централізоване управління.

Фізичний розподіл інфраструктури демонстраційної системи та межі керування хмарними ресурсами ілюструє схема, наведена на рис. 3.3. Схема показує, що кожен репозиторій інфраструктури та кожен мікросервіс керує власним набором хмарних ресурсів у межах спільного хмарного середовища.

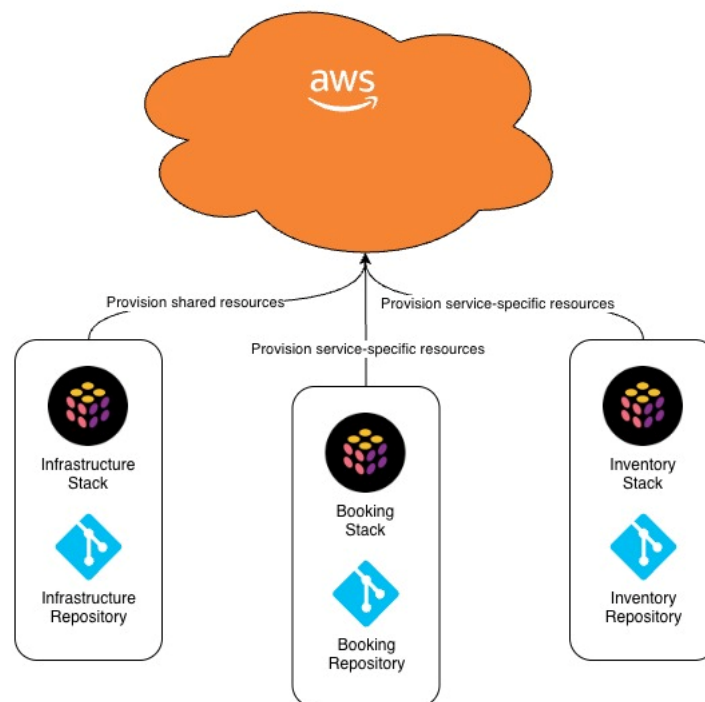


Рис. 3.3. Схема розгортання інфраструктури

Інфраструктура кожного мікросервісу ізольована у власному стеку та містить лише ті ресурси, які безпосередньо належать відповідному сервісу. Такий поділ дозволяє зберегти чіткі межі відповідальності та забезпечує незалежний розвиток мікросервісів, навіть у випадку використання спільних механізмів взаємодії. Розподіл хмарних ресурсів між інфраструктурними стеками узагальнено в таблиці 3.5.

Таблиця 3.5

## Розподіл хмарних ресурсів між інфраструктурними стеками

Ресурс	Infrastructure	Booking	Inventory
AWS Lambda		✓	✓
Amazon API Gateway		✓	✓
Amazon DynamoDB		✓	✓
Amazon SNS Topics	✓		
Amazon SQS Queues		✓	✓
IAM Roles		✓	✓
AWS Systems Manager Parameter Store		✓	✓

Наведений розподіл ресурсів відображає практичний підхід до організації інфраструктури мікросервісів у serverless-середовищі. Спільні ресурси винесені до окремого стеку, тоді як сервісні компоненти залишаються повністю ізольованими. Це дозволяє змінювати конфігурацію окремого мікросервісу без впливу на інші сервіси та без необхідності повторного застосування всієї інфраструктури.

Такий поділ також спрощує контроль доступу до ресурсів, оскільки кожен сервісний стек містить власні ролі та політики доступу, обмежені лише необхідними йому компонентами. У результаті зменшується ризик ненавмисного доступу до ресурсів інших сервісів і підвищується загальна надійність системи.

Наприкінці, використання окремих інфраструктурних стеків створює основу для наочного відображення архітектурної організації системи та взаємозв'язків між її компонентами, що буде проілюстровано у наступному підрозділі.

### **3.3.3. Управління конфігурацією та сервісною адресацією**

У serverless-середовищі мікросервіси не мають фіксованих мережесих адрес і можуть розгортатися незалежно один від одного. За таких умов жорстке задання адрес сервісів у прикладному коді є неприйнятним, оскільки будь-яка зміна інфраструктури одного мікросервісу призводить до необхідності модифікації та повторного розгортання інших компонентів системи. Це суперечить принципу незалежної еволюції мікросервісів.

Для вирішення цієї проблеми в демонстраційній системі реалізовано централізований підхід до управління конфігурацією та сервісною адресацією. Під час розгортання кожен мікросервіс публікує актуальну адресу свого HTTP-інтерфейсу в спільному сховищі конфігурацій. Таким чином, сервісна адреса формується та фіксується на етапі створення інфраструктури, а не на рівні прикладної логіки.

Під час виконання синхронних викликів інші мікросервіси динамічно зчитують відповідні значення зі сховища конфігурацій і використовують їх для побудови HTTP-запитів. Це дозволяє відокремити залежності між сервісами на етапі виконання від залежностей на етапі розгортання та уникнути жорсткого зв'язування між окремими компонентами системи.

Описаний механізм сервісної адресації реалізує просту форму service discovery, адаптовану до serverless-середовища. На відміну від традиційних підходів, заснованих на довгоживучих сервісах або спеціалізованих реєстрах, у даному випадку сервісна адресація інтегрована безпосередньо в процес розгортання інфраструктури та не потребує окремих компонентів для підтримки актуального стану.

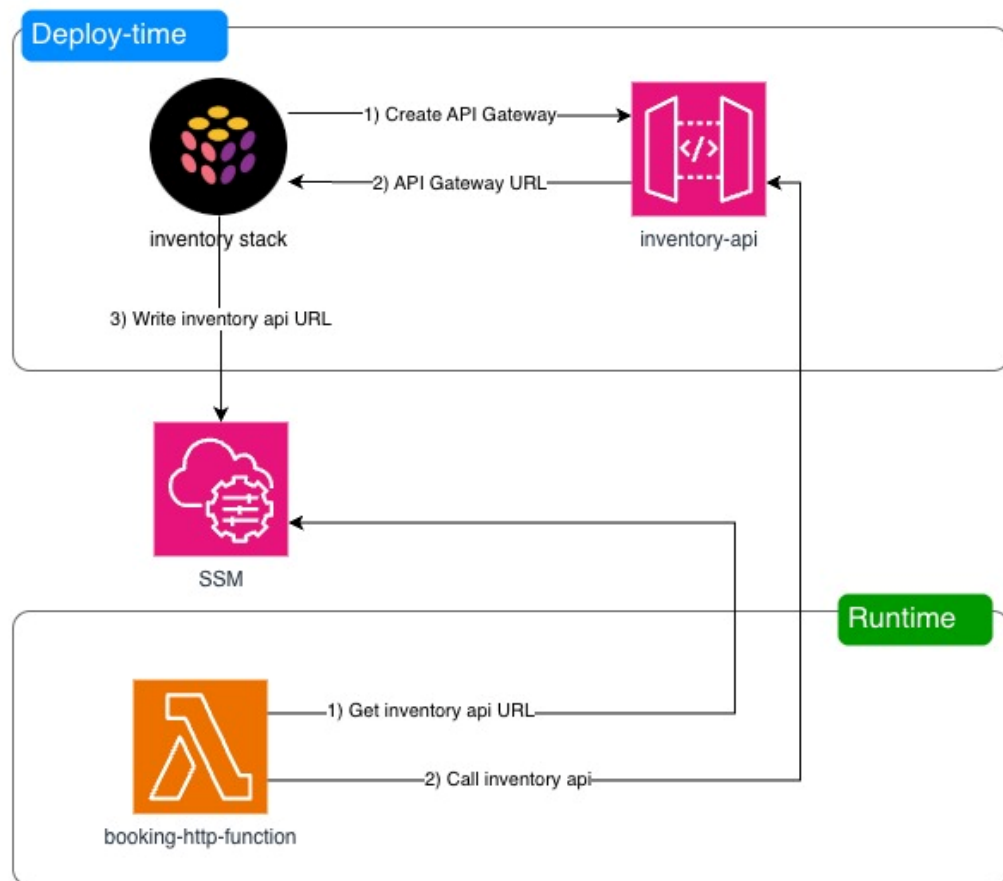


Рис. 3.4. Реалізація сервісної адресації мікросервісів у serverless-середовищі

На рис. 3.4 наведено наочну схему реалізації service discovery у демонстраційній системі, яка відображає розмежування відповідальностей між етапами розгортання та виконання. Схема ілюструє процес публікації адреси API мікросервісу під час створення інфраструктури та подальше використання цієї інформації іншими сервісами під час обробки запитів.

Застосування такого підходу дозволяє зберегти незалежний життєвий цикл мікросервісів і водночас підтримувати синхронну взаємодію у тих сценаріях, де вона необхідна. Управління конфігурацією та сервісною адресацією стає частиною інфраструктурної організації системи, що узгоджується з використанням Infrastructure as Code та загальною архітектурою демонстраційної реалізації.

### 3.4. Загальна архітектура взаємодії мікросервісів

Взаємодія між мікросервісами у демонстраційній системі побудована як поєднання синхронних та асинхронних механізмів обміну, що дозволяє реалізувати наскрізний сценарій резервування ресурсів без порушення меж відповідальності окремих сервісів. Такий підхід дає змогу розглядати систему не лише як набір ізольованих компонентів, а як узгоджену архітектурну композицію, у якій кожен мікросервіс виконує чітко визначену роль.

На високому рівні архітектура взаємодії включає три основні елементи: мікросервіс Booking, мікросервіс Inventory та інфраструктурний шар обміну повідомленнями, який використовується для передачі подій між ними. Ключові бізнес-рішення щодо доступності ресурсів приймаються асинхронно, тоді як синхронні виклики застосовуються лише у випадках, коли необхідно отримати додаткову інформацію без впливу на бізнес-логіку.

Далі наведено узагальнену схему архітектури взаємодії мікросервісів, яка ілюструє основні компоненти системи та напрямки обміну повідомленнями між ними (рис. 3.5).

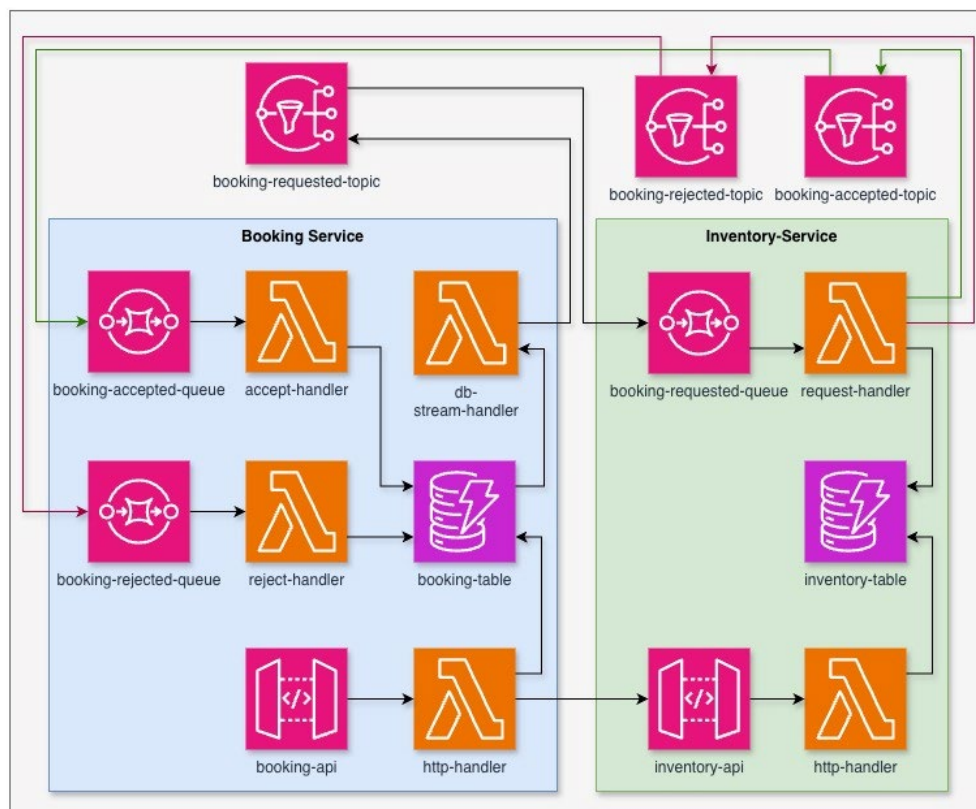


Рис. 3.5. Загальна схема архітектури взаємодії мікросервісів

Наскрізний сценарій резервування ресурсу починається з обробки HTTP-запиту мікросервісом Booking. Після прийому запиту та фіксації початкового стану резервування сервіс ініціює подальшу обробку шляхом публікації події, що відображає намір створення резервування. Ця подія передається через інфраструктурний шар обміну повідомленнями та асинхронно обробляється мікросервісом Inventory.

Отримавши подію, мікросервіс Inventory виконує перевірку доступності відповідного ресурсу з урахуванням часових обмежень та власного стану. За результатами обробки приймається рішення щодо підтвердження або відхилення резервування, після чого публікується відповідна подія, що відображає зміну статусу резервування. Мікросервіс Booking, у свою чергу, реагує на ці події та оновлює стан резервування у власному контексті.

Окрім асинхронної взаємодії, у системі використовується синхронна комунікація між мікросервісами для збагачення даних під час формування відповідей користувачу. Зокрема, мікросервіс Booking може звертатися до Inventory для отримання додаткової інформації про ресурси або їх характеристики. Такі виклики мають виключно інформаційний характер і не впливають на прийняття бізнес-рішень, що дозволяє зберегти автономність сервісів.

Таким чином, загальна архітектура взаємодії мікросервісів поєднує подієвий підхід для реалізації ключових бізнес-процесів та синхронні виклики для допоміжних операцій. Це забезпечує слабку зв'язаність між сервісами, підтримує їх незалежний розвиток і добре узгоджується з особливостями serverless-середовища виконання.

### **3.5. Синхронна взаємодія між мікросервісами**

Синхронна взаємодія у демонстраційній системі реалізується у вигляді HTTP-викликів і використовується для взаємодії зовнішніх клієнтів із системою, а також у межах окремих сценаріїв міжсервісної комунікації. Така



модель дозволяє здійснювати обмін даними у режимі запит–відповідь та формувати результати обробки безпосередньо під час виконання запиту.

У serverless-середовищі синхронні інтерфейси мікросервісів реалізуються на основі керованих хмарних сервісів. Для прийому HTTP-запитів використовується Amazon API Gateway, який виконує роль зовнішньої точки доступу до функціональності мікросервісів. Обробка запитів здійснюється за допомогою функцій AWS Lambda, що дозволяє виконувати прикладну логіку без необхідності розгортання та адміністрування власної серверної інфраструктури.

У демонстраційній системі застосовується підхід API-first, за якого інтерфейси мікросервісів попередньо описуються у вигляді OpenAPI-специфікацій. На основі цих описів автоматично формується конфігурація API Gateway, що забезпечує узгодженість між визначеним контрактом і фактичною реалізацією HTTP-інтерфейсів. Такий підхід спрощує еволюцію API та робить структуру синхронної взаємодії явною частиною архітектури системи. На рис. 3.6 наведено узагальнену схему формування HTTP-інтерфейсу мікросервісу на основі OpenAPI-специфікації.

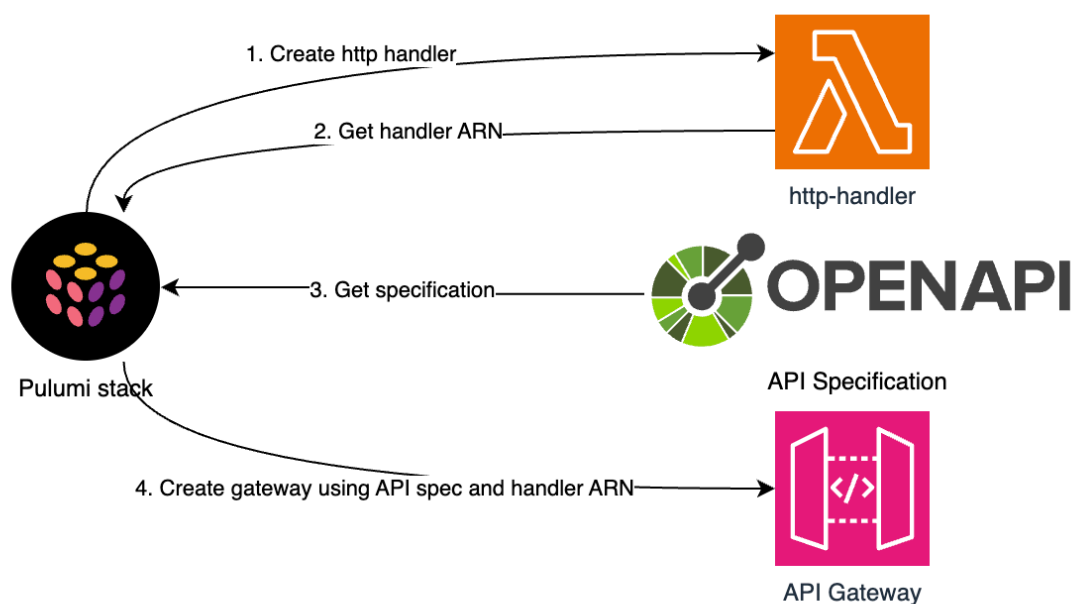


Рис. 3.6. Формування HTTP-інтерфейсу мікросервісу за підходом API-first

Оскільки мікросервіси у serverless-середовищі розгортаються незалежно та не мають фіксованих мережевих адрес, у системі застосовується механізм динамічної адресації сервісів. Під час розгортання кожен мікросервіс публікує адресу свого HTTP-інтерфейсу у централізованому сховищі конфігурацій, звідки ці значення зчитуються іншими сервісами під час виконання синхронних викликів. Це дозволяє уникнути жорсткого зв'язування сервісів на рівні адресації та зберегти незалежність їх розгортання.

Таким чином, синхронна взаємодія у демонстраційній системі реалізована як один зі способів організації обміну даними між компонентами у serverless-середовищі. Поряд із нею в системі використовується асинхронна модель взаємодії, що базується на подієвому обміні повідомленнями та розглядається у наступних підрозділах.

### **3.6. Асинхронна модель взаємодії**

У демонстраційній системі асинхронна взаємодія між мікросервісами використовується як базовий технічний механізм обміну повідомленнями у serverless-середовищі. Вона дозволяє організувати взаємодію між компонентами системи без прямих викликів та жорстких часових залежностей, що є критично важливим для середовищ з подієвим характером виконання.

У межах даного підрозділу бізнес-сценарій резервування використовується виключно як контекст для демонстрації технічних аспектів асинхронної взаємодії. Основна увага приділяється організації подієвих каналів, зв'язуванню керованих хмарних сервісів та особливостям обробки повідомлень у serverless-архітектурі.

#### **3.6.1. Технічна модель асинхронної взаємодії**

Асинхронна взаємодія між мікросервісами у демонстраційній системі реалізується на основі спільних подієвих ресурсів, що створюються в окремому інфраструктурному стеку. Такі ресурси використовуються кількома

мікросервісами одночасно та не належать жодному з них безпосередньо, що дозволяє уникнути концентрації відповідальності за обмін повідомленнями в одному сервісі.

Мікросервіси не здійснюють прямих викликів один одного і не мають інформації про конкретних споживачів подій. Сервіс-ініціатор публікує повідомлення у відповідний подієвий канал, тоді як інші мікросервіси самостійно визначають спосіб та механізм його обробки шляхом створення власних черг повідомлень і прив'язки їх до подієвих каналів.

З технічної точки зору така модель взаємодії передбачає наступну послідовність дій:

1. мікросервіс публікує повідомлення у спільний подієвий канал;
2. мікросервіс-споживач створює власну чергу повідомлень та підписується на подієвий канал;
3. повідомлення доставляється у чергу споживача відповідно до налаштованих правил;
4. функція обробки виконується асинхронно у відповідь на надходження повідомлення з черги.

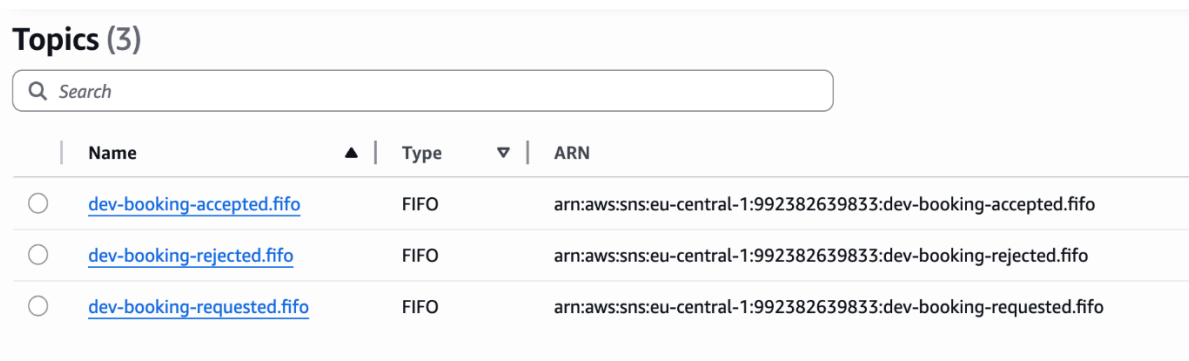
Ключовою особливістю такої технічної моделі є те, що спільні подієві ресурси не містять бізнес-логіки та використовуються виключно як механізм доставки повідомлень. Уся логіка обробки подій, а також правила їх інтерпретації залишаються локалізованими в межах відповідних мікросервісів, що зберігає їх автономність і підтримує незалежний життєвий цикл.

### **3.6.2. Організація подієвих каналів у хмарному середовищі**

Для реалізації асинхронної взаємодії у демонстраційній системі використано керовані хмарні сервіси, які розділяють відповідальність між публікацією, доставкою та обробкою повідомлень. Подієві канали

організовані таким чином, щоб мінімізувати зв'язування між мікросервісами та забезпечити контроль над процесом доставки.

У ролі механізму публікації повідомлень використовується сервіс SNS, який дозволяє одному відправнику передавати повідомлення кільком незалежним споживачам. На рис. 3.7 наведено перелік подієвих каналів, створених у хмарному середовищі для реалізації асинхронної взаємодії між мікросервісами. Кожен канал відповідає окремому типу події та використовується для розповсюдження повідомлень між незалежними компонентами системи.



	Name	Type	ARN
<input type="radio"/>	<a href="#">dev-booking-accepted.fifo</a>	FIFO	arn:aws:sns:eu-central-1:992382639833:dev-booking-accepted.fifo
<input type="radio"/>	<a href="#">dev-booking-rejected.fifo</a>	FIFO	arn:aws:sns:eu-central-1:992382639833:dev-booking-rejected.fifo
<input type="radio"/>	<a href="#">dev-booking-requested.fifo</a>	FIFO	arn:aws:sns:eu-central-1:992382639833:dev-booking-requested.fifo

Рис. 3.7. Подієві канали (SNS topics), створені для асинхронної взаємодії мікросервісів

Для доставки подій до конкретних мікросервісів у системі використовуються окремі черги повідомлень, які створюються в межах сервісних інфраструктурних стеків. Приклад таких черг, налаштованих для асинхронної взаємодії між мікросервісами, наведено на рис. 3.8. Кожен мікросервіс самостійно визначає набір черг та підписується на відповідні подієві канали.

### Queues (3)

<input type="text" value="Search queues by prefix"/>			
	Name	Type	Created
<input type="radio"/>	<a href="#">dev-booking-booking-accepted-queue.fifo</a>	FIFO	2025-12-26T19:06+02:00
<input type="radio"/>	<a href="#">dev-booking-booking-rejected-queue.fifo</a>	FIFO	2025-12-26T19:06+02:00
<input type="radio"/>	<a href="#">dev-inventory-booking-requested-queue.fifo</a>	FIFO	2025-12-25T19:00+02:00

Рис. 3.8. Черги повідомлень (SQS), використані для доставки подій до мікросервісів

Такий поділ дозволяє незалежно керувати параметрами доставки повідомлень, масштабуванням обробників та політиками повторної обробки, не змінюючи логіку публікації подій у мікросервісах.

#### 3.6.3. Зв'язування подієвих каналів та функцій обробки

Асинхронна обробка повідомлень у демонстраційній системі реалізована шляхом зв'язування подієвих каналів, черг повідомлень та функцій виконання у єдиний ланцюг доставки. Механізм публікації подій не взаємодіє безпосередньо з функціями обробки, що дозволяє уникнути жорсткої прив'язки між відправником та отримувачем.

Подієвий канал використовується виключно для розповсюдження повідомлень, тоді як черги повідомлень виконують роль проміжного шару, який забезпечує збереження подій, керування повторною доставкою та контроль паралельності обробки. Функції виконання підключаються до черг як джерела подій та викликаються асинхронно у відповідь на надходження повідомлень.

На рис. 3.9 наведено приклад практичного зв'язування компонентів асинхронної взаємодії на етапі створення інфраструктури. Схема ілюструє, що під час створення черги повідомлень формується підписка на відповідний

подієвий канал, у результаті чого всі повідомлення з каналу автоматично доставляються до черги мікросервісу.

Подальше створення функції обробки супроводжується налаштуванням зв'язку між функцією та чергою повідомлень, яка використовується як джерело подій. Таким чином, асинхронна обробка повідомлень визначається не прямими викликами, а конфігурацією інфраструктурних залежностей, заданих на етапі розгортання.

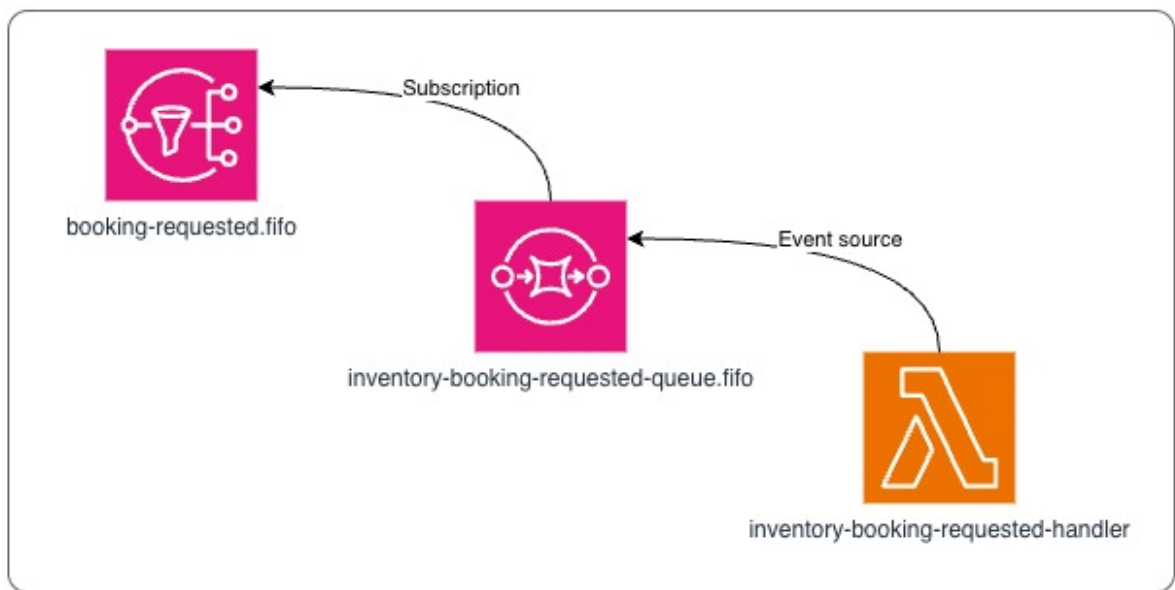


Рис. 3.9. Зв'язування подієвого каналу, черги повідомлень та функції обробки

Використання такого ланцюга доставки дозволяє ізолювати мікросервіси від деталей реалізації обробки подій та забезпечує гнучке масштабування обробників залежно від навантаження. Крім того, черги повідомлень виступають природною точкою контролю для обробки помилок та повторних спроб доставки, що є важливим у serverless-середовищі з короткоживучими функціями.

Асинхронна модель взаємодії передбачає можливість повторної доставки повідомлень та багаторазового виконання обробників, що є штатною

поведінкою керованих хмарних сервісів. Питання забезпечення коректності обробки таких сценаріїв розглядаються у наступному підрозділі.

### **3.7. Узгодженість даних та надійність обробки**

Побудова розподілених систем, до яких належить мікросервісна архітектура, пов'язана з низкою викликів, зокрема із забезпеченням узгодженості стану між окремими компонентами системи. У класичних підходах для вирішення цих проблем застосовуються механізми повторних спроб обробки, ідемпотентна поведінка операцій, шаблони на кшталт *transactional outbox* та інші техніки, спрямовані на підвищення надійності взаємодії між сервісами. У запропонованій *serverless*-реалізації ці підходи не усуваються, а навпаки – отримують практичне втілення з урахуванням особливостей керованої хмарної інфраструктури.

У демонстраційній системі забезпечення узгодженості стану досягається за рахунок поєднання подієвої взаємодії та локальної обробки змін у межах кожного мікросервісу. Кожен сервіс зберігає власний стан у окремому сховищі даних і реагує на події, що надходять від інших сервісів, без використання спільних транзакцій або централізованого керування. Передача змін між сервісами здійснюється через асинхронні повідомлення, які фіксують факти зміни стану, а не виклики конкретних операцій.

У *serverless*-середовищі повторна обробка повідомлень є очікуваною та штатною поведінкою, а не винятковою ситуацією. У демонстраційній системі у разі виникнення помилки під час обробки асинхронного повідомлення виконання функції завершується з помилкою, що призводить до повторної доставки повідомлення та повторної спроби обробки. Такий підхід дозволяє автоматично компенсувати тимчасові збої, пов'язані з нестабільністю зовнішніх залежностей або короткочасними помилками виконання, без необхідності додаткового керування станом обробки. Нижче наведений приклад повторної спроби обробки запиту на резервування ресурсу в демонстраційній системі (рис. 3.10).



Рис. 3.10. Повторна спроба обробки повідомлення

Оскільки повторна доставка повідомлень є штатною поведінкою асинхронної взаємодії, обробка подій у кожному мікросервісі реалізована з урахуванням ідемпотентності операцій. Повторне отримання однієї й тієї ж події не повинно призводити до некоректної зміни стану або дублювання ефектів обробки. У демонстраційній системі це досягається шляхом перевірки поточного стану об'єктів у локальному сховищі даних під час виконання змін, що дозволяє безпечно обробляти події незалежно від кількості їх повторних доставок. Далі наведено приклад ідемпотентної операції резервування ресурсу (рис. 3.11).

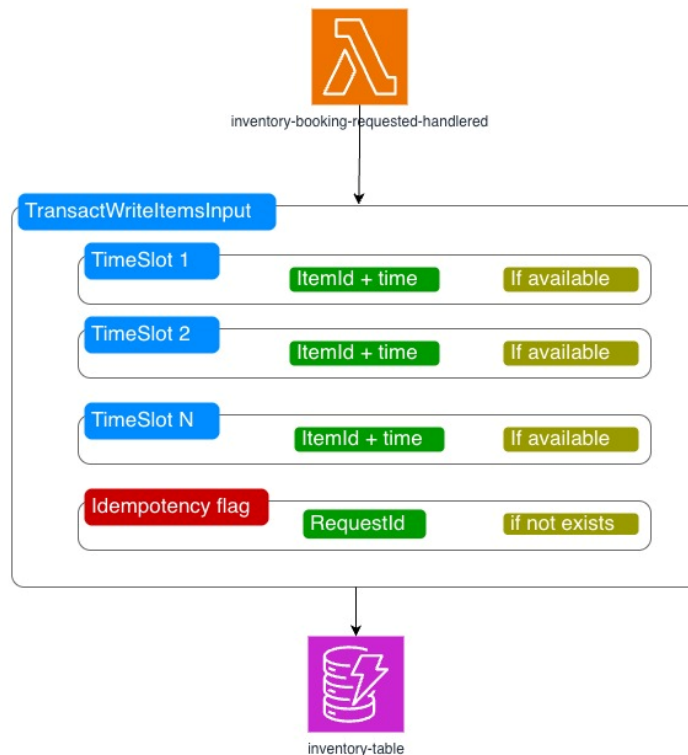


Рис. 3.11. Ідемпотентна операція резервування ресурсу



Окремим викликом у розподілених системах є необхідність узгодженого запису змін стану та публікації подій, що відображають ці зміни. Без додаткових механізмів це може призводити до ситуацій, коли зміна стану збережена у сховищі даних, але відповідна подія не була доставлена, або навпаки. Для розв'язання цієї проблеми у демонстраційній системі застосовано підхід transactional outbox, реалізований на основі локального сховища даних та механізму потоків змін. У межах цього підходу події фіксуються разом зі змінами доменного стану в одному сховищі, після чого їх публікація відбувається асинхронно на основі потоків змін даних. Така модель дозволяє уникнути необхідності розподілених транзакцій і забезпечує узгодженість між збереженим станом та опублікованими подіями навіть у разі збоїв під час обробки. На (рис. 3.12). наведено схему реалізації шаблону transactional outbox у демонстраційній системі.

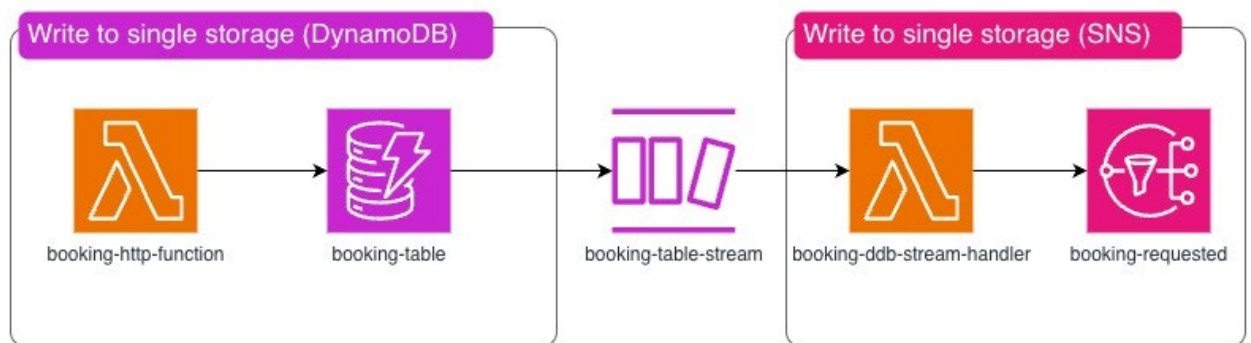


Рис. 3.12. Transactional outbox

Застосування асинхронної взаємодії у поєднанні з механізмами повторної обробки, ідемпотентної логіки та узгодженого збереження змін стану і подій дозволяє ізолювати збої між мікросервісами та зменшити їх взаємний вплив. Помилки або тимчасова недоступність одного сервісу не призводять до зупинки роботи інших компонентів системи, а обробка повідомлень може бути відновлена після усунення проблеми без втрати узгодженості між доменним станом і опублікованими подіями. Така модель

сприяє збереженню автономності мікросервісів і відповідає принципу незалежної еволюції та розгортання компонентів у розподіленій системі.

### Висновки до розділу 3

У межах практичної частини було продемонстровано, що ключові ознаки мікросервісної архітектури можуть бути збережені при реалізації у serverless-середовищі. Незважаючи на зміну моделі виконання та відсутність довгоживучих процесів, мікросервіси зберігають чітко окреслені межі відповідальності, автономність у прийнятті бізнес-рішень та незалежність життєвого циклу.

Практична реалізація показала, що serverless-підхід не скасовує мікросервісну модель, а трансформує спосіб її втілення. Бізнес-логіка мікросервісу розподіляється між окремими функціями та керованими хмарними ресурсами, проте з архітектурної точки зору сервіс продовжує існувати як єдина логічна одиниця з власним станом, API та каналами взаємодії з іншими сервісами.

Ключову роль у збереженні цілісності мікросервісу відіграє використання підходу Infrastructure as Code. Опис інфраструктури у вигляді коду дозволяє розглядати мікросервіс не як набір ізольованих функцій, а як єдину одиницю розгортання, що включає всі необхідні ресурси, конфігурації та інтеграції. Таким чином, мікросервіс у serverless-середовищі залишається цілісним, здатним до незалежного розгортання та еволюції.

Отримані результати підтверджують, що за умови свідомого архітектурного проектування та використання IaC підходу serverless-середовище може виступати ефективним середовищем реалізації мікросервісної архітектури без втрати її базових принципів та ідентичності.

## ВИСНОВКИ

У ході проведеного дослідження розглянуто архітектурні підходи до реалізації мікросервісів у serverless-середовищах з урахуванням специфіки середовища виконання та особливостей керованої хмарної інфраструктури. Основну увагу приділено аналізу того, яким чином ці особливості впливають на практичну реалізацію мікросервісної моделі, а також на архітектурні рішення, що приймаються під час проєктування та розвитку розподілених програмних систем. Такий підхід дозволив розглянути serverless не ізольовано, а в контексті еволюції сучасних архітектур.

Теоретичний аналіз еволюції архітектур програмних систем дозволив простежити послідовний перехід від монолітних рішень до розподілених та мікросервісних моделей, що супроводжувався зростанням вимог до масштабованості, гнучкості та незалежності компонентів. У цьому контексті serverless-підхід розглянуто як сучасний етап розвитку хмарних середовищ виконання, орієнтований на зменшення інфраструктурної складності та автоматизацію керування ресурсами. У процесі дослідження показано, що використання serverless-рішень накладає додаткову специфіку на спосіб реалізації мікросервісів, пов'язану з подієвою моделлю виконання, динамічним масштабуванням і обмеженнями платформи, водночас не змінюючи базових архітектурних принципів мікросервісної архітектури, таких як автономність, ізольованість і чітке розмежування відповідальностей.

У межах практичної частини роботи на прикладі демонстраційної системи підтверджено можливість реалізації ключових характеристик мікросервісної архітектури при використанні керованих сервісів хмарного провайдера. Практична реалізація дозволила проаналізувати, яким чином архітектурні принципи мікросервісів відображаються у serverless-середовищі за умови розподілу бізнес-логіки між окремими функціями та використання керованих хмарних ресурсів. Отримані результати продемонстрували, що за свідомого проєктування мікросервіс може зберігати власний домен,

незалежний життєвий цикл і архітектурну цілісність, залишаючись логічно самостійною одиницею системи.

Окрему увагу в роботі приділено питанням організації інфраструктури та керування залежностями між компонентами системи. Важливу роль у забезпеченні цілісності мікросервісів відіграє застосування підходу Infrastructure as Code, який дозволяє формалізувати інфраструктурні рішення та інтегрувати їх у загальну архітектуру системи. Використання IaC дає змогу розглядати мікросервіс як єдину логічну та інфраструктурну одиницю в межах serverless-середовища, забезпечуючи узгодженість між прикладною логікою, конфігурацією середовища та механізмами інтеграції.

Узагальнення результатів дослідження свідчить про доцільність розгляду serverless-середовищ як одного з можливих варіантів реалізації мікросервісної архітектури в сучасних хмарних системах. Водночас специфіка таких середовищ, різноманітність реалізацій у хмарних провайдерів та динаміка розвитку архітектурних підходів залишають простір для подальшого аналізу й уточнення запропонованих рішень у межах майбутніх досліджень.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, 2020. 432 с.
2. Lewis J., Fowler M. Microservices. *martinfowler.com*. URL: <https://martinfowler.com/articles/microservices.html> (дата звернення: 14.01.2026).
3. Newman S. Building Microservices, 2nd Edition. O'Reilly Media, Incorporated, 2021.
4. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications Co. LLC, 2018.
5. The Distributed Computing Manifesto. *All Things Distributed*. URL: <https://www.allthingsdistributed.com/2022/11/amazon-1998-distributed-computing-manifesto.html> (дата звернення: 14.01.2026).
6. Blog N. T. Rebuilding Netflix Video Processing Pipeline with Microservices. Medium. URL: <https://netflixtechblog.com/rebuilding-netflix-video-processing-pipeline-with-microservices-4e5e6310e359> (дата звернення: 14.01.2026).
7. Erl T. Service-Oriented Architecture (SOA): Concepts, Technology, and Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Prentice Hall PTR, 2005. 792 с.
8. AWS re:Invent 2014 Recap | Amazon Web Services. Amazon Web Services. URL: <https://aws.amazon.com/blogs/developer/aws-reinvent-2014-recap-2/> (дата звернення: 14.01.2026).
9. Fromm K. Why The Future Of Software And Apps Is Serverless. Medium. 08.02.2018. URL: <https://medium.com/a-cloud-guru/why-the-future-of-software-and-apps-is-serverless-reprinted-from-10-15-2012-b92ea572b2ef> (дата звернення: 14.01.2026).

10. Azure Functions overview. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview> (дата звернення: 14.01.2026).

11. Cloud Run functions. Google Cloud. URL: <https://cloud.google.com/functions> (дата звернення: 14.01.2026).

12. What is AWS Lambda? - AWS Lambda. URL: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html> (дата звернення: 14.01.2026)..

13. Velepucha V., Flores P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. IEEE Access. 2023. Т. 11. С. 88339–88358. URL: <https://doi.org/10.1109/access.2023.3305687> (дата звернення: 14.01.2026).

14. Parnas D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM. 1972. Т. 15, № 12. С. 1053–1058. URL: <https://doi.org/10.1145/361598.361623> (дата звернення: 14.01.2026).

15. Relating Edge Computing and Microservices by means of Architecture Approaches and Features, Orchestration, Choreography, and Offloading: A Systematic Literature Review / Lucas Fernando Souza de Castro & Sandro Rigo (2023) URL: <https://arxiv.org/abs/2301.07803> (дата звернення: 14.01.2026).

16. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley Professional, 2003. 560 с.

17. Serverless Computing: A Comprehensive Analysis of Infrastructure Abstraction in Modern Cloud Computing. International Journal For Multidisciplinary Research. 2024. Т. 6, № 6. URL: <https://doi.org/10.36948/ijfmr.2024.v06i06.30737> (дата звернення: 14.01.2026).

18. Fan C.-F., Jindal A., Gerndt M. Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application // Proceedings of the 10th International Conference on Cloud Computing and Services Science. 2020. С. 204–215. – DOI: <https://doi.org/10.5220/0009792702040215>.

19. Mera Menéndez J. A comparison between traditional and serverless technologies in a microservices setting. 2023. URL: <https://doi.org/10.48550/arXiv.2305.13933> (дата звернення: 14.01.2026).

20. Understanding and Remediating Cold Starts: An AWS Lambda Perspective | Amazon Web Services. *Amazon Web Services*. URL: <http://aws.amazon.com/blogs/compute/understanding-and-remediating-cold-starts-an-aws-lambda-perspective/> (дата звернення: 14.01.2026).

21. Configuring AWS Lambda functions - AWS Lambda. AWS. URL: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-functions.html> (дата звернення: 14.01.2026).

22. Попков Є. В., Переяславська С. О. Serverless архітектура як сучасна альтернатива традиційним базам даних // Стратегічні напрямки розвитку науки: фактори впливу та взаємодії : матеріали міжнар. наук.-практ. конф. (м. Хмельницький, 4 квіт. 2025 р.). Хмельницький, 2025. С. 148–153.

23. Наumenko, Т. О., Петренко, А. І. (2021). Аналіз проблем зберігання та обробки даних в безсерверних технологіях. *Technology Audit and Production Reserves*, 2(2(58)), 20–25. <https://doi.org/10.15587/2706-5448.2021.230174>

24. Боровскова Є. А. Вплив використання безсерверної інфраструктури на витрати на підтримку застосунків у серверній частині. *Вісник Херсонського національного технічного університету*. 2025. Т. 2, № 1(92). С. 31–37. URL: <https://doi.org/10.35546/kntu2078-4481.2025.1.2.4> (дата звернення: 14.01.2026).

25. Булах Б., Харченко К. Мікросервісна контейнерна архітектура системи керування потоками даних. *Інфокомунікаційні та комп'ютерні технології*. 2022. Т. 2, № 02. URL: <https://doi.org/10.36994/2788-5518-2021-02-02-17> (дата звернення: 14.01.2026).

26. Загоруйко А. В., Павленко В. І. Архітектура ефективної ci/cd-системи для програмних рішень, заснованих на MSA. *Інфокомунікаційні та комп'ютерні технології*. 2024. Т. 1, № 07. С. 56–60. URL: <https://doi.org/10.36994/2788-5518-2024-01-07-07> (дата звернення: 14.01.2026).



27. Коломійцев О. В., Бульба С. С., Носко С. В. Експериментальна валідація адаптивного методу управління потоками з трирівневою архітектурою back pressure. Системи обробки інформації. 2025. № 4(183). С. 15–22. URL: <https://doi.org/10.30748/soi.2025.183.02> (дата звернення: 14.01.2026).

28. Bashtovyi A., Fechan A. Distributed Transactions in Microservice Architecture: Informed Decision-making Strategies. Вісник Національного університету "Львівська політехніка". Серія Інформаційні системи та мережі. 2024. Т. 15. С. 449–459. URL: <https://doi.org/10.23939/sisn2024.15.449> (дата звернення: 14.01.2026).

29. Дюльгер В.Д., Сорокін А.Р. Аналіз методів інтеграції та узгодження мікросервісів в хмарній архітектурі. Системи управління, навігації та зв'язку. Збірник наукових праць. 2024. Т. 1, № 75. С. 58–60. URL: <https://doi.org/10.26906/sunz.2024.1.058> (дата звернення: 14.01.2026).

30. Kondratiuk T., Naumenko T. Comparison of using Apache OpenWhisk and Google Cloud Functions for development of serverless applications on Google Cloud Platform. System research and information technologies. 2021. No. 3. P. 47–54. URL: <https://doi.org/10.20535/srit.2308-8893.2021.3.04> (date of access: 14.01.2026).

31. Лесной В. І., Антоненко С. В. Безсерверна архітектура для чат-боту. Актуальні проблеми автоматизації та інформаційних технологій. 2021. Т. 25. URL: <https://doi.org/10.15421/432109> (дата звернення: 14.01.2026).

32. Ulichev O., Dorenskyi O., Kulahin V. Innovative Solutions and Benefits of Microservice Architecture for Software Products. Central Ukrainian Scientific Bulletin. Technical Sciences. 2024. Т. 1, № 10(41). С. 16–29. URL: [https://doi.org/10.32515/2664-262x.2024.10\(41\).16-29](https://doi.org/10.32515/2664-262x.2024.10(41).16-29) (дата звернення: 14.01.2026).

## ДОДАТКИ

### Додаток А. Вихідний код додатку

```
// Infrastructure/Makefile
# pulumi
STACK_NAME=dev

deploy:
    cd ./infra && pulumi up --stack ${STACK_NAME} --yes

// Infrastructure/infra/main.go
package main

import (
    "fmt"

    "github.com/pulumi/pulumi-aws/sdk/v7/go/aws/sns"
    "github.com/pulumi/pulumi/sdk/v3/go/pulumi"
)

func main() {
    pulumi.Run(func(ctx *pulumi.Context) error {
        return initTopics(ctx)
    })
}

func initTopics(ctx *pulumi.Context) error {
    steps := []func(*pulumi.Context) error{
        func(context *pulumi.Context) error { return
createFiFoTopic(context, "booking-requested") },
        func(context *pulumi.Context) error { return
createFiFoTopic(context, "booking-accepted") },
        func(context *pulumi.Context) error { return
createFiFoTopic(context, "booking-rejected") },
    }

    for _, fn := range steps {
        if err := fn(ctx); err != nil {
            return err
        }
    }
    return nil
}

func createFiFoTopic(ctx *pulumi.Context, name string) error {
    topicName := resourceName(ctx, fmt.Sprintf("%s.fifo", name))
    topic, err := sns.NewTopic(ctx, topicName, &sns.TopicArgs{
        Name:      pulumi.String(topicName),
        FifoTopic: pulumi.Bool(true),
    })

    if err != nil {
```

```

    return err
}

ctx.Export(fmt.Sprintf("%s.topicArn", name), topic.Arn)
return nil
}

func resourceName(ctx *pulum.Context, name string) string {
    return fmt.Sprintf("%s-%s", ctx.Stack(), name)
}

// Infrastructure/infra/Pulum.dev.yaml
encryptionsalt: <key>
config:
    aws:region: eu-central-1

// Infrastructure/infra/Pulum.yaml
name: infrastructure
description: A minimal AWS Go Pulum program
runtime: go
config:
    pulumi:tags:
        value:
            pulumi:template: aws-go

// Booking/Makefile
# artifacts
ARCH = arm64
BINARY_NAME=bootstrap
TARGET_DIR = bin

# openapi specification
GENERATED_MODEL=api/generated
MODEL=${GENERATED_MODEL}/api
OPENAPI_SPEC_LOCATION=api/schemas//openapi-spec.yaml
GENERATED_SPEC_LOCATION = ${MODEL}/openapi-spec.json

# pulumi
STACK_NAME=dev

clean:
    rm -rf ${TARGET_DIR}
    rm -rf ${GENERATED_MODEL}

build: clean build-oas generate-doc-from-oas
    GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
    ./bin/http/${BINARY_NAME} ./cmd/http/main.go
    GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
    ./bin/ddbs/${BINARY_NAME} ./cmd/ddbs/main.go
    GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
    ./bin/msg/booking/accepted/${BINARY_NAME}
    ./cmd/msg/booking/accepted/main.go
    GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
    ./bin/msg/booking/rejected/${BINARY_NAME}
    ./cmd/msg/booking/rejected/main.go

```

```

build-oas:
  mkdir -p ${MODEL}
  redocly bundle ${OPENAPI_SPEC_LOCATION} -o ${GENERATED_SPEC_LOCATION}
  go run github.com/oapi-codegen/oapi-codegen/v2/cmd/oapi-codegen --
generate types,skip-prune --package api ${GENERATED_SPEC_LOCATION} >
${MODEL}/model-spec.gen.go

```

```

generate-doc-from-oas: build-oas
  redocly build-docs ${GENERATED_SPEC_LOCATION} -o ${MODEL}/spec-
doc.html

```

```

deploy: build
  cd ./infra && pulumi up --stack ${STACK_NAME} -yes

```

```

// Booking/api/schemas/ApiError.yaml
type: object
required:
  - code
  - message
properties:
  code:
    type: string
  message:
    type: string

```

```

// Booking/api/schemas/Booking.yaml
type: object
required:
  - id
  - userId
  - itemId
  - status
  - from
  - to
  - createdAt
properties:
  id:
    type: string
  userId:
    type: string
  itemId:
    type: string
  itemInfo:
    $ref: "ItemInfo.yaml"
  status:
    type: string
  reason:
    type: string
  from:
    type: string
  to:

```

```

        type: string
    createdAt:
        type: string

// Booking/api/schemas/BookingRequest.yaml
type: object
required:
  - itemId
  - from
  - to
properties:
  itemId:
    type: string
  from:
    type: string
  to:
    type: string

// Booking/api/schemas/BookingsSearchResponse.yaml
required:
  - data
  - hasMore
properties:
  data:
    type: array
    items:
      $ref: 'Booking.yaml'
  nextToken:
    type: string
  hasMore:
    type: boolean

// Booking/api/schemas/ItemInfo.yaml
type: object
required:
  - name
  - description
  - type
properties:
  name:
    type: string
  description:
    type: string
  type:
    type: string

// Booking/api/schemas/openapi-spec.yaml
openapi: 3.0.0
info:
  title: Bookings API
  version: 0.0.1
paths:

  /bookings/{id}:

```

```

get:
  tags:
    - Bookings API
  summary: Get user's booking by id
  parameters:
    - in: header
      name: x-user-id
      required: true
      schema:
        type: string
    - in: path
      name: id
      required: true
      schema:
        type: string
  responses:
    '200':
      description: "User's bookings"
      content:
        application/json:
          schema:
            $ref: "Booking.yaml"
    '400':
      description: Bad request.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
    '500':
      description: Internal server error. The server encountered an
unexpected error.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
  x-amazon-apigateway-integration:
    uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
    responses:
      default:
        statusCode: "200"
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        contentHandling: "CONVERT_TO_TEXT"
        type: "aws_proxy"

/bookings:
get:
  tags:
    - Bookings API
  summary: Get user's bookings
  parameters:
    - in: header
      name: x-user-id
      required: true

```

```

      schema:
        type: string
    - in: query
      name: offsetToken
      required: false
      schema:
        type: string
    - in: query
      name: ascending
      required: false
      schema:
        type: boolean

responses:
  '200':
    description: "User's bookings"
    content:
      application/json:
        schema:
          $ref: "BookingsSearchResponse.yaml"
  '400':
    description: Bad request.
    content:
      application/json:
        schema:
          $ref: "ApiError.yaml"
  '500':
    description: Internal server error. The server encountered an
unexpected error.
    content:
      application/json:
        schema:
          $ref: "ApiError.yaml"
x-amazon-apigateway-integration:
  uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
  responses:
    default:
      statusCode: "200"
  passthroughBehavior: "when_no_match"
  httpMethod: "POST"
  contentHandling: "CONVERT_TO_TEXT"
  type: "aws_proxy"

post:
  tags:
    - Bookings API
  summary: Request a new booking
  parameters:
    - in: header
      name: x-user-id
      required: true
      schema:
        type: string
  requestBody:

```

```

        content:
          application/json:
            schema:
              $ref: "BookingRequest.yaml"
      responses:
        '200':
          description: "Created Booking"
          content:
            application/json:
              schema:
                $ref: "Booking.yaml"
        '400':
          description: Bad request.
          content:
            application/json:
              schema:
                $ref: "ApiError.yaml"
        '500':
          description: Internal server error. The server encountered an
unexpected error.
          content:
            application/json:
              schema:
                $ref: "ApiError.yaml"
    x-amazon-apigateway-integration:
      uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
      responses:
        default:
          statusCode: "200"
          passthroughBehavior: "when_no_match"
          httpMethod: "POST"
          contentHandling: "CONVERT_TO_TEXT"
          type: "aws_proxy"

// Booking/cmd/ddbs/main.go
package main

import (
    "booking/cmd/ddbs/handler"
    "booking/cmd/ddbs/handler/mappers"
    "booking/internal/clients"
    "booking/internal/properties"
    "booking/internal/repository/mapper"
    "booking/internal/service"
    "booking/pkg"
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/lambda"
    "go.uber.org/fx"
)

func main() {
    if err := getApp().Start(context.Background()); err != nil {

```



```

        fmt.Printf("Cannot start app: %s", err.Error())
    }
}

func getApp() *fx.App {
    return fx.New(
        fx.Provide(pkg.NewClock),
        fx.Provide(clients.NewSnsClient),
        fx.Provide(mapper.NewDbBookingMapper),
        fx.Provide(service.NewEventsPublisher),
        fx.Provide(properties.NewEnvProperties),

        fx.Provide(fx.Annotate(handler.NewDDBSHandler,
fx.ParamTags(`group:"entityChangeMappers"`))),
        fx.Provide(fx.Annotate(mappers.NewBookingCreatedMapper,
fx.ResultTags(`group:"entityChangeMappers"`))),

        fx.Invoke(start),
    )
}

func start(lifecycle fx.Lifecycle, handler *handler.DDBSHandler) {
    lifecycle.Append(fx.Hook{
        OnStart: func(context.Context) error {
            lambda.Start(handler.HandleStream)
            return nil
        }
    })
}

// Booking/cmd/ddbs/handler/handler.go
package handler

import (
    "booking/cmd/ddbs/handler/model"
    "booking/internal/domain"
    dbModel "booking/internal/repository/model"
    "context"
)

type EntityChangeMapper interface {
    Name() string
    HandledType() string
    MapChange(ctx context.Context, record model.DynamoEventRecord)
    ([]*domain.Event, error)
}

type DDBSHandler struct {
    eventsPublisher domain.EventsPublisher
    mappers          map[string][]EntityChangeMapper
}

func NewDDBSHandler(entityChangeMappers []EntityChangeMapper,
eventsPublisher domain.EventsPublisher) *DDBSHandler {
    mappers := make(map[string][]EntityChangeMapper)
    for _, m := range entityChangeMappers {

```

```

        handledType := m.HandledType()
        mappers[handledType] = append(mappers[handledType], m)
    }

    return &DDBSHandler{eventsPublisher: eventsPublisher, mappers:
mappers}
}

func (h *DDBSHandler) HandleStream(ctx context.Context, dynamoEvent
model.DynamoEvent) error {
    var mappedEvents []*domain.Event
    for _, record := range dynamoEvent.Records {
        if len(record.Change.NewImage) == 0 {
            continue
        }

        entityTypeField, hasEntityType :=
record.Change.NewImage[dbModel.EntityTypeField]
        if !hasEntityType || entityTypeField.S == nil {
            continue
        }

        targetMappers, hasMappers := h.mappers[*entityTypeField.S]
        if !hasMappers {
            continue
        }

        for _, mapper := range targetMappers {
            events, err := mapper.MapChange(ctx, record)
            if err != nil {
                return err
            }

            mappedEvents = append(mappedEvents, events...)
        }
    }

    return h.eventsPublisher.Publish(ctx, mappedEvents)
}

// Booking/cmd/ddbs/handler/model/model.go
package model

import (
    "github.com/aws/aws-sdk-go/service/dynamodb"
)

type DynamoEvent struct {
    Records []DynamoEventRecord `json:"Records"`
}

type DynamoEventRecord struct {
    EventID      string `json:"eventID"`
    EventName    string `json:"eventName"`
}

```

```

    Change      DynamoEventChange `json:"dynamodb"`
}

type DynamoEventChange struct {
    NewImage map[string]*dynamodb.AttributeValue
    `json:"NewImage,omitempty"`
    OldImage map[string]*dynamodb.AttributeValue
    `json:"OldImage,omitempty"`
}

// Booking/cmd/ddbs/handler/mappers/booking_requested_mapper.go
package mappers

import (
    "booking/cmd/ddbs/handler"
    "booking/cmd/ddbs/handler/model"
    "booking/internal/domain"
    "booking/internal/properties"
    "booking/internal/repository/mapper"
    dbModel "booking/internal/repository/model"
    "context"
    "encoding/json"
    "fmt"
)

type BookingRequestedMapper struct {
    props      *properties.Properties
    dbMapper mapper.DbBookingMapper
}

func NewBookingCreatedMapper(props *properties.Properties, dbMapper
mapper.DbBookingMapper) handler.EntityChangeMapper {
    return &BookingRequestedMapper{
        props:      props,
        dbMapper: dbMapper,
    }
}

func (b *BookingRequestedMapper) Name() string {
    return "BookingRequestedMapper"
}

func (b *BookingRequestedMapper) HandledType() string {
    return dbModel.BookingEntityType
}

func (b *BookingRequestedMapper) MapChange(_ context.Context, record
model.DynamoEventRecord) ([]*domain.Event, error) {
    var result []*domain.Event
    if len(record.Change.OldImage) != 0 || len(record.Change.NewImage) ==
0 {
        return result, nil
    }
}

```

```

dbBooking := &dbModel.Booking{}
err := dbBooking.FromDynamoDbAttributeValue(record.Change.NewImage)
if err != nil {
    return nil, err
}

booking := b.dbMapper.FromDbModel(dbBooking)
if booking == nil || booking.BookingStatus !=
domain.BookingStatusRequested {
    return result, nil
}

if len(booking.Id) == 0 || len(booking.ItemId) == 0 {
    fmt.Printf("Skipping booking request. Booking id or item id is
empty: %+v\n", booking)
    return result, nil
}

bytes, err := json.Marshal(&domain.BookingRequested{
    UserId:      booking.UserId,
    BookingId:   booking.Id,
    InventoryId: booking.ItemId,
    RequestedFrom: booking.From,
    RequestedTo: booking.To,
})
if err != nil {
    return nil, err
}

result = append(result, &domain.Event{
    Id:      booking.Id,
    GroupId: booking.ItemId,
    Topic:   b.props.BookingCreatedTopic,
    Content: bytes,
})

return result, nil
}

// Booking/cmd/http/main.go
package main

import (
    "booking/cmd/http/handler"
    apiMapper "booking/cmd/http/mapper"
    "booking/internal/clients"
    "booking/internal/properties"
    "booking/internal/repository"
    "booking/internal/repository/mapper"
    "booking/pkg"
    "booking/pkg/connector"
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/lambda"

```

```

    "go.uber.org/fx"
)

func main() {
    if err := getApp().Start(context.Background()); err != nil {
        fmt.Printf("Cannot start app: %s", err.Error())
    }
}

func getApp() *fx.App {
    return fx.New(
        fx.Provide(pkg.NewClock),
        fx.Provide(properties.NewEnvProperties),
        fx.Provide(clients.NewHttpClient),
        fx.Provide(clients.NewSnsClient),
        fx.Provide(clients.NewDynamoDbClient),
        fx.Provide(clients.NewSsmClient),
        fx.Provide(mapper.NewDbBookingMapper),
        fx.Provide(repository.NewBookingRepository),
        fx.Provide(apiMapper.NewHttpMapper),
        fx.Provide(connector.NewInventoryConnector),
        fx.Provide(handler.NewHttpHandler),
        fx.Invoke(start),
    )
}

func start(lifecycle fx.Lifecycle, handler *handler.HttpHandler) {
    lifecycle.Append(fx.Hook{
        OnStart: func(context.Context) error {
            lambda.Start(handler.Handle)
            return nil
        }
    })
}

// Booking/cmd/http/handler/handler.go
package handler

import (
    "booking/api/generated/api"
    "booking/cmd/http/mapper"
    "booking/internal/domain"
    "booking/pkg"
    "booking/pkg/connector"
    "context"
    "fmt"
    "strconv"

    "github.com/aws/aws-lambda-go/events"
    "github.com/samber/lo"
)

const (
    defaultSliceSize = 10
)

```

```

type Request = events.APIGatewayProxyRequest
type Response = events.APIGatewayProxyResponse

const (
    CodeNotFound          = "NOT_FOUND"
    CodeInvalidRequest    = "INVALID_REQUEST"
    CodeInternalServerError = "INTERNAL_ERROR"
)

type HttpHandler struct {
    clock          pkg.Clock
    apiMapper      mapper.HttpMapper
    bookingRepository domain.BookingRepository
    inventoryConnector connector.InventoryConnector

    routes map[string]func(ctx context.Context, request Request) (Response,
error)
}

func NewHttpHandler(
    clock pkg.Clock,
    apiMapper mapper.HttpMapper,
    bookingRepository domain.BookingRepository,
    inventoryConnector connector.InventoryConnector,
) *HttpHandler {

    handler := &HttpHandler{
        clock:          clock,
        apiMapper:      apiMapper,
        bookingRepository: bookingRepository,
        inventoryConnector: inventoryConnector,
    }

    handler.routes = map[string]func(ctx context.Context, request Request)
(Response, error){
        "POST_/bookings": handler.CreateNewBookings,

        "GET_/bookings":      handler.GetUsersBookings,
        "GET_/bookings/{id}": handler.GetUsersBookingById,
    }

    return handler
}

func (h *HttpHandler) Handle(ctx context.Context, request Request)
(Response, error) {
    route := h.getRout(request)
    f, ok := h.routes[route]
    if !ok {
        return events.APIGatewayProxyResponse{StatusCode: 404, Body: "Rout
not found"}, nil
    }
    return f(ctx, request)
}

```

```

func (h *HttpHandler) CreateNewBookings(ctx context.Context, request
Request) (Response, error) {
    booking, err := h.apiMapper.NewBookingFromCreateRequest(ctx, request)
    fmt.Printf("Booking request: %+v \n", booking)
    if err != nil {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
nil
    }

    err = h.bookingRepository.CreateBooking(ctx, booking)
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalError, err.Error()),
nil
    }

    fmt.Printf("Booking creation return no error \n")

    apiBooking := h.apiMapper.ToApiBooking(ctx, booking)
    fmt.Printf("Api booking response: %+v \n", apiBooking)

    enriched := h.enrichWithItemInfos(ctx, *apiBooking)
    fmt.Printf("enriched Api booking response: %+v \n", enriched)

    return h.apiMapper.ToApiResponse(200, enriched[0]), nil
}

func (h *HttpHandler) GetUsersBookings(ctx context.Context, request
Request) (Response, error) {
    userId, userIdSpecified := request.Headers["x-user-id"]
    if !userIdSpecified {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, "User id not
specified."), nil
    }

    var offsetToken *string
    if token, ok := request.QueryStringParameters["offsetToken"]; ok {
        offsetToken = &token
    }

    var ascending *bool
    if ascendingValue, ok := request.QueryStringParameters["ascending"];
ok {
        ascendingValueParsed, err := strconv.ParseBool(ascendingValue)
        if err != nil {
            return h.apiMapper.ToApiError(400, CodeInvalidRequest, "Invalid
ascending query param value."), nil
        }
        ascending = &ascendingValueParsed
    }

    bookingsSlice, err := h.bookingRepository.BookingsSlice(ctx, userId,
&pkg.SliceRequest{
        Limit:      defaultSliceSize,
        Token:      offsetToken,
        Ascending:  ascending,

```

```

    })
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalServerError, err.Error()),
        nil
    }

    apiSearchResponse := h.apiMapper.ToApiBookingSearchResponse(ctx,
    bookingsSlice)
    apiSearchResponse.Data = h.enrichWithItemInfos(ctx,
    apiSearchResponse.Data...)
    return h.apiMapper.ToApiResponse(200, apiSearchResponse), nil
}

func (h *HttpHandler) GetUsersBookingById(ctx context.Context, request
Request) (Response, error) {
    userId, userIdSpecified := request.Headers["x-user-id"]
    if !userIdSpecified {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, "User id not
specified."), nil
    }

    bookingId, bookingIdSpecified := request.PathParameters["id"]
    if !bookingIdSpecified {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, "Booking id
not specified."), nil
    }

    booking, err := h.bookingRepository.GetBookingById(ctx, userId,
    bookingId)
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalServerError, err.Error()),
        nil
    }

    if booking == nil {
        return h.apiMapper.ToApiError(404, CodeNotFound, "Booking not
found."), nil
    }

    apiBooking := h.apiMapper.ToApiBooking(ctx, booking)
    enrichedApiBooking := h.enrichWithItemInfos(ctx, *apiBooking)[0]

    return h.apiMapper.ToApiResponse(200, enrichedApiBooking), nil
}

func (h *HttpHandler) enrichWithItemInfos(ctx context.Context, bookings
...api.Booking) []api.Booking {
    inventoryItemIds := lo.Map(bookings, func(b api.Booking, _ int) string
{ return b.ItemId })
    inventoryItemIds = lo.Uniq(inventoryItemIds)
    inventoryItems, err :=
h.inventoryConnector.GetInventoryItemsByIds(ctx, inventoryItemIds...)
    if err != nil {
        fmt.Printf("Error getting inventory items: %s \n", err.Error())
        return bookings
    }

```



```

    }

    var enrichedBookings []api.Booking
    for _, booking := range bookings {
        enrichedBooking := booking
        if item, ok := inventoryItems[booking.ItemId]; ok {
            enrichedBooking.ItemInfo = &api.ItemInfo{
                Name:      item.Name,
                Description: item.Description,
                Type:      item.Type,
            }
        }
        enrichedBookings = append(enrichedBookings, enrichedBooking)
    }

    return enrichedBookings
}

func (h *HttpHandler) getRout(request events.APIGatewayProxyRequest)
string {
    return fmt.Sprintf("%s_%s", request.HTTPMethod, request.Resource)
}

//Booking/cmd/http/mapper/http_mapper.go
package mapper

import (
    "booking/api/generated/api"
    "booking/internal/domain"
    "booking/pkg"
    "context"
    "encoding/json"
    "fmt"

    "github.com/aws/aws-lambda-go/events"
    "github.com/google/uuid"
)

const (
    CodeInternalError = "INTERNAL_ERROR"
)

type HttpMapper interface {
    NewBookingFromCreateRequest(ctx context.Context, request
events.APIGatewayProxyRequest) (*domain.Booking, error)

    ToApiBooking(ctx context.Context, booking *domain.Booking)
*api.Booking
    ToApiBookingSearchResponse(ctx context.Context, slice
*pkg.Slice[*domain.Booking]) *api.BookingsSearchResponse

    ToApiResponse(code int, body any) events.APIGatewayProxyResponse
    ToApiError(code int, messageCode, messageText string)
events.APIGatewayProxyResponse
}

```

```

type httpMapper struct {
    clock pkg.Clock
}

func NewHttpMapper(clock pkg.Clock) HttpMapper {
    return &httpMapper{clock: clock}
}

func (h *httpMapper) NewBookingFromCreateRequest(_ context.Context,
request events.APIGatewayProxyRequest) (*domain.Booking, error) {
    userId, userIdSpecified := request.Headers["x-user-id"]
    if !userIdSpecified {
        return nil, fmt.Errorf("user id not specified")
    }

    bookingRequest := &api.BookingRequest{}
    err := json.Unmarshal([]byte(request.Body), bookingRequest)

    fromMilliseconds, err := h.clock.ToMilliseconds(bookingRequest.From)
    if err != nil {
        return nil, fmt.Errorf("invalid 'from' date format")
    }

    toMilliseconds, err := h.clock.ToMilliseconds(bookingRequest.To)
    if err != nil {
        return nil, fmt.Errorf("invalid 'to' date format")
    }

    if fromMilliseconds >= toMilliseconds {
        return nil, fmt.Errorf("'from' date must be before 'to' date")
    }

    id, err := uuid.NewV7()
    if err != nil {
        return nil, fmt.Errorf("unable to generate booking id")
    }

    return &domain.Booking{
        Id:            id.String(),
        UserId:        userId,
        ItemId:        bookingRequest.ItemId,
        BookingStatus: domain.BookingStatusRequested,
        From:          bookingRequest.From,
        To:            bookingRequest.To,
        CreatedAt:     h.clock.NowString(),
    }, nil
}

func (h *httpMapper) ToApiBooking(_ context.Context, booking
*domain.Booking) *api.Booking {
    return &api.Booking{
        Id:            booking.Id,
        UserId:        booking.UserId,
        ItemId:        booking.ItemId,
    }
}

```

```

        From:      booking.From,
        To:        booking.To,
        CreatedAt: booking.CreatedAt,
        Status:    string(booking.BookingStatus),
        Reason:    booking.Reason,
    }
}

func (h *httpMapper) ToApiBookingSearchResponse(ctx context.Context,
slice *pkg.Slice[*domain.Booking]) *api.BookingsSearchResponse {
    var apiBookings []api.Booking
    for _, item := range slice.Data {
        apiBooking := h.ToApiBooking(ctx, item)
        apiBookings = append(apiBookings, *apiBooking)
    }

    return &api.BookingsSearchResponse{
        Data:      apiBookings,
        HasMore:   slice.HasMore,
        NextToken: slice.NextToken,
    }
}

func (h *httpMapper) ToApiResponse(code int, body any)
events.APIGatewayProxyResponse {
    bodyJson, err := json.Marshal(body)
    if err != nil {
        return h.ToApiError(500, CodeInternalServerError, err.Error())
    }
    return events.APIGatewayProxyResponse{StatusCode: code, Body:
string(bodyJson)}
}

func (h *httpMapper) ToApiError(code int, messageCode, messageText
string) events.APIGatewayProxyResponse {
    response := &api.ApiError{Code: messageCode, Message: messageText}
    jsonResponse, _ := json.Marshal(response)
    return events.APIGatewayProxyResponse{StatusCode: code, Body:
string(jsonResponse)}
}

//Booking/cmd/msg/booking/accepted/main.go
package main

import (
    "booking/cmd/msg/booking/accepted/handler"
    "booking/internal/clients"
    "booking/internal/properties"
    "booking/internal/repository"
    "booking/internal/repository/mapper"
    "booking/internal/service"
    "booking/pkg"
    "context"
    "fmt"

```

```

    "github.com/aws/aws-lambda-go/lambda"
    "go.uber.org/fx"
)

func main() {
    if err := getApp().Start(context.Background()); err != nil {
        fmt.Printf("Cannot start app: %s", err.Error())
    }
}

func getApp() *fx.App {
    return fx.New(

        // Common
        fx.Provide(pkg.NewClock),
        fx.Provide(properties.NewEnvProperties),

        // Dynamo DB
        fx.Provide(mapper.NewDbBookingMapper),
        fx.Provide(clients.NewDynamoDbClient),
        fx.Provide(repository.NewBookingRepository),

        //SNS
        fx.Provide(clients.NewSnsClient),
        fx.Provide(service.NewEventsPublisher),

        // Handler
        fx.Provide(handler.NewBookingAcceptedHandler),
        fx.Invoke(start),
    )
}

func start(lifecycle fx.Lifecycle, handler
*handler.BookingAcceptedHandler) {
    lifecycle.Append(fx.Hook{
        OnStart: func(context.Context) error {
            lambda.Start(handler.Handle)
            return nil
        }
    })
}

//Booking/cmd/msg/booking/accepted/handler/handler.go
package handler

import (
    "booking/internal/domain"
    "booking/internal/properties"
    "context"
    "encoding/json"
    "fmt"

    "github.com/aws/aws-lambda-go/events"
    "golang.org/x/sync/errgroup"
)

```

```

type BookingAcceptedHandler struct {
    props          *properties.Properties
    publisher       domain.EventsPublisher
    bookingRepository domain.BookingRepository
}

func NewBookingAcceptedHandler(
    props *properties.Properties,
    publisher domain.EventsPublisher,
    bookingRepository domain.BookingRepository,
) *BookingAcceptedHandler {
    return &BookingAcceptedHandler{
        props:          props,
        publisher:       publisher,
        bookingRepository: bookingRepository,
    }
}

func (b *BookingAcceptedHandler) Handle(ctx context.Context, event
events.SQSEvent) error {
    acceptedRequests, err := b.extractEvents(event)
    if err != nil {
        return err
    }
    acceptedRequestsJsonBytes, _ := json.Marshal(acceptedRequests)
    fmt.Printf("Accepted booking requests received: %s \n",
acceptedRequestsJsonBytes)

    var eg errgroup.Group
    for _, request := range acceptedRequests {
        eg.Go(func() error {
            err := b.bookingRepository.AcceptBooking(ctx, request)
            if businessError := domain.AsBusinessError(err); businessError !=
nil {
                fmt.Printf("Non retrieble error occurred while processing
booking request: %s \n", businessError.Error())
                return nil
            }
            return err
        })
    }

    return eg.Wait()
}

func (b *BookingAcceptedHandler) extractEvents(event events.SQSEvent)
([]*domain.BookingAccepted, error) {
    var result []*domain.BookingAccepted
    for _, record := range event.Records {
        bookingAcceptedEvent := &domain.BookingAccepted{}
        err := json.Unmarshal([]byte(record.Body), bookingAcceptedEvent)
        if err != nil {
            return nil, err
        }
        result = append(result, bookingAcceptedEvent)
    }
}

```



```

var (
    lambdaRole          *iam.Role
    dynamodbTable       *dynamodb.Table
    bookingCreatedSNSTopic *NamedArn
    envVariables         = pulumi.StringMap{}
    infraStackRef        *pulumi.StackReference
)

type NamedArn struct {
    Name string
    Arn  pulumi.StringOutput
}

func main() {
    pulumi.Run(func(ctx *pulumi.Context) error {

        steps := []func(*pulumi.Context) error{
            initCommonEnv,
            initInfraStack,
            initTable,
            initTopics,
            initLambdaRole,
            initHttpApi,
            initDDBStream,
            initMsg,
        }

        for _, fn := range steps {
            if err := fn(ctx); err != nil {
                return err
            }
        }
        return nil
    })
}

func initCommonEnv(ctx *pulumi.Context) error {
    envVariables[properties.EnvKey] = pulumi.String(ctx.Stack())
    return nil
}

func initInfraStack(ctx *pulumi.Context) error {
    target := fmt.Sprintf("organization/infrastructure/%s", ctx.Stack())
    resourceName := resourceName(ctx, "infra-stack-ref")
    ref, err := pulumi.NewStackReference(ctx, resourceName,
        &pulumi.StackReferenceArgs{
            Name: pulumi.String(target),
        })

    if err != nil {
        return err
    }

    infraStackRef = ref
    return nil
}

```

```

}

func initTable(ctx *pulumi.Context) error {
    resourceName := resourceName(ctx, "table")
    table, err := dynamodb.NewTable(ctx, resourceName,
&dynamodb.TableArgs{
        Name:          pulumi.String(resourceName),
        StreamEnabled:  pulumi.BoolPtr(true),
        BillingMode:    pulumi.String("PAY_PER_REQUEST"),
        StreamViewType: pulumi.StringPtr("NEW_AND_OLD_IMAGES"),
        HashKey:       pulumi.String(model.HashKey),
        RangeKey:      pulumi.String(model.RangeKey),
        Attributes:    dynamodb.TableAttributeArray{
            &dynamodb.TableAttributeArgs{Name: pulumi.String(model.HashKey),
Type: pulumi.String(model.HashKeyType)},
            &dynamodb.TableAttributeArgs{Name: pulumi.String(model.RangeKey),
Type: pulumi.String(model.RangeKeyType)},
        },
    }, pulumi.RetainOnDelete(false))

    if err != nil {
        return err
    }

    dynamodbTable = table
    envVariables[properties.TableNameKey] = pulumi.String(resourceName)
    return nil
}

func initTopics(ctx *pulumi.Context) error {
    resourceName := fmt.Sprintf("%s-booking-requested-topic", ctx.Stack())
    bookingRequestedTopicArn :=
infraStackRef.GetOutput(pulumi.String("booking-requested.topicArn"))

    envVariables[properties.BookingRequestedTopic] =
bookingRequestedTopicArn.AsStringOutput()
    bookingCreatedSNSTopic = &NamedArn{Name: resourceName, Arn:
bookingRequestedTopicArn.AsStringOutput()}
    return nil
}

func initLambdaRole(ctx *pulumi.Context) error {
    // Base Role
    roleName := resourceName(ctx, "lambda-role")
    role, err := iam.NewRole(ctx, roleName,
&iam.RoleArgs{AssumeRolePolicy: pulumi.String(lambdaAssumePolicy)})
    if err != nil {
        return err
    }

    rolePolicyAttachmentName := resourceName(ctx, "lambda-basic-execution-
role")
    _, err = iam.NewRolePolicyAttachment(ctx, rolePolicyAttachmentName,
&iam.RolePolicyAttachmentArgs{
        Role:      role.Name,

```



```

    PolicyArn: pulumi.String("arn:aws:iam::aws:policy/service-
role/AWSLambdaBasicExecutionRole"),
})
if err != nil {
    return err
}

// DynamoDB Access
ddbPolicyName := resourceName(ctx, "ddb-access-policy")
ddbPolicy := pulumi.Sprintf(dynamoPolicy, dynamodbTable.Arn,
dynamodbTable.Arn)
_, err = iam.NewRolePolicy(ctx, ddbPolicyName,
&iam.RolePolicyArgs{Role: role.ID(), Policy: ddbPolicy})
if err != nil {
    return err
}

// DynamoDB Stream Access
ddbStreamPolicyName := resourceName(ctx, "ddb-stream-policy")
ddbStreamPolicy := pulumi.Sprintf(dynamoStreamPolicy,
dynamodbTable.StreamArn)
_, err = iam.NewRolePolicy(ctx, ddbStreamPolicyName,
&iam.RolePolicyArgs{Role: role.ID(), Policy: ddbStreamPolicy})
if err != nil {
    return err
}

// SSM Read Access
ssmReadPolicyName := resourceName(ctx, "ssm-read-policy")
ssmReadPolicy := pulumi.Sprintf(ssmReadPolicy)
_, err = iam.NewRolePolicy(ctx, ssmReadPolicyName,
&iam.RolePolicyArgs{Role: role.ID(), Policy: ssmReadPolicy})
if err != nil {
    return err
}

// SNS Access
policyName := fmt.Sprintf("%s-write-policy",
bookingCreatedSNSTopic.Name)
policy := pulumi.Sprintf(snsPublishPolicy, bookingCreatedSNSTopic.Arn)
_, err = iam.NewRolePolicy(ctx, policyName, &iam.RolePolicyArgs{Role:
role.ID(), Policy: policy})
if err != nil {
    return err
}

lambdaRole = role
return nil
}

func initHttpApi(ctx *pulumi.Context) error {
    tplBytes, err := os.ReadFile("../api/generated/api/openapi-spec.json")
    if err != nil {
        return err
    }
}

```

```

    tpl, err :=
template.New("openapi").Option("missingkey=error").Parse(string(tplBytes
))
    if err != nil {
        panic(err)
    }

    region, err := aws.GetRegion(ctx, nil, nil)
    if err != nil {
        return err
    }

    httpFunction, err := newLambda(ctx, "http-function",
"../bin/http/bootstrap")
    if err != nil {
        return err
    }

    renderedSpec := httpFunction.Arn.ApplyT(func(lambdaArn string)
(string, error) {
        var buf bytes.Buffer
        err := tpl.Execute(&buf, map[string]any{
            "AWS_REGION": region.Name,
            "httpHandler": lambdaArn,
        })
        if err != nil {
            return "", err
        }
        return buf.String(), nil
    }).(pulumi.StringOutput)

    apiName := resourceName(ctx, "api")
    api, err := apigateway.NewRestApi(ctx, apiName,
&apigateway.RestApiArgs{
        Name: pulumi.String(apiName),
        Body: renderedSpec,
    })
    if err != nil {
        return err
    }

    specHash := renderedSpec.ApplyT(func(spec string) string {
        sum := sha1.Sum([]byte(spec))
        return hex.EncodeToString(sum[:])
    }).(pulumi.StringOutput)

    deploymentName := resourceName(ctx, "api-deployment")
    deploy, err := apigateway.NewDeployment(ctx, deploymentName,
&apigateway.DeploymentArgs{
        RestApi: api.ID(),
        Triggers: pulumi.StringMap{"openapiSpecHash": specHash},
    }, pulumi.DependsOn([]pulumi.Resource{api}))
    if err != nil {
        return err
    }

```

```

stageName := "api"
_, err = apigateway.NewStage(ctx, stageName, &apigateway.StageArgs{
    RestApi:    api.ID(),
    Deployment: deploy.ID(),
    StageName:  pulumi.String(stageName),
})
if err != nil {
    return err
}

apiUrl := pulumi.Sprintf("https://%s.execute-api.%s.amazonaws.com/%s",
api.ID(), region.Name, stageName)
paramName := fmt.Sprintf("/services/%s/%s/url", ctx.Project(),
ctx.Stack())
_, err = ssm.NewParameter(ctx, resourceName(ctx, "apiUrlParam"),
&ssm.ParameterArgs{
    Name:    pulumi.String(paramName),
    Type:    pulumi.String("String"),
    Value:   apiUrl,
})
if err != nil {
    return err
}

invokePermissionName := resourceName(ctx, "allow-api-gateway-invoke")
_, err = lambda.NewPermission(ctx, invokePermissionName,
&lambda.PermissionArgs{
    Action:    pulumi.String("lambda:InvokeFunction"),
    Function:   httpFunction.Name,
    Principal: pulumi.String("apigateway.amazonaws.com"),
    SourceArn: pulumi.Sprintf("%s/*/*", api.ExecutionArn),
})
if err != nil {
    return err
}

return nil
}

func initMsg(ctx *pulumi.Context) error {
    // Booking accepted handler
    err := subscribeTopic(ctx, &subscribeTopicArgs{
        featureName: "booking-accepted",
        lambdaName:  "booking-accepted-handled",
        lambdaPath:  "../bin/msg/booking/accepted/bootstrap",
        queueName:   "booking-accepted-queue.fifo",
        topicArn:    infraStackRef.GetOutput(pulumi.String("booking-
accepted.topicArn")).AsStringOutput(),
    })
    if err != nil {
        return err
    }

    // Booking rejected handler

```

```

err = subscribeTopic(ctx, &subscribeTopicArgs{
    featureName: "booking-rejected",
    lambdaName:  "booking-rejected-handled",
    lambdaPath:  "../bin/msg/booking/rejected/bootstrap",
    queueName:   "booking-rejected-queue.fifo",
    topicArn:    infraStackRef.GetOutput(pulumi.String("booking-
rejected.topicArn")).AsStringOutput(),
})
if err != nil {
    return err
}

return nil
}

func initDDBStream(ctx *pulumi.Context) error {
    ddbStreamFunction, err := newLambda(ctx, "ddb-stream-handler",
    "../bin/ddbs/bootstrap")
    if err != nil {
        return err
    }

    ddbStreamMappingName := resourceName(ctx, "ddb-stream-mapping")
    _, err = lambda.NewEventSourceMapping(ctx, ddbStreamMappingName,
    &lambda.EventSourceMappingArgs{
        EventSourceArn: dynamodbTable.StreamArn,
        FunctionName:   ddbStreamFunction.Arn,

        StartingPosition: pulumi.String("TRIM_HORIZON"),
        BatchSize:        pulumi.Int(100),
    })
    if err != nil {
        return err
    }

    return nil
}

func newLambda(ctx *pulumi.Context, name, path string)
(*lambda.Function, error) {
    httpFunctionName := resourceName(ctx, name)
    return lambda.NewFunction(ctx, httpFunctionName, &lambda.FunctionArgs{
        Role:          lambdaRole.Arn,
        Handler:       pulumi.String("bootstrap"),
        Name:          pulumi.String(httpFunctionName),
        Timeout:       pulumi.Int(defaultLambdaTimeout),
        MemorySize:    pulumi.Int(defaultLambdaMemory),
        Runtime:       pulumi.String(lambda.RuntimeCustomAL2023),
        Architectures: pulumi.StringArray{pulumi.String("arm64")},
        Environment:   &lambda.FunctionEnvironmentArgs{Variables:
envVariables},
        LoggingConfig: lambda.FunctionLoggingConfigArgs{LogFormat:
pulumi.String("JSON")},
        Code:          pulumi.NewAssetArchive(map[string]any{"bootstrap":
pulumi.NewFileAsset(path)}),

```

```

    })
}

func resourceName(ctx *pulumi.Context, name string) string {
    return fmt.Sprintf("%s-%s-%s", ctx.Stack(), ctx.Project(), name)
}

//Booking/infra/messaging.go
package main

import (
    "fmt"

    "github.com/pulumi/pulumi-aws/sdk/v6/go/aws/iam"
    "github.com/pulumi/pulumi-aws/sdk/v6/go/aws/lambda"
    "github.com/pulumi/pulumi-aws/sdk/v6/go/aws/sns"
    "github.com/pulumi/pulumi-aws/sdk/v6/go/aws/sqs"
    "github.com/pulumi/pulumi/sdk/v3/go/pulumi"
)

type subscribeTopicArgs struct {
    featureName string
    topicArn     pulumi.StringOutput
    queueName    string
    lambdaName   string
    lambdaPath   string
}

func subscribeTopic(ctx *pulumi.Context, args *subscribeTopicArgs) error {
    lambdaHandler, err := newLambda(ctx, args.lambdaName, args.lambdaPath)
    if err != nil {
        return err
    }

    queueResourceName := resourceName(ctx, args.queueName)
    queue, err := sqs.NewQueue(ctx, queueResourceName, &sqs.QueueArgs{
        Name:                pulumi.String(queueResourceName),
        FifoQueue:           pulumi.Bool(true),
        VisibilityTimeoutSeconds: pulumi.Int(30),
    })
    if err != nil {
        return err
    }

    snsToSqsPolicyName := resourceName(ctx, fmt.Sprintf("sns-to-sqs-policy-%s", args.featureName))
    _, err = sqs.NewQueuePolicy(ctx, snsToSqsPolicyName,
    &sqs.QueuePolicyArgs{
        QueueUrl: queue.ID(),
        Policy:    pulumi.Sprintf(snsToSqsPublishPolicy, queue.Arn,
args.topicArn),
    })
    if err != nil {
        return err
    }

```

```

    }

    sqsToLambdaPolicyName := resourceName(ctx, fmt.Sprintf("sqs-to-lambda-
policy-%s", args.featureName))
    _, err = iam.NewRolePolicy(ctx, sqsToLambdaPolicyName,
&iam.RolePolicyArgs{
        Role:    lambdaRole.Name,
        Policy: pulumi.Sprintf(sqsToLambdaPublishPolicy, queue.Arn),
    })
    if err != nil {
        return err
    }

    subscriptionName := resourceName(ctx, fmt.Sprintf("%s-subscription",
args.featureName))
    _, err = sns.NewTopicSubscription(ctx, subscriptionName,
&sns.TopicSubscriptionArgs{
        Topic:    args.topicArn,
        Protocol: pulumi.String("sqs"),
        Endpoint: queue.Arn,

        RawMessageDelivery: pulumi.Bool(true),
    })
    if err != nil {
        return err
    }

    mappingName := resourceName(ctx, fmt.Sprintf("%s-mapping",
args.featureName))
    _, err = lambda.NewEventSourceMapping(ctx, mappingName,
&lambda.EventSourceMappingArgs{
        EventSourceArn: queue.Arn,
        FunctionName:   lambdaHandler.Arn,
        BatchSize:      pulumi.Int(10),
    })

    return nil
}

//Booking/infra/Pulumi.dev.yaml
encryptionsalt: <key>
config: aws:region: eu-central-1

//Booking/infra/Pulumi.yaml
name: booking
description: A minimal AWS Go Pulumi program
runtime: go
config:
  pulumi:tags:
    value:
      pulumi:template: aws-go

//Booking/internal/clients/dynamo.go
package clients

```

```

import (
    "context"
    "strings"

    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/aws/ratelimit"
    "github.com/aws/aws-sdk-go-v2/aws/retry"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb"
)

func NewDynamoDbClient() *dynamodb.Client {
    cfg, _ := config.LoadDefaultConfig(context.Background(),
    WithRetryer(), config.WithHTTPClient(NewHttpClient()))
    return dynamodb.NewFromConfig(cfg)
}

func WithRetryer() config.LoadOptionsFunc {
    return config.WithRetryer(func() aws.Retryer {
        return retry.NewStandard(func(so *retry.StandardOptions) {
            so.MaxAttempts = 10
            so.RateLimiter = ratelimit.NewTokenRateLimit(1_000_000)
            so.Retryables = append(so.Retryables,
&TransactionConflictDetector{})
        })
    })
}

type TransactionConflictDetector struct {
}

func (t *TransactionConflictDetector) IsErrorRetryable(err error)
aws.Ternary {
    if strings.Contains(err.Error(), "TransactionConflict") {
        return aws.TrueTernary
    }
    return aws.UnknownTernary
}

//Booking/internal/clients/http.go
package clients

import (
    "net"
    "net/http"
    "time"
)

func NewHttpClient() *http.Client {
    transport := http.DefaultTransport.(*http.Transport).Clone()
    transport.MaxIdleConns = 128
    transport.MaxConnsPerHost = 128
    transport.MaxIdleConnsPerHost = 128
    transport.IdleConnTimeout = 0
    transport.TLSClientConfig.InsecureSkipVerify = true
}

```

```

transport.DialContext = (&net.Dialer{
    Timeout: 10 * time.Second,
    KeepAlive: 15 * time.Minute,
}).DialContext
return &http.Client{
    Transport: transport,
    Timeout: 10 * time.Second,
}
}

//Booking/internal/clients/sns.go
package clients

import (
    "context"

    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/sns"
)

func NewSnsClient() *sns.Client {
    cfg, _ := config.LoadDefaultConfig(context.Background(),
    config.WithHTTPClient(NewHttpClient()))
    return sns.NewFromConfig(cfg)
}

//Booking/internal/clients/ssm.go
package clients

import (
    "context"

    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/ssm"
)

func NewSsmClient() *ssm.Client {
    cfg, _ := config.LoadDefaultConfig(context.Background(),
    config.WithHTTPClient(NewHttpClient()))
    return ssm.NewFromConfig(cfg)
}

//Booking/internal/domain/errors.go
package domain

import "errors"

type BusinessError struct {
    Message string
}

func (e *BusinessError) Error() string {
    return e.Message
}

```



```

func AsBusinessError(err error) *BusinessError {
    var be *BusinessError
    if errors.As(err, &be) {
        return be
    }
    return nil
}

//Booking/internal/domain/events.go
package domain

import "encoding/json"

type Event struct {
    Id      string
    GroupId string
    Topic   string
    Content json.RawMessage
}

type BookingRequested struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`

    RequestedFrom string `json:"requestedFrom"`
    RequestedTo   string `json:"requestedTo"`
}

type BookingAccepted struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`

    BookingStartTime string `json:"bookingStartTime"`
    BookingEndTime   string `json:"bookingEndTime"`
}

type BookingRejected struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`

    RejectionReason string `json:"rejectionReason"`
}

type BookingCancellationRequested struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`
}

type BookingCancellationAccepted struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`
}

```

```

}

type BookingCancellationRejected struct {
    UserId      string `json:"userId"`
    BookingId   string `json:"bookingId"`
    InventoryId string `json:"inventoryId"`

    RejectionReason string `json:"rejectionReason"`
}

//Booking/internal/domain/model.go
package domain

type BookingStatus string

const (
    BookingStatusRequested BookingStatus = "Requested"
    BookingStatusAccepted  BookingStatus = "Accepted"
    BookingStatusRejected  BookingStatus = "Rejected"
)

type Booking struct {
    Id      string
    UserId  string
    ItemId  string

    BookingStatus BookingStatus
    Reason         *string

    From      string
    To        string
    CreatedAt string
}

//Booking/internal/domain/repository.go
package domain

import (
    "booking/pkg"
    "context"
)

type BookingRepository interface {
    CreateBooking(ctx context.Context, booking *Booking) error
    AcceptBooking(ctx context.Context, bookingEvent *BookingAccepted)
    error
    RejectBooking(ctx context.Context, bookingEvent *BookingRejected)
    error

    GetBookingById(ctx context.Context, userId, bookingId string)
    (*Booking, error)
    BookingsSlice(ctx context.Context, userId string, sliceRequest
    *pkg.SliceRequest) (*pkg.Slice[*Booking], error)
}

```

```

// Booking/internal/domain/service.go
package domain

import (
    "context"
)

type EventsPublisher interface {
    Publish(ctx context.Context, events []*Event) error
}

//Booking/internal/properties/properties.go
package properties

import "os"

const (
    EnvKey          = "ENV"
    TableNameKey    = "TABLE_NAME"
    BookingRequestedTopic = "BOOKING_REQUESTED_TOPIC"
)

type Properties struct {
    Env          string
    TableName    string
    BookingCreatedTopic string
}

func NewEnvProperties() *Properties {
    return &Properties{
        Env:          os.Getenv(EnvKey),
        TableName:    os.Getenv(TableNameKey),
        BookingCreatedTopic: os.Getenv(BookingRequestedTopic),
    }
}

//Booking/internal/repository/booking_repository.go
package repository

import (
    "booking/internal/domain"
    "booking/internal/properties"
    "booking/internal/repository/mapper"
    "booking/internal/repository/model"
    "booking/pkg"
    "context"
    "encoding/base64"
    "errors"
    "fmt"

    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"

```

```

    "github.com/samber/lo"
)

type bookingRepository struct {
    dbMapper    mapper.DbBookingMapper
    dbClient    *dynamodb.Client
    properties  *properties.Properties
}

func NewBookingRepository(
    dbMapper mapper.DbBookingMapper,
    dbClient *dynamodb.Client,
    properties *properties.Properties,
) domain.BookingRepository {
    return &bookingRepository{
        dbMapper:    dbMapper,
        dbClient:    dbClient,
        properties:  properties,
    }
}

func (b *bookingRepository) CreateBooking(ctx context.Context, booking
domain.Booking) error {
    dbModel := b.dbMapper.ToDbModel(booking)
    avModel, err := dbModel.AsAttributeValues()
    if err != nil {
        return err
    }

    _, err = b.dbClient.PutItem(ctx, &dynamodb.PutItemInput{
        Item:                avModel,
        TableName:           aws.String(b.properties.TableName),
        ConditionExpression: aws.String("attribute_not_exists(#PK) AND
attribute_not_exists(#SK)"),
        ExpressionAttributeNames: map[string]string{"#PK": model.HashKey,
"#SK": model.RangeKey},
    })

    return err
}

func (b *bookingRepository) AcceptBooking(ctx context.Context,
bookingEvent *domain.BookingAccepted) error {
    _, err := b.dbClient.UpdateItem(ctx, &dynamodb.UpdateItemInput{
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.BookingPk(bookingEvent.UserId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.BookingSk(bookingEvent.BookingId)},
        },
        TableName:
aws.String(b.properties.TableName),
        ReturnValuesOnConditionCheckFailure:
types.ReturnValuesOnConditionCheckFailureAllOld,
    })

```

```

        UpdateExpression:          aws.String("SET #Status =
:status, #From = :from, #To = :to"),
        ConditionExpression:
aws.String("attribute_exists(#Status) and #Status = :expectedStatus"),
        ExpressionAttributeNames: map[string]string{
            "#Status": model.BookingStatusField,
            "#From":   model.BookingFromField,
            "#To":     model.BookingToField,
        },
        ExpressionAttributeValues: map[string]types.AttributeValue{
            ":status":      &types.AttributeValueMemberS{Value:
string(domain.BookingStatusAccepted)},
            ":from":       &types.AttributeValueMemberS{Value:
bookingEvent.BookingStartTime},
            ":to":        &types.AttributeValueMemberS{Value:
bookingEvent.BookingEndTime},
            ":expectedStatus": &types.AttributeValueMemberS{Value:
string(domain.BookingStatusRequested)},
        },
    })

    if err != nil {
        if conditionalException := asConditionalCheckFailedException(err);
conditionalException != nil {

            // Expected status mismatch case
            if len(conditionalException.Item) > 0 {
                dbBooking := &model.Booking{}
                if mappingError :=
dbBooking.FromAttributeValues(conditionalException.Item); mappingError
!= nil {
                    return mappingError
                }

                booking := b.dbMapper.FromDbModel(dbBooking)
                if booking.BookingStatus == domain.BookingStatusAccepted {
                    // Important retry. Booking already accepted
                    return nil
                }

                // Incompatible booking status
                message := fmt.Sprintf("Booking with id %s cannot be accepted
from incompatible status %s", booking.Id, booking.BookingStatus)
                return &domain.BusinessError{Message: message}
            }

            // Booking not found
            message := fmt.Sprintf("User's %s booking %s not found",
bookingEvent.UserId, bookingEvent.BookingId)
            return &domain.BusinessError{Message: message}
        }
    }
    return err
}

```

```

func (b *bookingRepository) RejectBooking(ctx context.Context,
bookingEvent *domain.BookingRejected) error {
    _, err := b.dbClient.UpdateItem(ctx, &dynamodb.UpdateItemInput{
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.BookingPk(bookingEvent.UserId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.BookingSk(bookingEvent.BookingId)},
        },
        TableName:
aws.String(b.properties.TableName),
        ReturnValuesOnConditionCheckFailure:
types.ReturnValuesOnConditionCheckFailureAllOld,
        UpdateExpression:
            aws.String("SET #Status =
:status, #Reason = :reason"),
        ConditionExpression:
aws.String("attribute_exists(#Status) and #Status = :expectedStatus"),
        ExpressionAttributeNames: map[string]string{
            "#Status": model.BookingStatusField,
            "#Reason": model.BookingReasonField,
        },
        ExpressionAttributeValues: map[string]types.AttributeValue{
            ":status": &types.AttributeValueMemberS{Value:
string(domain.BookingStatusRejected)},
            ":reason": &types.AttributeValueMemberS{Value:
bookingEvent.RejectionReason},
            ":expectedStatus": &types.AttributeValueMemberS{Value:
string(domain.BookingStatusRequested)},
        },
    })

    if err != nil {
        if conditionalException := asConditionalCheckFailedException(err);
conditionalException != nil {

            // Expected status mismatch case
            if len(conditionalException.Item) > 0 {
                dbBooking := &model.Booking{}
                if mappingError :=
dbBooking.FromAttributeValues(conditionalException.Item); mappingError
!= nil {
                    return mappingError
                }

                booking := b.dbMapper.FromDbModel(dbBooking)
                if booking.BookingStatus == domain.BookingStatusRejected {
                    // Important retry. Booking already rejected
                    return nil
                }

                // Incompatible booking status
                message := fmt.Sprintf("Booking with id %s cannot be rejected
from incompatible status %s", booking.Id, booking.BookingStatus)
                return &domain.BusinessError{Message: message}
            }
        }
    }
}

```

```

        // Booking not found
        message := fmt.Sprintf("User's %s booking %s not found",
            bookingEvent.UserId, bookingEvent.BookingId)
        return &domain.BusinessError{Message: message}
    }
}

return err
}

func (b *bookingRepository) GetBookingById(ctx context.Context, userId,
    bookingId string) (*domain.Booking, error) {
    out, err := b.dbClient.GetItem(ctx, &dynamodb.GetItemInput{
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
                model.BookingPk(userId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
                model.BookingSk(bookingId)},
        },
        TableName:      aws.String(b.properties.TableName),
        ConsistentRead: aws.Bool(true),
    })
    if err != nil {
        return nil, err
    }

    if len(out.Item) == 0 {
        return nil, nil
    }

    dbModel := &model.Booking{}
    err = dbModel.FromAttributeValues(out.Item)
    if err != nil {
        return nil, err
    }

    return b.dbMapper.FromDbModel(dbModel), nil
}

func (b *bookingRepository) BookingsSlice(ctx context.Context, userId
    string, sliceRequest *pkg.SliceRequest) (*pkg.Slice[*domain.Booking],
    error) {
    keyCondition :=
        expression.Key(model.HashKey).Equal(expression.Value(model.BookingPk(userId)))
    expr, err :=
        expression.NewBuilder().WithKeyCondition(keyCondition).Build()
    if err != nil {
        return nil, err
    }

    paginator := dynamodb.NewQueryPaginator(b.dbClient,
    &dynamodb.QueryInput{
        TableName:      aws.String(b.properties.TableName),
        ExpressionAttributeNames: expr.Names(),
    })

```

```

        ExpressionAttributeValues: expr.Values(),
        KeyConditionExpression:    expr.KeyCondition(),
        FilterExpression:         expr.Filter(),
        ScanIndexForward:
aws.Bool(lo.FromPtr(sliceRequest.Ascending)),
        ExclusiveStartKey:        TokenToKey(sliceRequest.Token),
        Limit:                    aws.Int32(sliceRequest.Limit),
    })
    if !paginator.HasMorePages() {
        return &pkg.Slice[*domain.Booking]{HasMore: false}, nil
    }

    page, err := paginator.NextPage(ctx)
    if err != nil {
        return nil, err
    }

    var data []*domain.Booking
    for _, item := range page.Items {
        tournamentDb := &model.Booking{}
        err := tournamentDb.FromAttributeValues(item)
        if err != nil {
            return nil, err
        }

        data = append(data, b.dbMapper.FromDbModel(tournamentDb))
    }

    return &pkg.Slice[*domain.Booking]{
        Data:        data,
        HasMore:     paginator.HasMorePages(),
        NextToken:   KeyToToken(page.LastEvaluatedKey),
    }, nil
}

func TokenToKey(token *string) map[string]types.AttributeValue {
    if token == nil {
        return nil
    }
    sDec, _ := base64.StdEncoding.DecodeString(*token)
    key, _ := attributevalue.UnmarshalMapJSON(sDec)
    return key
}

func KeyToToken(key map[string]types.AttributeValue) *string {
    if len(key) == 0 {
        return nil
    }
    data, _ := attributevalue.MarshalMapJSON(key)
    return lo.ToPtr(base64.StdEncoding.EncodeToString(data))
}

func asConditionalCheckFailedException(err error)
*types.ConditionalCheckFailedException {
    if err == nil {

```



```

        return nil
    }

    var e *types.ConditionalCheckFailedException
    if errors.As(err, &e) {
        return e
    }

    return nil
}

//Booking/internal/repository/mapper/booking.go
package mapper

import (
    "booking/internal/domain"
    "booking/internal/repository/model"
)

type DbBookingMapper interface {
    ToDbModel(booking *domain.Booking) *model.Booking
    FromDbModel(booking *model.Booking) *domain.Booking
}

type dbBookingMapper struct {
}

func NewDbBookingMapper() DbBookingMapper {
    return &dbBookingMapper{}
}

func (d *dbBookingMapper) ToDbModel(booking *domain.Booking)
*model.Booking {
    return &model.Booking{
        PK:          model.BookingPk(booking.UserId),
        SK:          model.BookingSk(booking.Id),
        EntityType:  model.BookingEntityType,
        Id:          booking.Id,
        UserId:      booking.UserId,
        ItemId:      booking.ItemId,
        BookingStatus: string(booking.BookingStatus),
        Reason:      booking.Reason,
        From:        booking.From,
        To:          booking.To,
        CreatedAt:   booking.CreatedAt,
    }
}

func (d *dbBookingMapper) FromDbModel(booking *model.Booking)
*domain.Booking {
    return &domain.Booking{
        Id:          booking.Id,
        UserId:      booking.UserId,
        ItemId:      booking.ItemId,
        BookingStatus: domain.BookingStatus(booking.BookingStatus),
    }
}

```

```

        Reason:      booking.Reason,
        From:        booking.From,
        To:          booking.To,
        CreatedAt:    booking.CreatedAt,
    }
}

//Booking/internal/repository/model/booking.go
package model

import (
    "fmt"

    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
    "github.com/aws/aws-sdk-go/service/dynamodb"
    "github.com/aws/aws-sdk-go/service/dynamodb/dynamodbattribute"
)

const (
    BookingPkTemplate = "User:%s"
    BookingSkTemplate = "Booking:%s"
    BookingEntityType = "Booking"

    BookingStatusField = "bookingStatus"
    BookingReasonField = "reason"
    BookingFromField   = "from"
    BookingToField     = "to"
)

func BookingPk(userId string) string {
    return fmt.Sprintf(BookingPkTemplate, userId)
}

func BookingSk(bookingId string) string {
    return fmt.Sprintf(BookingSkTemplate, bookingId)
}

type Booking struct {
    PK          string `dynamodbav:"PK"`
    SK          string `dynamodbav:"SK"`
    EntityType  string `dynamodbav:"entityType"`

    Id          string `dynamodbav:"id"`
    UserId      string `dynamodbav:"userId"`
    ItemId      string `dynamodbav:"itemId"`

    BookingStatus string `dynamodbav:"bookingStatus"`
    Reason        *string `dynamodbav:"reason,omitempty"`

    From        string `dynamodbav:"from"`
    To          string `dynamodbav:"to"`
    CreatedAt   string `dynamodbav:"createdAt"`
}

```

```

func (b *Booking) AsAttributeValues() (map[string]types.AttributeValue,
error) {
    return attributevalue.MarshalMap(b)
}

func (b *Booking) FromAttributeValues(av
map[string]types.AttributeValue) error {
    return attributevalue.UnmarshalMap(av, b)
}

func (b *Booking) FromDynamoDbAttributeValue(av
map[string]*dynamodb.AttributeValue) error {
    return dynamodbattribute.UnmarshalMap(av, b)
}

func (b *Booking) AsDynamoDbAttributeValue()
(map[string]*dynamodb.AttributeValue, error) {
    return dynamodbattribute.MarshalMap(b)
}

//Booking/internal/repository/model/fields.go
package model

const (
    HashKey      = "PK"
    HashKeyType  = "S"

    RangeKey     = "SK"
    RangeKeyType = "S"

    EntityTypeField = "entityType"
)

//Booking/internal/service/events_publisher.go
package service

import (
    "booking/internal/domain"
    "context"

    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/service/sns"
    "github.com/samber/lo"
    "golang.org/x/sync/errgroup"
)

type eventsPublisher struct {
    snsClient *sns.Client
}

func NewEventsPublisher(snsClient *sns.Client) domain.EventsPublisher {
    return &eventsPublisher{snsClient: snsClient}
}

```

```

func (e *eventsPublisher) Publish(ctx context.Context, events
[*domain.Event) error {
    var inputs []*sns.PublishInput
    for _, event := range events {
        inputs = append(inputs, &sns.PublishInput{
            MessageDeduplicationId: aws.String(event.Id),
            MessageGroupId:        aws.String(event.GroupId),
            TopicArn:              aws.String(event.Topic),
            Message:               aws.String(string(event.Content)),
        })
    }

    var eg errgroup.Group
    var uniqueInputs = lo.UniqBy(inputs, func(pi *sns.PublishInput) string
{ return lo.FromPtr(pi.MessageDeduplicationId) })
    for _, input := range uniqueInputs {
        eg.Go(func() error {
            _, err := e.snsClient.Publish(ctx, input)
            return err
        })
    }

    return eg.Wait()
}

//Booking/pkg/connector/inventory.go
package connector

import (
    "booking/internal/domain"
    "booking/internal/properties"
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/service/ssm"
    "github.com/samber/lo"
)

const (
    lookupUrl = "/inventory/items:lookup"
)

type InventoryItemsLookupRequest struct {
    Ids []string `json:"ids"`
}

type InventoryItemsLookupResponse struct {
    Data []*InventoryItem `json:"data"`
}

type InventoryItem struct {

```

```

    Id            string `json:"id"`
    Name          string `json:"name"`
    Type          string `json:"type"`
    Description    string `json:"description"`
}

type InventoryConnector interface {
    GetInventoryItemsByIds(ctx context.Context, ids ...string)
    (map[string]*InventoryItem, error)
}

type inventoryConnector struct {
    httpClient *http.Client
    ssmClient  *ssm.Client
    props      *properties.Properties
}

func NewInventoryConnector(httpClient *http.Client, ssmClient
*ssm.Client, props *properties.Properties) InventoryConnector {
    return &inventoryConnector{
        httpClient: httpClient,
        ssmClient:  ssmClient,
        props:      props,
    }
}

func (i *inventoryConnector) GetInventoryItemsByIds(ctx context.Context,
ids ...string) (map[string]*InventoryItem, error) {
    inventoryBaseUrl, err := i.getInventoryServiceBaseUrl(ctx)
    if err != nil {
        return nil, err
    }

    body, err := json.Marshal(&InventoryItemsLookupRequest{Ids: ids})
    if err != nil {
        return nil, err
    }

    url := fmt.Sprintf("%s%s", inventoryBaseUrl, lookupUrl)
    httpReq, err := http.NewRequestWithContext(ctx, http.MethodPost, url,
bytes.NewReader(body))
    if err != nil {
        return nil, err
    }

    httpReq.Header.Set("Content-Type", "application/json")
    httpReq.Header.Set("Accept", "application/json")

    resp, err := i.httpClient.Do(httpReq)
    if err != nil {
        return nil, err
    }

    defer func() {
        _ = resp.Body.Close()
    }()
}

```

```

}()

if resp.StatusCode != 200 {
    return nil, fmt.Errorf("unexpected status %d", resp.StatusCode)
}

var result *InventoryItemsLookupResponse
dec := json.NewDecoder(resp.Body)
dec.DisallowUnknownFields()

if err := dec.Decode(&result); err != nil {
    return nil, err
}

return lo.SliceToMap(result.Data, func(item *InventoryItem) (string,
*InventoryItem) {
    return item.Id, item
}), nil
}

func (i *inventoryConnector) getInventoryServiceBaseUrl(ctx
context.Context) (string, error) {
    out, err := i.ssmClient.GetParameter(ctx, &ssm.GetParameterInput{
        Name: aws.String(fmt.Sprintf("/services/inventory/%s/url",
i.props.Env)),
    })

    if err != nil {
        return "", err
    }

    if out == nil || out.Parameter == nil || out.Parameter.Value == nil {
        return "", &domain.BusinessError{Message: "inventory service url not
found"}
    }

    return *out.Parameter.Value, nil
}

//Booking/pkg/clock.go
package pkg

import "time"

type Clock interface {
    NowString() string
    NowDateTime() (date, time string)
    NowMilliseconds() int64
    ToString(milliseconds int64) string
    ToMilliseconds(dateTime string) (int64, error)
    ToSeconds(dateTime string) (int64, error)
    IsValid(dateTime string) bool
    Split(dateTime string) (date, time string)
    Parse(dateTime string) (time.Time, error)
}

```

```

const (
    defaultDateLayout      = "2006-01-02"
    defaultTimeLayout      = "15:04:05.000"
    defaultDateTimeLayout = defaultDateLayout + "T" + defaultTimeLayout
)

type LayoutBasedClock struct {
    dateTimeLayout string
    timeLayout      string
    dateLayout      string
}

func NewClock() Clock {
    return &LayoutBasedClock{
        dateTimeLayout: defaultDateTimeLayout,
        timeLayout:      defaultTimeLayout,
        dateLayout:      defaultDateLayout,
    }
}

func (d *LayoutBasedClock) NowString() string {
    return time.Now().UTC().Format(d.dateTimeLayout)
}

func (d *LayoutBasedClock) NowDateTime() (date, time string) {
    return d.Split(d.NowString())
}

func (d *LayoutBasedClock) NowMilliseconds() int64 {
    return time.Now().UnixMilli()
}

func (d *LayoutBasedClock) ToString(milliseconds int64) string {
    return time.UnixMilli(milliseconds).UTC().Format(d.dateTimeLayout)
}

func (d *LayoutBasedClock) ToMilliseconds(dateTime string) (int64,
error) {
    if value, err := time.Parse(d.dateTimeLayout, dateTime); err == nil {
        return value.UnixMilli(), nil
    } else {
        return 0, err
    }
}

func (d *LayoutBasedClock) ToSeconds(dateTime string) (int64, error) {
    if value, err := time.Parse(d.dateTimeLayout, dateTime); err == nil {
        return value.Unix(), nil
    } else {
        return 0, err
    }
}

func (d *LayoutBasedClock) IsValid(dateTime string) bool {

```

```

_, err := time.Parse(d.dateTimeLayout, dateTime)
return err == nil
}

func (d *LayoutBasedClock) Split(dateTime string) (string, string) {
    if value, err := time.Parse(d.dateTimeLayout, dateTime); err == nil {
        return value.Format(d.dateLayout), value.Format(d.timeLayout)
    } else {
        panic(err)
    }
}

func (d *LayoutBasedClock) Parse(dateTime string) (time.Time, error) {
    return time.Parse(d.dateTimeLayout, dateTime)
}

//Booking/pkg/slice.go
package pkg

type SliceRequest struct {
    Limit      int32
    Token      *string
    Ascending  *bool
}

type Slice[T any] struct {
    Data       []T
    HasMore    bool
    NextToken  *string
}

//Inventory/Makefile
# artifacts
ARCH = arm64
BINARY_NAME=bootstrap
TARGET_DIR = bin

# openapi specification
GENERATED_MODEL=api/generated
MODEL=${GENERATED_MODEL}/api
OPENAPI_SPEC_LOCATION=api/schemas//openapi-spec.yaml
GENERATED_SPEC_LOCATION = ${MODEL}/openapi-spec.json

# pulum
STACK_NAME=dev

clean:
    rm -rf ${TARGET_DIR}
    rm -rf ${GENERATED_MODEL}

build: clean build-oas generate-doc-from-oas

```



```

GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
./bin/http/${BINARY_NAME} ./cmd/http/main.go
GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
./bin/ddbs/${BINARY_NAME} ./cmd/ddbs/main.go
GOOS=linux GOARCH=${ARCH} go build -tags lambda.norpc -o
./bin/msg/booking/requested/${BINARY_NAME}
./cmd/msg/booking/requested/main.go

```

```

build-oas:
  mkdir -p ${MODEL}
  redocly bundle ${OPENAPI_SPEC_LOCATION} -o ${GENERATED_SPEC_LOCATION}
  go run github.com/oapi-codegen/oapi-codegen/v2/cmd/oapi-codegen --
generate types,skip-prune --package api ${GENERATED_SPEC_LOCATION} >
${MODEL}/model-spec.gen.go

```

```

generate-doc-from-oas: build-oas
  redocly build-docs ${GENERATED_SPEC_LOCATION} -o ${MODEL}/spec-
doc.html

```

```

deploy: build
  cd ./infra && pulumi up --stack ${STACK_NAME} -yes

```

```
//Inventory/api/schemas/ApiError.yaml
```

```

type: object
required:
  - code
  - message
properties:
  code:
    type: string
  message:
    type: string

```

```
//Inventory/api/schemas/InventoryItem.yaml
```

```

type: object
required:
  - id
  - type
  - name
  - description
properties:
  id:
    type: string
  type:
    type: string
  name:
    type: string
  description:
    type: string

```

```
//Inventory/api/schemas/InventoryItemAvailability.yaml
```

```

required:
  - slots

```

```

properties:
  slots:
    type: array
    items:
      $ref: 'InventoryItemAvailabilitySlot.yaml'

//Inventory/api/schemas/InventoryItemAvailabilitySlot.yaml
required:
  - slot
  - available
properties:
  slot:
    type: string
  available:
    type: boolean

//Inventory/api/schemas/InventoryItemCreateRequest.yaml
type: object
required:
  - type
  - name
  - description
properties:
  id:
    type: string
  type:
    type: string
  name:
    type: string
  description:
    type: string

//Inventory/api/schemas/InventoryItemsLookupRequest.yaml
required:
  - ids
properties:
  ids:
    type: array
    items:
      type: string

//Inventory/api/schemas/InventoryItemsLookupResponse.yaml
required:
  - data
properties:
  data:
    type: array
    items:
      $ref: "InventoryItem.yaml"

//Inventory/api/schemas/InventoryItemsSearchResponse.yaml
required:
  - data
  - hasMore
properties:

```

```

data:
  type: array
  items:
    $ref: 'InventoryItem.yaml'
nextToken:
  type: string
hasMore:
  type: boolean

//Inventory/api/schemas/openapi-spec.yaml
openapi: 3.0.0
info:
  title: Inventory API
  version: 0.0.1
paths:

  /inventory/items:
    get:
      tags:
        - Inventory Items API
      summary: Get Inventory Items
      parameters:
        - in: query
          name: offsetToken
          required: false
          schema:
            type: string
        - in: query
          name: ascending
          required: false
          schema:
            type: boolean
      responses:
        '200':
          description: "Inventory Items"
          content:
            application/json:
              schema:
                $ref: "InventoryItemsSearchResponse.yaml"
        '400':
          description: Bad request.
          content:
            application/json:
              schema:
                $ref: "ApiError.yaml"
        '500':
          description: Internal server error. The server encountered an
unexpected error.
          content:
            application/json:
              schema:
                $ref: "ApiError.yaml"
      x-amazon-apigateway-integration:
        uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"

```

```

    responses:
      default:
        statusCode: "200"
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        contentHandling: "CONVERT_TO_TEXT"
        type: "aws_proxy"

post:
  tags:
    - Inventory Items API
  summary: Create New Inventory Item
  requestBody:
    content:
      application/json:
        schema:
          $ref: "InventoryItemCreateRequest.yaml"
  responses:
    '200':
      description: "Created Inventory Item"
      content:
        application/json:
          schema:
            $ref: "InventoryItem.yaml"
    '400':
      description: Bad request.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
    '500':
      description: Internal server error. The server encountered an
unexpected error.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
  x-amazon-apigateway-integration:
    uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
    responses:
      default:
        statusCode: "200"
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        contentHandling: "CONVERT_TO_TEXT"
        type: "aws_proxy"

/inventory/items:lookup:
  post:
    tags:
      - Inventory Items API
    summary: Lookup Inventory Items by list of ids
    requestBody:
      content:

```

```

    application/json:
      schema:
        $ref: "InventoryItemsLookupRequest.yaml"
  responses:
    '200':
      description: "Inventory Items"
      content:
        application/json:
          schema:
            $ref: "InventoryItemsLookupResponse.yaml"
    '400':
      description: Bad request.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
    '500':
      description: Internal server error. The server encountered an
unexpected error.
      content:
        application/json:
          schema:
            $ref: "ApiError.yaml"
  x-amazon-apigateway-integration:
    uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
    responses:
      default:
        statusCode: "200"
    passthroughBehavior: "when_no_match"
    httpMethod: "POST"
    contentHandling: "CONVERT_TO_TEXT"
    type: "aws_proxy"

/inventory/items/{type}:
  get:
    tags:
      - Inventory Items API
    summary: Get Inventory Items
    parameters:
      - in: query
        name: offsetToken
        required: false
        schema:
          type: string
      - in: query
        name: ascending
        required: false
        schema:
          type: boolean
      - in: path
        name: type
        schema:
          type: string
        required: true

```

```

        description: Item type
responses:
  '200':
    description: "Inventory Items"
    content:
      application/json:
        schema:
          $ref: "InventoryItemsSearchResponse.yaml"
  '400':
    description: Bad request.
    content:
      application/json:
        schema:
          $ref: "ApiError.yaml"
  '500':
    description: Internal server error. The server encountered an
unexpected error.
    content:
      application/json:
        schema:
          $ref: "ApiError.yaml"
x-amazon-apigateway-integration:
  uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
  responses:
    default:
      statusCode: "200"
  passthroughBehavior: "when_no_match"
  httpMethod: "POST"
  contentHandling: "CONVERT_TO_TEXT"
  type: "aws_proxy"

/inventory/items/availability/{id}:
get:
  tags:
    - Inventory Items API
  summary: Get Item Availability
  parameters:
    - in: query
      name: date
      required: true
      schema:
        type: string
    - in: path
      name: id
      schema:
        type: string
      required: true
      description: Item id
  responses:
    '200':
      description: "Inventory Items"
      content:
        application/json:
          schema:

```

```

        $ref: "InventoryItemAvailability.yaml"
    '400':
        description: Bad request.
        content:
            application/json:
                schema:
                    $ref: "ApiError.yaml"
    '500':
        description: Internal server error. The server encountered an
unexpected error.
        content:
            application/json:
                schema:
                    $ref: "ApiError.yaml"
    x-amazon-apigateway-integration:
        uri: "arn:aws:apigateway:{{.AWS_REGION}}:lambda:path/2015-03-
31/functions/{{.httpHandler}}/invocations"
        responses:
            default:
                statusCode: "200"
                passthroughBehavior: "when_no_match"
                httpMethod: "POST"
                contentHandling: "CONVERT_TO_TEXT"
                type: "aws_proxy"

```

```

//Inventory/cmd/ddbs/main.go
package main

```

```

import (
    "context"
    "fmt"
    "inventory/cmd/ddbs/handler"
    "inventory/internal/properties"

    "github.com/aws/aws-lambda-go/lambda"
    "go.uber.org/fx"
)

func main() {
    if err := getApp().Start(context.Background()); err != nil {
        fmt.Printf("Cannot start app: %s", err.Error())
    }
}

func getApp() *fx.App {
    return fx.New(
        fx.Provide(properties.NewEnvProperties),
        fx.Provide(fx.Annotate(handler.NewDDBSHandler,
fx.ParamTags(`group:"entityChangeMappers"`))),
        fx.Invoke(start),
    )
}

func start(lifecycle fx.Lifecycle, handler *handler.DDBSHandler) {
    lifecycle.Append(fx.Hook{

```

```

        OnStart: func(context.Context) error {
            lambda.Start(handler.HandleStream)
            return nil
        })
    })
}

//Inventory/cmd/ddbs/handler/handler.go
package handler

import (
    "context"
    "encoding/json"
    "fmt"
    "inventory/cmd/ddbs/handler/model"
    "inventory/internal/domain"
)

type EntityChangeMapper interface {
    Name() string
    HandledType() string
    MapChange(ctx context.Context, record model.DynamoEventRecord)
    ([]*domain.Event, error)
}

type DDBSHandler struct {
    mappers map[string][]EntityChangeMapper
}

func NewDDBSHandler(entityChangeMappers []EntityChangeMapper)
*DDBSHandler {
    mappers := make(map[string][]EntityChangeMapper)
    for _, m := range entityChangeMappers {
        handledType := m.HandledType()
        mappers[handledType] = append(mappers[handledType], m)
    }

    return &DDBSHandler{mappers: mappers}
}

func (h *DDBSHandler) HandleStream(_ context.Context, dynamoEvent
model.DynamoEvent) error {
    dynamoDbEventJson, err := json.Marshal(dynamoEvent)
    if err != nil {
        return err
    }

    fmt.Printf("Received event: %s \n", dynamoDbEventJson)
    return nil
}

//Inventory/cmd/ddbs/handler/model/model.go
package model

import (
    "github.com/aws/aws-sdk-go/service/dynamodb"

```



```

)

type DynamoEvent struct {
    Records []DynamoEventRecord `json:"Records"`
}

type DynamoEventRecord struct {
    EventID    string          `json:"eventID"`
    EventName  string          `json:"eventName"`
    Change     DynamoEventChange `json:"dynamodb"`
}

type DynamoEventChange struct {
    NewImage map[string]*dynamodb.AttributeValue
    `json:"NewImage,omitempty"`
    OldImage map[string]*dynamodb.AttributeValue
    `json:"OldImage,omitempty"`
}

//Inventory/cmd/http/main.go
package handler

import (
    "context"
    "encoding/json"
    "fmt"
    "inventory/cmd/ddbs/handler/model"
    "inventory/internal/domain"
)

type EntityChangeMapper interface {
    Name() string
    HandledType() string
    MapChange(ctx context.Context, record model.DynamoEventRecord)
    ([]*domain.Event, error)
}

type DDBSHandler struct {
    mappers map[string][]EntityChangeMapper
}

func NewDDBSHandler(entityChangeMappers []EntityChangeMapper)
*DDBSHandler {
    mappers := make(map[string][]EntityChangeMapper)
    for _, m := range entityChangeMappers {
        handledType := m.HandledType()
        mappers[handledType] = append(mappers[handledType], m)
    }

    return &DDBSHandler{mappers: mappers}
}

func (h *DDBSHandler) HandleStream(_ context.Context, dynamoEvent
model.DynamoEvent) error {
    dynamoDbEventJson, err := json.Marshal(dynamoEvent)

```

```

    if err != nil {
        return err
    }

    fmt.Printf("Received event: %s \n", dynamoDbEventJson)
    return nil
}

//Inventory/cmd/http/handler/handler.go
package handler

import (
    "context"
    "fmt"
    "inventory/cmd/http/mapper"
    "inventory/internal/domain"

    "github.com/aws/aws-lambda-go/events"
)

type Request = events.APIGatewayProxyRequest
type Response = events.APIGatewayProxyResponse

const (
    CodeInvalidRequest = "INVALID_REQUEST"
    CodeInternalError  = "INTERNAL_ERROR"
)

type HttpHandler struct {
    apiMapper      mapper.HttpMapper
    itemsService   domain.InventoryItemService
    routes         map[string]func(ctx context.Context, request Request)
                  (Response, error)
}

func NewHttpHandler(apiMapper mapper.HttpMapper, itemsService
domain.InventoryItemService) *HttpHandler {
    handler := &HttpHandler{
        apiMapper:      apiMapper,
        itemsService:   itemsService,
    }

    handler.routes = map[string]func(ctx context.Context, request Request)
(Response, error){
        "POST_/inventory/items":          handler.CreateItem,
        "POST_/inventory/items:lookup":    handler.LookupItems,

        "GET_/inventory/items":             handler.ListItems,
        "GET_/inventory/items/{type}":      handler.ListItemsByType,
        "GET_/inventory/items/availability/{id}": handler.ItemAvailability,
    }

    return handler
}

```

```

func (h *HttpHandler) Handle(ctx context.Context, request Request)
(Response, error) {
    route := h.getRout(request)
    f, ok := h.routs[route]
    if !ok {
        return events.APIGatewayProxyResponse{StatusCode: 404, Body: "Rout
not found"}, nil
    }
    return f(ctx, request)
}

func (h *HttpHandler) getRout(request events.APIGatewayProxyRequest)
string {
    return fmt.Sprintf("%s_%s", request.HTTPMethod, request.Resource)
}

func (h *HttpHandler) CreateItem(ctx context.Context, request Request)
(Response, error) {
    newItem, err := h.apiMapper.NewItemFromRequest(request)
    if err != nil {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
nil
    }

    if err = h.itemsService.CreateItem(ctx, newItem); err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalError, err.Error()),
nil
    }

    return h.apiMapper.ToApiResponse(200,
h.apiMapper.ToApiInventoryItem(newItem)), nil
}

func (h *HttpHandler) LookupItems(ctx context.Context, request Request)
(Response, error) {
    ids, err := h.apiMapper.IdsFromLookupRequest(request)
    if err != nil {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
nil
    }

    items, err := h.itemsService.GetItemsByIds(ctx, ids...)
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalError, err.Error()),
nil
    }

    return h.apiMapper.ToApiResponse(200,
h.apiMapper.ToApiLookupResponse(items)), nil
}

func (h *HttpHandler) ListItems(ctx context.Context, request Request)
(Response, error) {
    sliceRequest, err := h.apiMapper.ExtractSliceRequest(request)
    if err != nil {

```

```

        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
        nil
    }

    itemsSlice, err := h.itemsService.ItemsSlice(ctx, sliceRequest)
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalError, err.Error()),
        nil
    }

    return h.apiMapper.ToApiResponse(200,
h.apiMapper.ToApiSearchResponse(itemsSlice)), nil
}

func (h *HttpHandler) ListItemsByType(ctx context.Context, request
Request) (Response, error) {
    sliceRequest, err := h.apiMapper.ExtractSliceRequest(request)
    if err != nil {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
        nil
    }

    itemsSlice, err := h.itemsService.ItemsByTypeSlice(ctx,
request.PathParameters["type"], sliceRequest)
    if err != nil {
        return h.apiMapper.ToApiError(500, CodeInternalError, err.Error()),
        nil
    }

    return h.apiMapper.ToApiResponse(200,
h.apiMapper.ToApiSearchResponse(itemsSlice)), nil
}

func (h *HttpHandler) ItemAvailability(ctx context.Context, request
Request) (Response, error) {
    var (
        id    = request.PathParameters["id"]
        date = request.QueryStringParameters["date"]
    )

    slots, err := h.itemsService.GetItemAvailability(ctx, id, date)
    if err != nil {
        return h.apiMapper.ToApiError(400, CodeInvalidRequest, err.Error()),
        nil
    }

    return h.apiMapper.ToApiResponse(200,
h.apiMapper.ToApiAvailabilityResponse(slots)), nil
}

//Inventory/cmd/http/mapper/http_mapper.go
package mapper

import (
    "encoding/json"

```

```

    "inventory/api/generated/api"
    "inventory/internal/domain"
    "inventory/pkg"
    "strconv"

    "github.com/aws/aws-lambda-go/events"
    "github.com/google/uuid"
)

type Request = events.APIGatewayProxyRequest
type Response = events.APIGatewayProxyResponse

const (
    CodeInternalError = "INTERNAL_ERROR"
    defaultSliceSize = 10
)

type HttpMapper interface {
    NewItemFromRequest(Request) (*domain.InventoryItem, error)
    IdsFromLookupRequest(Request) ([]string, error)

    ToApiInventoryItem(*domain.InventoryItem) *api.InventoryItem
    ToApiSearchResponse(*pkg.Slice[*domain.InventoryItem])
    *api.InventoryItemsSearchResponse
    ToApiAvailabilityResponse([]*domain.InventoryItemSlot)
    *api.InventoryItemAvailability
    ToApiLookupResponse([]*domain.InventoryItem)
    *api.InventoryItemsLookupResponse

    ExtractSliceRequest(Request) (*pkg.SliceRequest, error)

    ToApiError(code int, messageCode, messageText string) Response
    ToApiResponse(code int, body any) Response
}

type httpMapper struct {
}

func NewHttpMapper() HttpMapper {
    return &httpMapper{}
}

func (h *httpMapper) NewItemFromRequest(request
events.APIGatewayProxyRequest) (*domain.InventoryItem, error) {
    itemCreationRequest := &api.InventoryItemCreateRequest{}
    err := json.Unmarshal([]byte(request.Body), itemCreationRequest)
    if err != nil {
        return nil, err
    }

    itemId, err := uuid.NewV7()
    if err != nil {
        return nil, err
    }
}

```

```

return &domain.InventoryItem{
    Id:          itemId.String(),
    Name:        itemCreationRequest.Name,
    Description: itemCreationRequest.Description,
    Type:        itemCreationRequest.Type,
}, nil
}

func (h *httpMapper) IdsFromLookupRequest(request Request) ([]string,
error) {
    lookupRequest := &api.InventoryItemsLookupRequest{}
    err := json.Unmarshal([]byte(request.Body), lookupRequest)
    if err != nil {
        return nil, err
    }

    var ids []string
    for _, id := range lookupRequest.Ids {
        ids = append(ids, id)
    }

    return ids, nil
}

func (h *httpMapper) ToApiInventoryItem(item *domain.InventoryItem)
*api.InventoryItem {
    return &api.InventoryItem{
        Id:          item.Id,
        Name:        item.Name,
        Description: item.Description,
        Type:        item.Type,
    }
}

func (h *httpMapper) ToApiSearchResponse(slice
*pkg.Slice[*domain.InventoryItem]) *api.InventoryItemsSearchResponse {
    var apiInventoryItems []api.InventoryItem
    for _, item := range slice.Data {
        apiInventoryItems = append(apiInventoryItems,
*h.ToApiInventoryItem(item))
    }

    return &api.InventoryItemsSearchResponse{
        Data:        apiInventoryItems,
        HasMore:     slice.HasMore,
        NextToken:   slice.NextToken,
    }
}

func (h *httpMapper) ToApiAvailabilityResponse(slots
[*domain.InventoryItemSlot]) *api.InventoryItemAvailability {
    var apiSlots []api.InventoryItemAvailabilitySlot
    for _, slot := range slots {
        apiSlots = append(apiSlots, api.InventoryItemAvailabilitySlot{

```

```

        Slot:      slot.SlotStartTime,
        Available: slot.BookedBy == nil,
    })
}

return &api.InventoryItemAvailability{
    Slots: apiSlots,
}
}

func (h *httpMapper) ToApiLookupResponse(items []*domain.InventoryItem)
*api.InventoryItemsLookupResponse {
    var data []api.InventoryItem
    for _, item := range items {
        data = append(data, *h.ToApiInventoryItem(item))
    }

    return &api.InventoryItemsLookupResponse{Data: data}
}

func (h *httpMapper) ExtractSliceRequest(request
events.APIGatewayProxyRequest) (*pkg.SliceRequest, error) {
    var (
        offsetToken *string
        ascending    *bool
    )

    if token, ok := request.QueryStringParameters["offsetToken"]; ok {
        offsetToken = &token
    }

    if ascendingValue, ok := request.QueryStringParameters["ascending"];
ok {
        ascendingValueParsed, err := strconv.ParseBool(ascendingValue)
        if err != nil {
            return nil, err
        }
        ascending = &ascendingValueParsed
    }

    return &pkg.SliceRequest{Limit: defaultSliceSize, Token: offsetToken,
Ascending: ascending}, nil
}

func (h *httpMapper) ToApiResponse(code int, body any)
events.APIGatewayProxyResponse {
    bodyJson, err := json.Marshal(body)
    if err != nil {
        return h.ToApiError(500, CodeInternalServerError, err.Error())
    }

    return events.APIGatewayProxyResponse{StatusCode: code, Body:
string(bodyJson)}
}

```

```

func (h *httpMapper) ToApiError(code int, messageCode, messageText
string) events.APIGatewayProxyResponse {
    response := &api.ApiError{Code: messageCode, Message: messageText}
    jsonResponse, _ := json.Marshal(response)
    return events.APIGatewayProxyResponse{StatusCode: code, Body:
string(jsonResponse)}
}

//Inventory/cmd/msg/booking/requested/main.go
package main

import (
    "context"
    "fmt"
    "inventory/cmd/msg/booking/requested/handler"
    "inventory/internal/clients"
    "inventory/internal/properties"
    "inventory/internal/repository"
    "inventory/internal/repository/mapper"
    "inventory/internal/service"
    "inventory/pkg"

    "github.com/aws/aws-lambda-go/lambda"
    "go.uber.org/fx"
)

func main() {
    if err := getApp().Start(context.Background()); err != nil {
        fmt.Printf("Cannot start app: %s", err.Error())
    }
}

func getApp() *fx.App {
    return fx.New(

        // Common
        fx.Provide(pkg.NewClock),
        fx.Provide(properties.NewEnvProperties),

        // Dynamo DB
        fx.Provide(mapper.NewDbInventoryMapper),
        fx.Provide(clients.NewDynamoDbClient),
        fx.Provide(repository.NewInventoryItemRepository),
        fx.Provide(service.NewInventoryItemService),

        //SNS
        fx.Provide(clients.NewSnsClient),
        fx.Provide(service.NewEventsPublisher),

        // Handler
        fx.Provide(handler.NewBookingRequestedHandler),
        fx.Invoke(start),
    )
}

```



```

func start(lifecycle fx.Lifecycle, handler
*handler.BookingRequestedHandler) {
    lifecycle.Append(fx.Hook{
        OnStart: func(context.Context) error {
            lambda.Start(handler.Handle)
            return nil
        })
    })
}

//Inventory/cmd/msg/booking/requested/handler/handler.go
package handler

import (
    "context"
    "encoding/json"
    "fmt"
    "inventory/internal/domain"
    "inventory/internal/properties"

    "github.com/aws/aws-lambda-go/events"
    "github.com/samber/lo"
    "golang.org/x/sync/errgroup"
)

type BookingRequestedHandler struct {
    props          *properties.Properties
    publisher       domain.EventsPublisher
    inventoryService domain.InventoryItemService
}

type BookingRequestProcessingResult struct {
    Accepted *domain.BookingAccepted `json:"accepted,omitempty"`
    Rejected *domain.BookingRejected `json:"rejected,omitempty"`
}

func NewBookingRequestedHandler(
    props *properties.Properties,
    publisher domain.EventsPublisher,
    inventoryService domain.InventoryItemService,
) *BookingRequestedHandler {
    return &BookingRequestedHandler{
        props:          props,
        publisher:       publisher,
        inventoryService: inventoryService,
    }
}

func (b *BookingRequestedHandler) Handle(ctx context.Context, event
events.SQSEvent) error {
    bookingRequests, err := b.extractEvents(event)
    if err != nil {
        return err
    }
    bookingRequestsJsonBytes, _ := json.Marshal(bookingRequests)

```

```

    fmt.Printf("Booking requests received: %s \n",
bookingRequestsJsonBytes)

    processingResults, err := b.processBookingRequests(ctx,
bookingRequests)
    if err != nil {
        return err
    }
    resultsJsonBytes, _ := json.Marshal(processingResults)
    fmt.Printf("Booking requests processed: %s \n",
string(resultsJsonBytes))

    eventsToPublish, err := b.toEvents(processingResults)
    return b.publisher.Publish(ctx, eventsToPublish)
}

func (b *BookingRequestedHandler) processBookingRequests(ctx
context.Context, requests []*domain.BookingRequested)
([]*BookingRequestProcessingResult, error) {
    var (
        eg      errgroup.Group
        results = make([]*BookingRequestProcessingResult, len(requests))
    )

    for i, bookingRequest := range requests {
        eg.Go(func() error {
            result, err := b.processBookingRequest(ctx, bookingRequest)
            if err != nil {
                return err
            }

            results[i] = result
            return nil
        })
    }

    if err := eg.Wait(); err != nil {
        return nil, err
    }

    return results, nil
}

func (b *BookingRequestedHandler) processBookingRequest(ctx
context.Context, request *domain.BookingRequested)
(*BookingRequestProcessingResult, error) {
    bookedSlots, err := b.inventoryService.BookItem(ctx,
&domain.BookingItemParams{
        ItemId:      request.InventoryId,
        BookingId: request.BookingId,
        StartTime: request.RequestedFrom,
        EndTime:     request.RequestedTo,
    })

    if err != nil {

```

```

    if bErr := domain.AsBussinessError(err); bErr != nil {
        return &BookingRequestProcessingResult{
            Rejected: &domain.BookingRejected{
                UserId:      request.UserId,
                BookingId:    request.BookingId,
                InventoryId:  request.InventoryId,
                RejectionReason: err.Error(),
            },
        }, nil
    }
    return nil, err
}

return &BookingRequestProcessingResult{
    Accepted: &domain.BookingAccepted{
        UserId:      request.UserId,
        BookingId:    request.BookingId,
        InventoryId:  request.InventoryId,
        BookingStartTime: b.minSlotStartTime(bookedSlots),
        BookingEndTime:  b.maxSlotEndTime(bookedSlots),
    },
}, nil
}

func (b *BookingRequestedHandler) toEvents(results
[]*BookingRequestProcessingResult) ([]*domain.Event, error) {
    var resultEvents []*domain.Event
    for _, result := range results {
        if accepted := result.Accepted; accepted != nil {
            messageJson, err := json.Marshal(accepted)
            if err != nil {
                return nil, err
            }

            resultEvents = append(resultEvents, &domain.Event{
                Id:      fmt.Sprintf("accepted-%s", accepted.BookingId),
                GroupId: accepted.InventoryId,
                Topic:    b.props.BookingAcceptedTopic,
                Content: messageJson,
            })
        }

        if rejected := result.Rejected; rejected != nil {
            messageJson, err := json.Marshal(rejected)
            if err != nil {
                return nil, err
            }

            resultEvents = append(resultEvents, &domain.Event{
                Id:      fmt.Sprintf("rejected-%s", rejected.BookingId),
                GroupId: rejected.InventoryId,
                Topic:    b.props.BookingRejectedTopic,
                Content: messageJson,
            })
        }
    }
}

```



```

const (
    defaultLambdaTimeout = 20
    defaultLambdaMemory  = 512

    lambdaAssumePolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["sts:AssumeRole"], "Principal": { "Service": "lambda.amazonaws.com" } } ] }`
    snsPublishPolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["sns:*"], "Resource": "%s" } ] }`

    snsToSqsPublishPolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["sqs:SendMessage"], "Principal": { "Service": "sns.amazonaws.com" }, "Resource": "%s", "Condition": { "ArnEquals": { "aws:SourceArn": "%s" } } } ] }`
    sqsToLambdaPublishPolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["sqs:ReceiveMessage", "sqs:DeleteMessage", "sqs:GetQueueAttributes", "sqs:ChangeMessageVisibility"], "Resource": "%s" } ] }`
    dynamoPolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["dynamodb:*"], "Resource": ["%s", "%s/index/*"] } ] }`
    dynamoStreamPolicy = `{ "Version": "2012-10-
17", "Statement": [{ "Effect": "Allow", "Action": ["dynamodb:*"], "Resource": "%s" } ] }`
)

var (
    lambdaRole      *iam.Role
    dynamodbTable  *dynamodb.Table
    envVariables    = pulumi.StringMap{}
    infraStackRef   *pulumi.StackReference
)

func main() {
    pulumi.Run(func(ctx *pulumi.Context) error {

        steps := []func(*pulumi.Context) error{
            initTable,
            initInfraStack,
            initLambdaRole,
            initHttpApi,
            initDDBStream,
            initMSG,
        }

        for _, fn := range steps {
            if err := fn(ctx); err != nil {
                return err
            }
        }
        return nil
    })
}

```

```

func initInfraStack(ctx *pulumi.Context) error {
    ref, err := pulumi.NewStackReference(ctx,
fmt.Sprintf("organization/infrastructure/%s", ctx.Stack()), nil)
    if err != nil {
        return err
    }

    infraStackRef = ref
    return nil
}

func initTable(ctx *pulumi.Context) error {
    resourceName := resourceName(ctx, "table")
    table, err := dynamodb.NewTable(ctx, resourceName,
&dynamodb.TableArgs{
    Name:          pulumi.String(resourceName),
    StreamEnabled: pulumi.BoolPtr(true),
    BillingMode:    pulumi.String("PAY_PER_REQUEST"),
    StreamViewType: pulumi.StringPtr("NEW_AND_OLD_IMAGES"),
    HashKey:       pulumi.String(model.HashKey),
    RangeKey:      pulumi.String(model.RangeKey),
    Attributes:    dynamodb.TableAttributeArray{

        // Main table key
        &dynamodb.TableAttributeArgs{Name: pulumi.String(model.HashKey),
Type: pulumi.String(model.HashKeyType)},
        &dynamodb.TableAttributeArgs{Name: pulumi.String(model.RangeKey),
Type: pulumi.String(model.RangeKeyType)},

        // GSI1 key
        &dynamodb.TableAttributeArgs{Name:
pulumi.String(model.GSI1HashKey), Type:
pulumi.String(model.GSI1HashKeyType)},
        &dynamodb.TableAttributeArgs{Name:
pulumi.String(model.GSI1RangeKey), Type:
pulumi.String(model.GSI1RangeKeyType)},

        // GSI2 key
        &dynamodb.TableAttributeArgs{Name:
pulumi.String(model.GSI2HashKey), Type:
pulumi.String(model.GSI2HashKeyType)},
        &dynamodb.TableAttributeArgs{Name:
pulumi.String(model.GSI2RangeKey), Type:
pulumi.String(model.GSI2RangeKeyType)},
    },
    GlobalSecondaryIndexes: dynamodb.TableGlobalSecondaryIndexArray{

        dynamodb.TableGlobalSecondaryIndexArgs{
            Name:          pulumi.String(model.GSI1),
            HashKey:       pulumi.String(model.GSI1HashKey),
            RangeKey:      pulumi.String(model.GSI1RangeKey),
            ProjectionType: pulumi.String("ALL"),
        },

        dynamodb.TableGlobalSecondaryIndexArgs{

```

```

        Name:          pulumi.String(model.GSI2),
        HashKey:       pulumi.String(model.GSI2HashKey),
        RangeKey:      pulumi.String(model.GSI2RangeKey),
        ProjectionType: pulumi.String("ALL"),
    },
},
}, pulumi.RetainOnDelete(false))

if err != nil {
    return err
}

dynamodbTable = table
envVariables[properties.TableNameKey] = pulumi.String(resourceName)
return nil
}

func initLambdaRole(ctx *pulumi.Context) error {
    // Base Role
    roleName := resourceName(ctx, "lambda-role")
    role, err := iam.NewRole(ctx, roleName,
&iam.RoleArgs{AssumeRolePolicy: pulumi.String(lambdaAssumePolicy)})
    if err != nil {
        return err
    }

    rolePolicyAttachmentName := resourceName(ctx, "lambda-basic-execution-
role")
    _, err = iam.NewRolePolicyAttachment(ctx, rolePolicyAttachmentName,
&iam.RolePolicyAttachmentArgs{
        Role:      role.Name,
        PolicyArn: pulumi.String("arn:aws:iam::aws:policy/service-
role/AWSLambdaBasicExecutionRole"),
    })
    if err != nil {
        return err
    }

    // DynamoDB Access
    ddbPolicyName := resourceName(ctx, "ddb-access-policy")
    ddbPolicy := pulumi.Sprintf(dynamoPolicy, dynamodbTable.Arn,
dynamodbTable.Arn)
    _, err = iam.NewRolePolicy(ctx, ddbPolicyName,
&iam.RolePolicyArgs{Role: role.ID(), Policy: ddbPolicy})
    if err != nil {
        return err
    }

    // DynamoDB Stream Access
    ddbStreamPolicyName := resourceName(ctx, "ddb-stream-policy")
    ddbStreamPolicy := pulumi.Sprintf(dynamoStreamPolicy,
dynamodbTable.StreamArn)
    _, err = iam.NewRolePolicy(ctx, ddbStreamPolicyName,
&iam.RolePolicyArgs{Role: role.ID(), Policy: ddbStreamPolicy})
    if err != nil {

```

```

        return err
    }

    // SNS Access
    err = writeToTopicPolicy(ctx, "booking-accepted",
properties.BookingAcceptedTopicKey, role)
    if err != nil {
        return err
    }

    err = writeToTopicPolicy(ctx, "booking-rejected",
properties.BookingRejectedTopicKey, role)
    if err != nil {
        return err
    }

    lambdaRole = role
    return nil
}

func writeToTopicPolicy(ctx *pulumi.Context, topicName, env string, role
*iam.Role) error {
    policyName := fmt.Sprintf("%s-topic-write-policy", topicName)
    topicArn :=
infraStackRef.GetOutput(pulumi.String(fmt.Sprintf("%s.topicArn",
topicName))).AsStringOutput()
    policy := pulumi.Sprintf(snsPublishPolicy, topicArn)
    _, err := iam.NewRolePolicy(ctx, policyName, &iam.RolePolicyArgs{Role:
role.ID(), Policy: policy})
    if err != nil {
        return err
    }

    envVariables[env] = topicArn
    return nil
}

func initHttpApi(ctx *pulumi.Context) error {
    tplBytes, err := os.ReadFile("../api/generated/api/openapi-spec.json")
    if err != nil {
        return err
    }
    tpl, err :=
template.New("openapi").Option("missingkey=error").Parse(string(tplBytes
))
    if err != nil {
        panic(err)
    }

    region, err := aws.GetRegion(ctx, nil, nil)
    if err != nil {
        return err
    }

```



```

    httpFunction, err := newLambda(ctx, "http-function",
    "../bin/http/bootstrap")
    if err != nil {
        return err
    }

    renderedSpec := httpFunction.Arn.ApplyT(func(lambdaArn string)
(string, error) {
        var buf bytes.Buffer
        err := tpl.Execute(&buf, map[string]any{
            "AWS_REGION": region.Region,
            "httpHandler": lambdaArn,
        })
        if err != nil {
            return "", err
        }
        return buf.String(), nil
    }).(pulumi.StringOutput)

    apiName := resourceName(ctx, "api")
    api, err := apigateway.NewRestApi(ctx, apiName,
    &apigateway.RestApiArgs{
        Name: pulumi.String(apiName),
        Body: renderedSpec,
    })
    if err != nil {
        return err
    }

    specHash := renderedSpec.ApplyT(func(spec string) string {
        sum := sha1.Sum([]byte(spec))
        return hex.EncodeToString(sum[:])
    }).(pulumi.StringOutput)

    deploymentName := resourceName(ctx, "api-deployment")
    deploy, err := apigateway.NewDeployment(ctx, deploymentName,
    &apigateway.DeploymentArgs{
        RestApi: api.ID(),
        Triggers: pulumi.StringMap{"openapiSpecHash": specHash},
    }, pulumi.DependsOn([]pulumi.Resource{api}))
    if err != nil {
        return err
    }

    stageName := "api"
    _, err = apigateway.NewStage(ctx, stageName, &apigateway.StageArgs{
        RestApi: api.ID(),
        Deployment: deploy.ID(),
        StageName: pulumi.String(stageName),
    })
    if err != nil {
        return err
    }

```

```

    apiUrl := pulumi.Sprintf("https://%s.execute-api.%s.amazonaws.com/%s",
api.ID(), region.Region, stageName)
    paramName := fmt.Sprintf("/services/%s/%s/url", ctx.Project(),
ctx.Stack())
    _, err = ssm.NewParameter(ctx, resourceName(ctx, "apiUrlParam"),
&ssm.ParameterArgs{
        Name:  pulumi.String(paramName),
        Type:  pulumi.String("String"),
        Value: apiUrl,
    })
    if err != nil {
        return err
    }

    invokePermissionName := resourceName(ctx, "allow-api-gateway-invoke")
    _, err = lambda.NewPermission(ctx, invokePermissionName,
&lambda.PermissionArgs{
        Action:    pulumi.String("lambda:InvokeFunction"),
        Function:   httpFunction.Name,
        Principal: pulumi.String("apigateway.amazonaws.com"),
        SourceArn: pulumi.Sprintf("%s/*/*", api.ExecutionArn),
    })
    if err != nil {
        return err
    }

    return nil
}

func initDDBStream(ctx *pulumi.Context) error {
    ddbStreamFunction, err := newLambda(ctx, "ddb-stream-handler",
"../bin/ddbs/bootstrap")
    if err != nil {
        return err
    }

    ddbStreamMappingName := resourceName(ctx, "ddb-stream-mapping")
    _, err = lambda.NewEventSourceMapping(ctx, ddbStreamMappingName,
&lambda.EventSourceMappingArgs{
        EventSourceArn: dynamodbTable.StreamArn,
        FunctionName:   ddbStreamFunction.Arn,

        StartingPosition: pulumi.String("TRIM_HORIZON"),
        BatchSize:        pulumi.Int(100),
    })
    if err != nil {
        return err
    }

    return nil
}

func initMSG(ctx *pulumi.Context) error {
    return subscribeTopic(ctx, &subscribeTopicArgs{
        featureName: "booking-requested",
    })
}

```

```

        lambdaName: "booking-requested-handler",
        lambdaPath: "../bin/msg/booking/requested/bootstrap",
        queueName: "booking-requested-queue.fifo",
        topicArn: infraStackRef.GetOutput(pulumi.String("booking-
requested.topicArn")).AsStringOutput(),
    })
}

```

```

type subscribeTopicArgs struct {
    featureName string
    topicArn     pulumi.StringOutput
    queueName    string
    lambdaName   string
    lambdaPath   string
}

```

```

func subscribeTopic(ctx *pulumi.Context, args *subscribeTopicArgs) error {
    lambdaHandler, err := newLambda(ctx, args.lambdaName, args.lambdaPath)
    if err != nil {
        return err
    }
}

```

```

queueResourceName := resourceName(ctx, args.queueName)
queue, err := sqs.NewQueue(ctx, queueResourceName, &sqs.QueueArgs{
    Name:                pulumi.String(queueResourceName),
    FifoQueue:           pulumi.Bool(true),
    VisibilityTimeoutSeconds: pulumi.Int(30),
})
if err != nil {
    return err
}

```

```

snsToSqsPolicyName := resourceName(ctx, fmt.Sprintf("sns-to-sqs-
policy-%s", args.featureName))
_, err = sqs.NewQueuePolicy(ctx, snsToSqsPolicyName,
&sqs.QueuePolicyArgs{
    QueueUrl: queue.ID(),
    Policy:   pulumi.Sprintf(snsToSqsPublishPolicy, queue.Arn,
args.topicArn),
})
if err != nil {
    return err
}

```

```

sqsToLambdaPolicyName := resourceName(ctx, fmt.Sprintf("sqs-to-lambda-
policy-%s", args.featureName))
_, err = iam.NewRolePolicy(ctx, sqsToLambdaPolicyName,
&iam.RolePolicyArgs{
    Role:   lambdaRole.Name,
    Policy: pulumi.Sprintf(sqsToLambdaPublishPolicy, queue.Arn),
})
if err != nil {
    return err
}

```

```

    subscriptionName := resourceName(ctx, fmt.Sprintf("%s-subscription",
args.featureName))
_, err = sns.NewTopicSubscription(ctx, subscriptionName,
&sns.TopicSubscriptionArgs{
    Topic:      args.topicArn,
    Protocol:   pulumi.String("sqs"),
    Endpoint:   queue.Arn,

    RawMessageDelivery: pulumi.Bool(true),
})
if err != nil {
    return err
}

mappingName := resourceName(ctx, fmt.Sprintf("%s-mapping",
args.featureName))
_, err = lambda.NewEventSourceMapping(ctx, mappingName,
&lambda.EventSourceMappingArgs{
    EventSourceArn: queue.Arn,
    FunctionName:   lambdaHandler.Arn,
    BatchSize:      pulumi.Int(10),
})

return nil
}

func newLambda(ctx *pulumi.Context, name, path string)
(*lambda.Function, error) {
    functionName := resourceName(ctx, name)
    return lambda.NewFunction(ctx, functionName, &lambda.FunctionArgs{
        Role:          lambdaRole.Arn,
        Handler:       pulumi.String("bootstrap"),
        Name:          pulumi.String(functionName),
        Timeout:       pulumi.Int(defaultLambdaTimeout),
        MemorySize:    pulumi.Int(defaultLambdaMemory),
        Runtime:       pulumi.String(lambda.RuntimeCustomAL2023),
        Architectures: pulumi.StringArray{pulumi.String("arm64")},
        Environment:   &lambda.FunctionEnvironmentArgs{Variables:
envVariables},
        LoggingConfig: lambda.FunctionLoggingConfigArgs{LogFormat:
pulumi.String("JSON")},
        Code:          pulumi.NewAssetArchive(map[string]any{"bootstrap":
pulumi.NewFileAsset(path)}),
    })
}

func resourceName(ctx *pulumi.Context, name string) string {
    return fmt.Sprintf("%s-%s-%s", ctx.Stack(), ctx.Project(), name)
}

//Inventory/infra/Pulumi.dev.yaml
encryptionSalt: <key>
config:
    aws:region: eu-central-1

```

```

// Inventory/infra/Pulumi.yaml
name: inventory
description: A minimal AWS Go Pulumi program
runtime: go
config:
  pulumi:tags:
    value:
      pulumi:template: aws-go

//Inventory/internal/domain/model.go
package domain

type InventoryItem struct {
    Id          string
    Type        string
    Name        string
    Description string
}

type InventoryItemSlot struct {
    InventoryItemId string
    SlotStartTime   string
    SlotEndTime     string

    BookedBy *string
}

type BookingItemParams struct {
    ItemId      string
    BookingId   string

    StartTime string
    EndTime   string

    Slots []*InventoryItemSlot
}

type CancelBookingParams struct {
    ItemId      string
    BookingId   string
}

//Inventory/internal/domain/repository.go
package domain

import (
    "context"
    "inventory/pkg"
)

type InventoryItemRepository interface {
    CreateItem(ctx context.Context, item *InventoryItem) error
    GetItem(ctx context.Context, id string) (*InventoryItem, error)
    GetByIds(ctx context.Context, ids ...string) ([]*InventoryItem, error)
}

```

```

    GetBookedSlotItems(ctx context.Context, itemId string, date string)
    ([]*InventoryItemSlot, error)

    ItemsSlice(ctx context.Context, sliceRequest *pkg.SliceRequest)
    (*pkg.Slice[*InventoryItem], error)
    ItemsByTypeSlice(ctx context.Context, itemType string, sliceRequest
    *pkg.SliceRequest) (*pkg.Slice[*InventoryItem], error)

    BookItem(ctx context.Context, params *BookingItemParams)
    ([]*InventoryItemSlot, error)
    CancelBooking(ctx context.Context, params *CancelBookingParams) error
}

//Inventory/internal/domain/service.go
package domain

import (
    "context"
    "inventory/pkg"
)

type InventoryItemService interface {
    CreateItem(ctx context.Context, item *InventoryItem) error
    GetItem(ctx context.Context, itemId string) (*InventoryItem, error)
    GetItemsByIds(ctx context.Context, ids ...string) ([]*InventoryItem,
    error)

    GetItemAvailability(ctx context.Context, itemId string, date string)
    ([]*InventoryItemSlot, error)

    BookItem(ctx context.Context, params *BookingItemParams)
    ([]*InventoryItemSlot, error)
    CancelBooking(ctx context.Context, params *CancelBookingParams) error

    ItemsSlice(ctx context.Context, sliceRequest *pkg.SliceRequest)
    (*pkg.Slice[*InventoryItem], error)
    ItemsByTypeSlice(ctx context.Context, itemType string, sliceRequest
    *pkg.SliceRequest) (*pkg.Slice[*InventoryItem], error)
}

type EventsPublisher interface {
    Publish(ctx context.Context, events []*Event) error
}

//Inventory/internal/properties/properties.go
package properties

import "os"

const (
    TableNameKey = "TABLE_NAME"

    BookingAcceptedTopicKey = "BOOKING_ACCEPTED_TOPIC"
    BookingRejectedTopicKey = "BOOKING_REJECTED_TOPIC"

```

```

    BookingCancellationAcceptedTopicKey =
"BOOKING_CANCELLATION_ACCEPTED_TOPIC"
    BookingCancellationRejectedTopicKey =
"BOOKING_CANCELLATION_REJECTED_TOPIC"
)

type Properties struct {
    TableName string

    BookingAcceptedTopic string
    BookingRejectedTopic string

    BookingCancellationAcceptedTopic string
    BookingCancellationRejectedTopic string
}

func NewEnvProperties() *Properties {
    return &Properties{
        TableName: os.Getenv(TableNameKey),

        BookingAcceptedTopic: os.Getenv(BookingAcceptedTopicKey),
        BookingRejectedTopic: os.Getenv(BookingRejectedTopicKey),

        BookingCancellationAcceptedTopic:
os.Getenv(BookingCancellationAcceptedTopicKey),
        BookingCancellationRejectedTopic:
os.Getenv(BookingCancellationRejectedTopicKey),
    }
}

//Inventory/internal/repository/inventory_repository.go
package repository

import (
    "context"
    "encoding/base64"
    "errors"
    "fmt"
    "inventory/internal/domain"
    "inventory/internal/properties"
    "inventory/internal/repository/mapper"
    "inventory/internal/repository/model"
    "inventory/pkg"

    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
    "github.com/samber/lo"

    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const (
    conditionalCheckFailedCode = "ConditionalCheckFailed"

```

```

)

type inventoryItemRepository struct {
    dbMapper    mapper.DbInventoryMapper
    dbClient    *dynamodb.Client
    properties  *properties.Properties
}

func NewInventoryItemRepository(
    dbMapper mapper.DbInventoryMapper,
    dbClient *dynamodb.Client,
    properties *properties.Properties,
) domain.InventoryItemRepository {
    return &inventoryItemRepository{
        dbMapper:    dbMapper,
        dbClient:    dbClient,
        properties:  properties,
    }
}

func (i *inventoryItemRepository) CreateItem(ctx context.Context, item
*domain.InventoryItem) error {
    dbItem := i.dbMapper.ToDbInventoryItem(item)
    avItem, err := dbItem.AsAttributeValues()
    if err != nil {
        return err
    }

    _, err = i.dbClient.PutItem(ctx, &dynamodb.PutItemInput{
        Item:                avItem,
        TableName:           aws.String(i.properties.TableName),
        ConditionExpression:  aws.String("attribute_not_exists(#PK) AND
attribute_not_exists(#SK)"),
        ExpressionAttributeNames: map[string]string{"#PK": model.HashKey,
"#SK": model.RangeKey},
    })
    if err != nil {
        return err
    }

    return nil
}

func (i *inventoryItemRepository) GetItem(ctx context.Context, itemId
string) (*domain.InventoryItem, error) {
    out, err := i.dbClient.GetItem(ctx, &dynamodb.GetItemInput{
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemPk(itemId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemSk()},
        },
        TableName:      aws.String(i.properties.TableName),
        ConsistentRead: aws.Bool(true),
    })
}

```



```

    if err != nil {
        return nil, err
    }

    if len(out.Item) == 0 {
        return nil, nil
    }

    dbItem := &model.InventoryItem{}
    err = dbItem.FromAttributeValues(out.Item)
    if err != nil {
        return nil, err
    }

    return i.dbMapper.FromDbInventoryItem(dbItem), nil
}

func (i *inventoryItemRepository) GetByIds(ctx context.Context, ids
...string) ([]*domain.InventoryItem, error) {
    var (
        keys    []map[string]types.AttributeValue
        items    []map[string]types.AttributeValue
        result []*domain.InventoryItem
    )

    for _, id := range lo.Uniq(ids) {
        keys = append(keys, map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemPk(id)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemSk()},
        })
    }

    var unprocessedKeys = map[string]types.KeysAndAttributes{
        i.properties.TableName: {Keys: keys, ConsistentRead:
aws.Bool(true)},
    }

    for len(unprocessedKeys) > 0 {
        out, err := i.dbClient.BatchGetItem(ctx,
&dynamodb.BatchGetItemInput{
            RequestItems: map[string]types.KeysAndAttributes{
                i.properties.TableName: {
                    Keys: keys,
                    ConsistentRead: aws.Bool(true),
                },
            },
        })
        if err != nil {
            return nil, err
        }
    }
}

```

```

    items = append(items, out.Responses[i.properties.TableName]...)
    unprocessedKeys = out.UnprocessedKeys
}

for _, item := range items {
    dbItem := &model.InventoryItem{}
    err := dbItem.FromAttributeValues(item)
    if err != nil {
        return nil, err
    }

    result = append(result, i.dbMapper.FromDbInventoryItem(dbItem))
}

return result, nil
}

func (i *inventoryItemRepository) GetBookedSlotItems(ctx
context.Context, itemId string, date string)
([]*domain.InventoryItemSlot, error) {
    expr, _ := expression.NewBuilder().

        WithKeyCondition(expression.Key(model.HashKey).Equal(expression.Value(
model.InventoryItemSlotPk(itemId))).

        And(expression.Key(model.RangeKey).BeginsWith(model.InventoryItemSlots
k(date)))).
        Build()

    paginator := dynamodb.NewQueryPaginator(i.dbClient,
&dynamodb.QueryInput{
        TableName:            aws.String(i.properties.TableName),
        ExpressionAttributeNames: expr.Names(),
        ExpressionAttributeValues: expr.Values(),
        KeyConditionExpression:  expr.KeyCondition(),
    })

    var slots []*domain.InventoryItemSlot
    for paginator.HasMorePages() {
        page, err := paginator.NextPage(ctx)
        if err != nil {
            return nil, err
        }

        for _, item := range page.Items {
            slot := &model.InventoryItemSlot{}
            err := slot.FromAttributeValues(item)
            if err != nil {
                return nil, err
            }
            slots = append(slots, i.dbMapper.FromDbInventoryItemSlot(slot))
        }
    }

    return slots, nil
}

```

```

}

func (i *inventoryItemRepository) ItemsSlice(ctx context.Context,
sliceRequest *pkg.SliceRequest) (*pkg.Slice[*domain.InventoryItem],
error) {
    keyCondition :=
expression.Key(model.GSI1HashKey).Equal(expression.Value(model.Inventory
ItemGSI1Pk()))
    expr, err :=
expression.NewBuilder().WithKeyCondition(keyCondition).Build()
    if err != nil {
        return nil, err
    }

    return i.itemsSlice(ctx, &itemsSliceArgs{
        indexName:    model.GSI1,
        expression:    expr,
        sliceRequest: sliceRequest,
    })
}

func (i *inventoryItemRepository) ItemsByTypeSlice(ctx context.Context,
itemType string, sliceRequest *pkg.SliceRequest)
(*pkg.Slice[*domain.InventoryItem], error) {
    keyCondition :=
expression.Key(model.GSI2HashKey).Equal(expression.Value(model.Inventory
ItemGSI2Pk(itemType)))
    expr, err :=
expression.NewBuilder().WithKeyCondition(keyCondition).Build()
    if err != nil {
        return nil, err
    }

    return i.itemsSlice(ctx, &itemsSliceArgs{
        indexName:    model.GSI2,
        expression:    expr,
        sliceRequest: sliceRequest,
    })
}

func (i *inventoryItemRepository) BookItem(ctx context.Context, params
*domain.BookingItemParams) ([]*domain.InventoryItemSlot, error) {
    preparedInventoryItemBooking := i.prepareInventoryItemBooking(params,
params.Slots)
    preparedInventoryItemBookingAv, err :=
preparedInventoryItemBooking.AsAttributeValues()
    if err != nil {
        return nil, err
    }

    var writeItems []types.TransactWriteItem

    // Make sure inventory item exists
    itemExistsCheckIndex := 0
    writeItems = append(writeItems, types.TransactWriteItem{

```

```

        ConditionCheck: &types.ConditionCheck{
            ConditionExpression: aws.String("attribute_exists(#PK) AND
attribute_exists(#SK)"),
            Key: map[string]types.AttributeValue{
                model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemPk(params.ItemId)},
                model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemSk()},
            },
            TableName:
aws.String(i.properties.TableName),
            ExpressionAttributeNames:          map[string]string{"#PK":
model.HashKey, "#SK": model.RangeKey},
            ReturnValuesOnConditionCheckFailure:
types.ReturnValuesOnConditionCheckFailureAllOld,
        },
    })

    // CreateItem a booking making sure it doesn't already exist
    bookingDoesntExistCheckIndex := 1
    writeItems = append(writeItems, types.TransactWriteItem{
        Put: &types.Put{
            Item:
preparedInventoryItemBookingAv,
            TableName:
aws.String(i.properties.TableName),
            ConditionExpression:
aws.String("attribute_not_exists(#PK) AND attribute_not_exists(#SK)"),
            ExpressionAttributeNames:          map[string]string{"#PK":
model.HashKey, "#SK": model.RangeKey},
            ReturnValuesOnConditionCheckFailure:
types.ReturnValuesOnConditionCheckFailureAllOld,
        },
    })

    // Mark requested slots as booked, making sure they available
    slotsAraAvailableCheckStartingIndex := 2
    for _, slot := range params.Slots {
        dbSlot := i.dbMapper.ToDbInventoryItemSlot(slot)
        slotAv, err := dbSlot.AsAttributeValues()
        if err != nil {
            return nil, err
        }

        writeItems = append(writeItems, types.TransactWriteItem{
            Put: &types.Put{
                Item:                slotAv,
                TableName:
aws.String(i.properties.TableName),
                ConditionExpression:
aws.String("attribute_not_exists(#PK) AND attribute_not_exists(#SK)"),
                ExpressionAttributeNames:          map[string]string{"#PK":
model.HashKey, "#SK": model.RangeKey},
                ReturnValuesOnConditionCheckFailure:
types.ReturnValuesOnConditionCheckFailureAllOld,
            },
        })
    }

```

```

    },
  })
}

// Commit transaction
_, err = i.dbClient.TransactWriteItems(ctx,
&dynamodb.TransactWriteItemsInput{TransactItems: writeItems})
if err != nil {

  // Handle transaction canceled exception
  if tce := asTransactionCanceledException(err); tce != nil &&
len(tce.CancellationReasons) == len(writeItems) {

    // Inventory item does not exists
    if lo.FromPtr(tce.CancellationReasons[itemExistsCheckIndex].Code)
== conditionalCheckFailedCode {
      return nil, &domain.BusinessError{Message:
fmt.Sprintf("inventory item with ID %s was not found", params.ItemId)}
    }

    // Booking already processed
    if
lo.FromPtr(tce.CancellationReasons[bookingDoesntExistCheckIndex].Code)
== conditionalCheckFailedCode {
      booking := model.InventoryItemBooking{}
      err :=
booking.FromAttributeValues(tce.CancellationReasons[1].Item)
      if err != nil {
        return nil, err
      }

      var result []*domain.InventoryItemSlot
      for _, slot := range booking.BookedSlots {
        result = append(result, &domain.InventoryItemSlot{
          InventoryItemId: params.ItemId,
          BookedBy:         &params.BookingId,
          SlotStartTime:    slot.SlotStartTime,
          SlotEndTime:      slot.SlotEndTime,
        })
      }

      return result, nil
    }

    for idx := slotsAraAvailableCheckStartingIndex; idx <
len(tce.CancellationReasons); idx++ {
      if lo.FromPtr(tce.CancellationReasons[idx].Code) ==
conditionalCheckFailedCode {
        slot := model.InventoryItemSlot{}
        err =
slot.FromAttributeValues(tce.CancellationReasons[idx].Item)
        if err != nil {
          return nil, err
        }
      }
    }
  }
}

```

```

        errorMessage := fmt.Sprintf("inventory item %s is already
booked for the requested time slot (%s)", params.ItemId,
slot.SlotStartTime)
        return nil, &domain.BusinessError{Message: errorMessage}
    }
}
}
return nil, err
}

var result []*domain.InventoryItemSlot
for _, slot := range params.Slots {
    result = append(result, &domain.InventoryItemSlot{
        InventoryItemId: params.ItemId,
        SlotStartTime:   slot.SlotStartTime,
        SlotEndTime:     slot.SlotEndTime,
        BookedBy:        &params.BookingId,
    })
}

return result, nil
}

func (i *inventoryItemRepository) CancelBooking(ctx context.Context,
params *domain.CancelBookingParams) error {
    out, err := i.dbClient.GetItem(ctx, &dynamodb.GetItemInput{
        TableName: aws.String(i.properties.TableName),
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemBookingPk(params.ItemId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemBookingSk(params.BookingId)},
        },
    })

    if err != nil {
        return err
    }

    // Booking wasn't found, no actions required
    if len(out.Item) == 0 {
        return nil
    }

    booking := model.InventoryItemBooking{}
    err = booking.FromAttributeValues(out.Item)
    if err != nil {
        return err
    }

    var deleteItems []types.TransactWriteItem

    // Delete booking
    deleteItems = append(deleteItems, types.TransactWriteItem{
        Delete: &types.Delete{

```

```

        TableName: aws.String(i.properties.TableName),
        Key: map[string]types.AttributeValue{
            model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemBookingPk(params.ItemId)},
            model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemBookingSk(params.BookingId)},
        },
    },
    })

// Delete booked slots
for _, slot := range booking.BookedSlots {
    deleteItems = append(deleteItems, types.TransactWriteItem{
        Delete: &types.Delete{
            TableName: aws.String(i.properties.TableName),
            Key: map[string]types.AttributeValue{
                model.HashKey: &types.AttributeValueMemberS{Value:
model.InventoryItemSlotPk(params.ItemId)},
                model.RangeKey: &types.AttributeValueMemberS{Value:
model.InventoryItemSlotSk(slot.SlotStartTime)},
            },
        },
    })
}

// Commit transaction
_, err = i.dbClient.TransactWriteItems(ctx,
&dynamodb.TransactWriteItemsInput{TransactItems: deleteItems})
if err != nil {
    return err
}

return nil
}

func (i *inventoryItemRepository) prepareInventoryItemBooking(params
*domain.BookingItemParams, slots []*domain.InventoryItemSlot)
*model.InventoryItemBooking {
    var bookedSlots []model.BookedSlot
    for _, slot := range slots {
        bookedSlots = append(bookedSlots, model.BookedSlot{
            SlotStartTime: slot.SlotStartTime,
            SlotEndTime: slot.SlotEndTime,
        })
    }

    return &model.InventoryItemBooking{
        PK: model.InventoryItemBookingPk(params.ItemId),
        SK: model.InventoryItemBookingSk(params.BookingId),
        EntityType: model.InventoryItemBookingEntityType,
        BookedSlots: bookedSlots,
    }
}

type itemsSliceArgs struct {

```

```

    indexName    string
    expression    expression.Expression
    sliceRequest *pkg.SliceRequest
}

func (i *inventoryItemRepository) itemsSlice(ctx context.Context, args
*itemsSliceArgs) (*pkg.Slice[*domain.InventoryItem], error) {
    paginator := dynamodb.NewQueryPaginator(i.dbClient,
&dynamodb.QueryInput{
        TableName:      aws.String(i.properties.TableName),
        IndexName:      aws.String(args.indexName),
        ExpressionAttributeNames: args.expression.Names(),
        ExpressionAttributeValues: args.expression.Values(),
        KeyConditionExpression:  args.expression.KeyCondition(),
        FilterExpression:      args.expression.Filter(),
        ScanIndexForward:
aws.Bool(!o.FromPtr(args.sliceRequest.Ascending)),
        ExclusiveStartKey:      TokenToKey(args.sliceRequest.Token),
        Limit:                  aws.Int32(args.sliceRequest.Limit),
    })
    if !paginator.HasMorePages() {
        return &pkg.Slice[*domain.InventoryItem]{HasMore: false}, nil
    }

    page, err := paginator.NextPage(ctx)
    if err != nil {
        return nil, err
    }

    var data []*domain.InventoryItem
    for _, item := range page.Items {
        tournamentDb := &model.InventoryItem{}
        err := tournamentDb.FromAttributeValues(item)
        if err != nil {
            return nil, err
        }

        data = append(data, i.dbMapper.FromDbInventoryItem(tournamentDb))
    }

    return &pkg.Slice[*domain.InventoryItem]{
        Data:      data,
        HasMore:    paginator.HasMorePages(),
        NextToken:  KeyToToken(page.LastEvaluatedKey),
    }, nil
}

func TokenToKey(token *string) map[string]types.AttributeValue {
    if token == nil {
        return nil
    }
    sDec, _ := base64.StdEncoding.DecodeString(*token)
    key, _ := attributevalue.UnmarshalMapJSON(sDec)
    return key
}

```



```

func KeyToToken(key map[string]types.AttributeValue) *string {
    if len(key) == 0 {
        return nil
    }
    data, _ := attributevalue.MarshalMapJSON(key)
    return lo.ToPtr(base64.StdEncoding.EncodeToString(data))
}

func asTransactionCanceledException(err error)
*types.TransactionCanceledException {
    if err == nil {
        return nil
    }

    var tce *types.TransactionCanceledException
    if errors.As(err, &tce) {
        return tce
    }

    return nil
}

//Inventory/internal/repository/mapper/mapper.go
package mapper

import (
    "inventory/internal/domain"
    "inventory/internal/repository/model"

    "github.com/samber/lo"
)

type DbInventoryMapper interface {
    ToDbInventoryItem(item *domain.InventoryItem) *model.InventoryItem
    FromDbInventoryItem(item *model.InventoryItem) *domain.InventoryItem

    ToDbInventoryItemSlot(itemSlot *domain.InventoryItemSlot)
    *model.InventoryItemSlot
    FromDbInventoryItemSlot(itemSlot *model.InventoryItemSlot)
    *domain.InventoryItemSlot
}

type dbInventoryMapper struct {
}

func NewDbInventoryMapper() DbInventoryMapper {
    return &dbInventoryMapper{}
}

func (d *dbInventoryMapper) ToDbInventoryItem(item
*domain.InventoryItem) *model.InventoryItem {
    return &model.InventoryItem{
        PK: model.InventoryItemPk(item.Id),
        SK: model.InventoryItemSk(),
    }
}

```

```

        GSI1PK: model.InventoryItemGSI1Pk(),
        GSI1SK: model.InventoryItemGSI1Sk(item.Id),

        GSI2PK: model.InventoryItemGSI2Pk(item.Type),
        GSI2SK: model.InventoryItemGSI2Sk(item.Id),

        EntityType: model.InventoryItemEntityType,

        Id:          item.Id,
        Type:         item.Type,
        Name:         item.Name,
        Description:  item.Description,
    }
}

func (d *dbInventoryMapper) FromDbInventoryItem(item
*model.InventoryItem) *domain.InventoryItem {
    return &domain.InventoryItem{
        Id:          item.Id,
        Type:         item.Type,
        Name:         item.Name,
        Description:  item.Description,
    }
}

func (d *dbInventoryMapper) FromDbInventoryItemSlot(itemSlot
*model.InventoryItemSlot) *domain.InventoryItemSlot {
    return &domain.InventoryItemSlot{
        InventoryItemId: itemSlot.InventoryItemId,
        SlotStartTime:   itemSlot.SlotStartTime,
        SlotEndTime:     itemSlot.SlotEndTime,
        BookedBy:        &itemSlot.BookedBy,
    }
}

func (d *dbInventoryMapper) ToDbInventoryItemSlot(itemSlot
*domain.InventoryItemSlot) *model.InventoryItemSlot {
    return &model.InventoryItemSlot{
        PK:
        model.InventoryItemSlotPk(itemSlot.InventoryItemId),
        SK:          model.InventoryItemSlotSk(itemSlot.SlotStartTime),
        EntityType:   model.InventoryItemSlotEntityType,
        InventoryItemId: itemSlot.InventoryItemId,
        SlotStartTime: itemSlot.SlotStartTime,
        SlotEndTime:   itemSlot.SlotEndTime,
        BookedBy:      lo.FromPtr(itemSlot.BookedBy),
    }
}

//Inventory/internal/repository/model/inventory_item.go
package model

import (
    "fmt"

```

```

    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
    "github.com/aws/aws-sdk-go/service/dynamodb"
    "github.com/aws/aws-sdk-go/service/dynamodb/dynamodbattribute"
)

const (
    InventoryItemPkTemplate = "InventoryItem:%s"
    InventoryItemSkTemplate = "InventoryItem"
    InventoryItemEntityType = "InventoryItem"

    InventoryItemGSI1PkTemplate = "InventoryItems"
    InventoryItemGSI1SkTemplate = "InventoryItem:%s"

    InventoryItemGSI2PkTemplate = "InventoryItems:%s"
    InventoryItemGSI2SkTemplate = "InventoryItem:%s"
)

func InventoryItemPk(inventoryItemId string) string {
    return fmt.Sprintf(InventoryItemPkTemplate, inventoryItemId)
}

func InventoryItemSk() string {
    return fmt.Sprintf(InventoryItemSkTemplate)
}

func InventoryItemGSI1Pk() string {
    return fmt.Sprintf(InventoryItemGSI1PkTemplate)
}

func InventoryItemGSI1Sk(inventoryItemId string) string {
    return fmt.Sprintf(InventoryItemGSI1SkTemplate, inventoryItemId)
}

func InventoryItemGSI2Pk(itemType string) string {
    return fmt.Sprintf(InventoryItemGSI2PkTemplate, itemType)
}

func InventoryItemGSI2Sk(inventoryItemId string) string {
    return fmt.Sprintf(InventoryItemGSI2SkTemplate, inventoryItemId)
}

type InventoryItem struct {
    PK string `dynamodbav:"PK"`
    SK string `dynamodbav:"SK"`

    GSI1PK string `dynamodbav:"GSI1_PK"`
    GSI1SK string `dynamodbav:"GSI1_SK"`

    GSI2PK string `dynamodbav:"GSI2_PK"`
    GSI2SK string `dynamodbav:"GSI2_SK"`

    EntityType string `dynamodbav:"entityType"`
}

```

```

    Id            string `dynamodbav:"id"`
    Type          string `dynamodbav:"type"`
    Name          string `dynamodbav:"name"`
    Description   string `dynamodbav:"description"`
}

func (i *InventoryItem) AsAttributeValues()
(map[string]types.AttributeValue, error) {
    return attributevalue.MarshalMap(i)
}

func (i *InventoryItem) FromAttributeValues(av
map[string]types.AttributeValue) error {
    return attributevalue.UnmarshalMap(av, i)
}

func (i *InventoryItem) FromDynamoDbAttributeValue(av
map[string]*dynamodb.AttributeValue) error {
    return dynamodbattribute.UnmarshalMap(av, i)
}

func (i *InventoryItem) AsDynamoDbAttributeValue()
(map[string]*dynamodb.AttributeValue, error) {
    return dynamodbattribute.MarshalMap(i)
}

const (
    InventoryItemSlotPkTemplate = "InventoryItem:%s"
    InventoryItemSlotSkTemplate = "InventoryItemSlot:%s"
    InventoryItemSlotEntityType = "InventoryItemSlot"
)

func InventoryItemSlotPk(inventoryItemId string) string {
    return fmt.Sprintf(InventoryItemSlotPkTemplate, inventoryItemId)
}

func InventoryItemSlotSk(slotStartTime string) string {
    return fmt.Sprintf(InventoryItemSlotSkTemplate, slotStartTime)
}

type InventoryItemSlot struct {
    PK            string `dynamodbav:"PK"`
    SK            string `dynamodbav:"SK"`
    EntityType    string `dynamodbav:"entityType"`

    InventoryItemId string `dynamodbav:"inventoryItemId"`
    SlotStartTime   string `dynamodbav:"slotStartTime"`
    SlotEndTime     string `dynamodbav:"slotEndTime"`

    BookedBy      string `dynamodbav:"bookedBy"`
}

func (i *InventoryItemSlot) AsAttributeValues()
(map[string]types.AttributeValue, error) {
    return attributevalue.MarshalMap(i)
}

```

```

}

func (i *InventoryItemSlot) FromAttributeValues(av
map[string]types.AttributeValue) error {
    return attributevalue.UnmarshalMap(av, i)
}

func (i *InventoryItemSlot) FromDynamoDbAttributeValue(av
map[string]*dynamodb.AttributeValue) error {
    return dynamodbattribute.UnmarshalMap(av, i)
}

func (i *InventoryItemSlot) AsDynamoDbAttributeValue()
(map[string]*dynamodb.AttributeValue, error) {
    return dynamodbattribute.MarshalMap(i)
}

const (
    InventoryItemBookingPkTemplate = "InventoryItem:%s"
    InventoryItemBookingSkTemplate = "InventoryItemBooking:%s"
    InventoryItemBookingEntityType = "InventoryItemBooking"
)

func InventoryItemBookingPk(inventoryItemId string) string {
    return fmt.Sprintf(InventoryItemBookingPkTemplate, inventoryItemId)
}

func InventoryItemBookingSk(bookingId string) string {
    return fmt.Sprintf(InventoryItemBookingSkTemplate, bookingId)
}

type InventoryItemBooking struct {
    PK          string `dynamodbav:"PK"`
    SK          string `dynamodbav:"SK"`
    EntityType  string `dynamodbav:"entityType"`

    BookedSlots []BookedSlot `dynamodbav:"bookedSlots"`
}

type BookedSlot struct {
    SlotStartTime string `dynamodbav:"slotStartTime"`
    SlotEndTime   string `dynamodbav:"slotEndTime"`
}

func (i *InventoryItemBooking) AsAttributeValues()
(map[string]types.AttributeValue, error) {
    return attributevalue.MarshalMap(i)
}

func (i *InventoryItemBooking) FromAttributeValues(av
map[string]types.AttributeValue) error {
    return attributevalue.UnmarshalMap(av, i)
}

```

```

func (i *InventoryItemBooking) FromDynamoDbAttributeValue(av
map[string]*dynamodb.AttributeValue) error {
    return dynamodbattribute.UnmarshalMap(av, i)
}

func (i *InventoryItemBooking) AsDynamoDbAttributeValue()
(map[string]*dynamodb.AttributeValue, error) {
    return dynamodbattribute.MarshalMap(i)
}

//Inventory/internal/service/inventory_service.go
package service

import (
    "context"
    "fmt"
    "inventory/internal/domain"
    "inventory/pkg"
    "time"
)

type inventoryItemService struct {
    clock      pkg.Clock
    repository domain.InventoryItemRepository
}

func NewInventoryItemService(
    clock pkg.Clock,
    repository domain.InventoryItemRepository,
) domain.InventoryItemService {
    return &inventoryItemService{
        clock:      clock,
        repository: repository,
    }
}

func (i *inventoryItemService) CreateItem(ctx context.Context, item
*domain.InventoryItem) error {
    return i.repository.CreateItem(ctx, item)
}

func (i *inventoryItemService) GetItem(ctx context.Context, itemId
string) (*domain.InventoryItem, error) {
    return i.repository.GetItem(ctx, itemId)
}

func (i *inventoryItemService) GetItemsByIds(ctx context.Context, ids
...string) ([]*domain.InventoryItem, error) {
    return i.repository.GetByIds(ctx, ids...)
}

func (i *inventoryItemService) GetItemAvailability(ctx context.Context,
itemId string, date string) ([]*domain.InventoryItemSlot, error) {
    item, err := i.repository.GetItem(ctx, itemId)
    if err != nil {

```

```

    return nil, err
}

if item == nil {
    return nil, &domain.BusinessError{Message: fmt.Sprintf("item with ID
%s not found", itemId)}
}

dateMilli, err :=
i.clock.ToMilliseconds(fmt.Sprintf("%sT00:00:00.000", date))
if err != nil {
    return nil, fmt.Errorf("invalid date format")
}

slots, err := i.repository.GetBookedSlotItems(ctx, itemId, date)
if err != nil {
    return nil, err
}

bookedByTimeSlots := make(map[string]*domain.InventoryItemSlot)
for _, slot := range slots {
    bookedByTimeSlots[slot.SlotStartTime] = slot
}

dateTime := time.UnixMilli(dateMilli).UTC()
tomorrow := dateTime.AddDate(0, 0, 1)

var result []*domain.InventoryItemSlot
for slotTime := dateTime; slotTime.Before(tomorrow); slotTime =
slotTime.Add(1 * time.Hour) {
    if bookedSlot, ok :=
bookedByTimeSlots[i.clock.ToString(slotTime.UnixMilli())]; ok {
        result = append(result, bookedSlot)
        continue
    }

    result = append(result, &domain.InventoryItemSlot{
        InventoryItemId: itemId,
        SlotStartTime:    i.clock.ToString(slotTime.UnixMilli()),
        SlotEndTime:      i.clock.ToString(slotTime.Add(1 *
time.Hour).UnixMilli()),
    })
}

return result, nil
}

func (i *inventoryItemService) BookItem(ctx context.Context, params
*domain.BookingItemParams) ([]*domain.InventoryItemSlot, error) {
    slotsForBooking, err := i.sliceSlotsForBooking(params)
    if err != nil {
        return nil, err
    }

    params.Slots = slotsForBooking

```

```

    return i.repository.BookItem(ctx, params)
}

func (i *inventoryItemService) sliceSlotsForBooking(params
*domain.BookingItemParams) ([]*domain.InventoryItemSlot, error) {
    unixSecondsStartTime, err := i.clock.ToMilliseconds(params.StartTime)
    if err != nil {
        return nil, err
    }

    unixStartTime := time.UnixMilli(unixSecondsStartTime).UTC()
    unixStartTimeTruncated := unixStartTime.Truncate(1 * time.Hour)

    unixSecondsEndTime, err := i.clock.ToMilliseconds(params.EndTime)
    if err != nil {
        return nil, err
    }

    unixEndTime := time.UnixMilli(unixSecondsEndTime).UTC()
    unixEndTimeTruncated := unixEndTime.Add(1 * time.Hour).Truncate(1 *
time.Hour)

    if unixSecondsEndTime < unixSecondsStartTime {
        return nil, fmt.Errorf("invalid booking time range: start time must
be before end time")
    }

    var result []*domain.InventoryItemSlot
    for start := unixStartTimeTruncated;
start.Before(unixEndTimeTruncated); start = start.Add(1 * time.Hour) {
        slotStartTime := i.clock.ToString(start.UnixMilli())
        slotEndTime := i.clock.ToString(start.Add(1 *
time.Hour).UnixMilli())

        result = append(result, &domain.InventoryItemSlot{
            InventoryItemId: params.ItemId,
            SlotStartTime:    slotStartTime,
            SlotEndTime:      slotEndTime,
            BookedBy:         &params.BookingId,
        })
    }

    return result, nil
}

func (i *inventoryItemService) CancelBooking(ctx context.Context, params
*domain.CancelBookingParams) error {
    return i.repository.CancelBooking(ctx, params)
}

func (i *inventoryItemService) ItemsSlice(ctx context.Context,
sliceRequest *pkg.SliceRequest) (*pkg.Slice[*domain.InventoryItem],
error) {
    return i.repository.ItemsSlice(ctx, sliceRequest)
}

```



```
func (i *inventoryItemService) ItemsByTypeSlice(ctx context.Context,
itemType string, sliceRequest *pkg.SliceRequest)
(*pkg.Slice[*domain.InventoryItem], error) {
    return i.repository.ItemsByTypeSlice(ctx, itemType, sliceRequest)
}
```