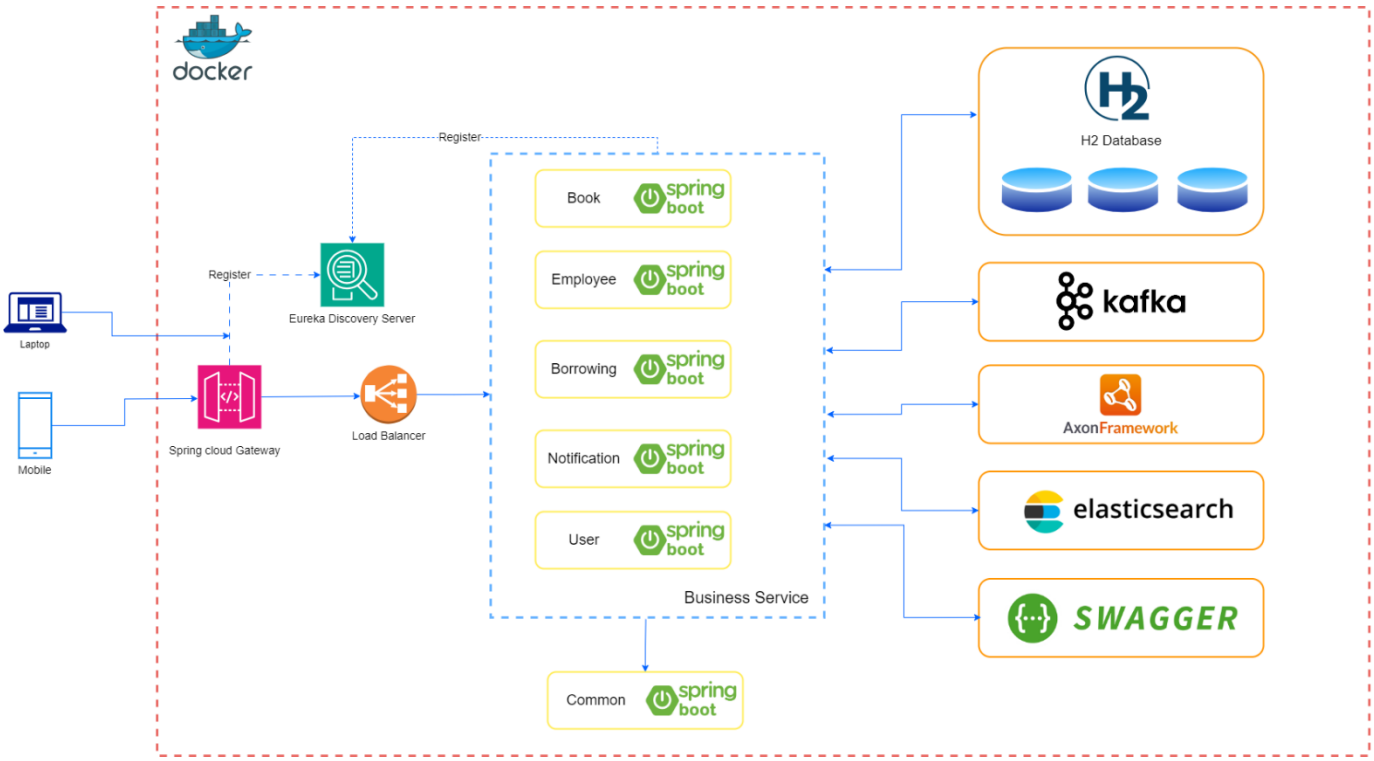
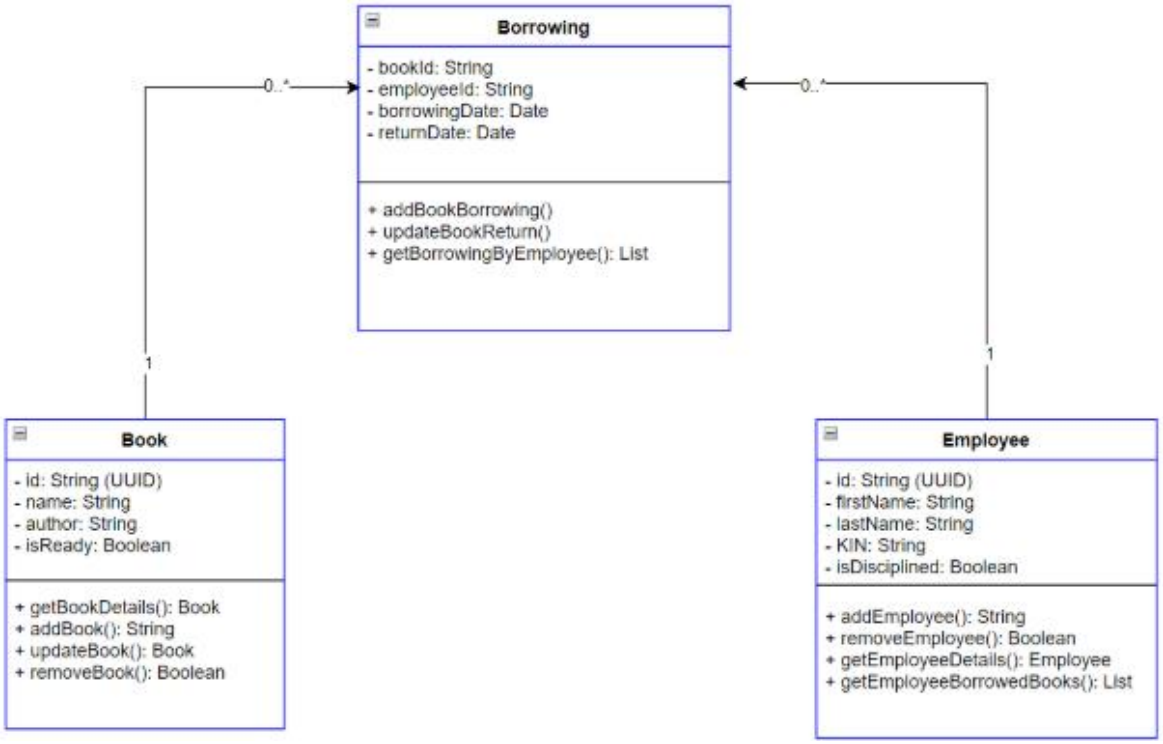
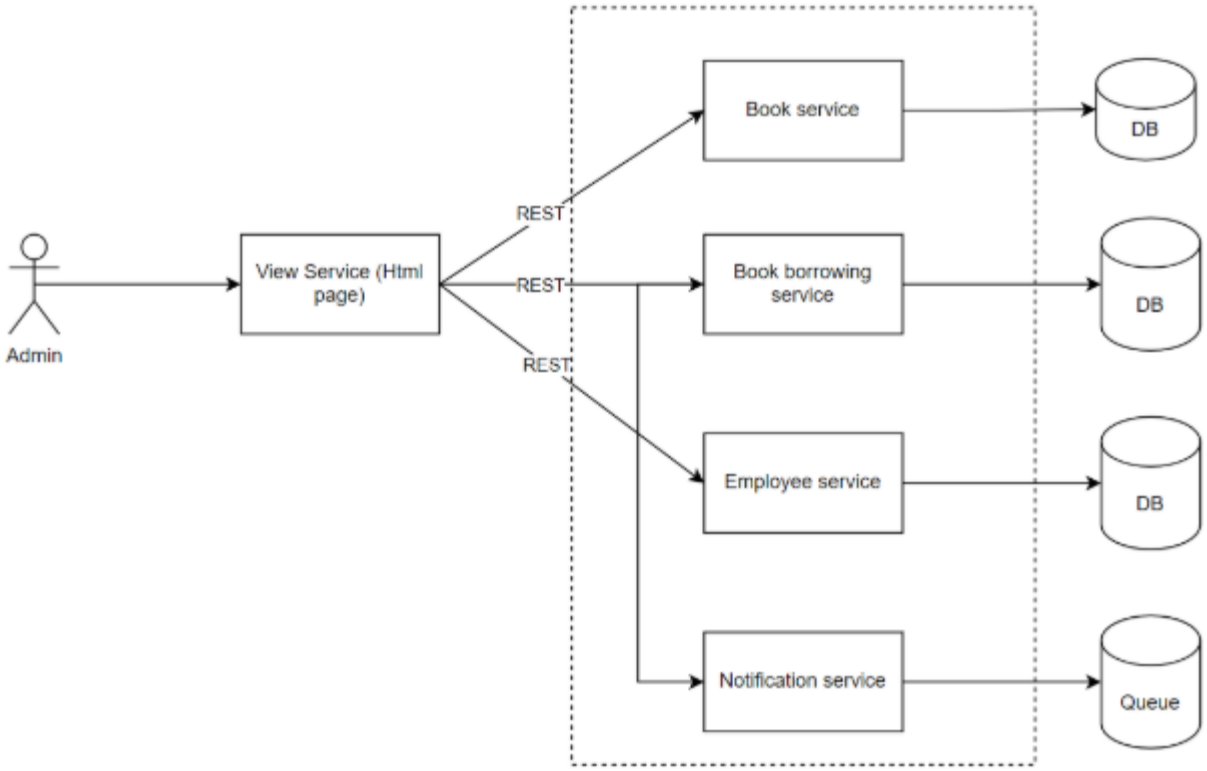


Sơ đồ tổng quan dự án



Sơ đồ Database



Architecture

1. Các kiến thức cần nắm

- Các khái niệm trong Axon Framework
  - Là 1 framework được ứng dụng phổ biến trong microservice để xây dựng các ứng dụng hướng sự kiện
  - **Aggregate**: Hiểu đơn giản nó giống như Entity trong SpringBoot
  - **Command**: là 1 lệnh dùng để thay đổi trạng thái ứng dụng. Lưu ý các command này chưa thật sự thay đổi trạng thái của nó
  - **Event**: Trái ngược với command khi event xuất hiện tức là đã có sự thay đổi trạng thái nào đó đã xảy ra. Cụ thể sẽ xuất hiện sau khi các command đã sử lý thành công. Các event này chủ yếu dùng để gửi thông báo đến nơi cần được xử lý khi trạng thái ứng dụng bị thay đổi
  - **Query**: Loại Message thứ ba này đơn giản chỉ là một request lấy thông tin nào đó, có thể là thông tin trạng thái hiện tại của ứng dụng
- Mô hình Microservice

Mô hình microservice là một kiến trúc phần mềm trong đó một ứng dụng lớn được chia nhỏ thành các dịch vụ nhỏ độc lập, mỗi dịch vụ đảm nhận một chức năng cụ thể và hoạt động như một ứng dụng riêng lẻ. Đây là một sự cải tiến từ mô hình monolithic, nơi toàn bộ ứng dụng được gộp chung trong một khối mã nguồn duy nhất

Đặc điểm:

1. Độc lập:
    - Mỗi service có thể được phát triển, triển khai và bảo trì một cách độc lập mà không ảnh hưởng đến các microservice khác
    - Mỗi dịch vụ có thể sử dụng ngôn ngữ lập trình, cơ sở dữ liệu, hoặc framework khác nhau phù hợp với chức năng của nó
2. Phân chia theo domain:
- Các dịch vụ được thiết kế xung quanh các domain logic cụ thể. Ví dụ: quản lý người dùng, thanh toán, xử lý đơn hàng
3. Truyền thông qua giao thức nhẹ:
- Các dịch vụ giao tiếp với nhau qua giao thức nhẹ như HTTP/REST hoặc thông qua message broker (RabbitMQ,KafKa)
4. Tính mở rộng:
- Mỗi microsevice có thể mở rộng riêng lẻ theo nhu cầu giúp tối ưu tài nguyên

Công nghệ phổ biến trong microservice

1. Containers và Orchestration
- **Docker**, **Kubernetes**
2. Giao tiếp giữa các dịch vụ
- **REST API**, RabbitMQ, **Kafka**
3. Quản lý dịch vụ:
- Service discovery: **Eureka**, Consul
- API Gateway: **Kong**, NGINX(cổng trung gian để giao tiếp giữa các service và client)
- DDD(Domain Driven Design)

Là 1 design pattern

- CQRS Pattern

Trong mô hình CQRS, Command và Querry là hai khái niệm cốt lõi, đảm nhận các vai trò khác nhau trong quản lý và truy vấn dữ liệu

I. Command(lệnh)

Định nghĩa:

- Command là các yêu cầu ghi(write), dùng để thay đổi trạng thái của hệ thống
- Các Command không trả về dữ liệu vì mục tiêu của chúng là thực hiện một hành động hoặc một thay đổi

Đặc điểm:

- Mỗi command biểu diễn một ý định cụ thể từ người dùng, ví dụ như thêm mới, cập nhật, hoặc xóa
  Ví dụ thực tế:
- Đăng ký người dùng: RegisterCommand
- Đặt sách: PlaceOrderCommand
  Luồng xử lý:
  1. Người dùng hoặc dịch vụ gửi yêu cầu command
  2. CommandHandler nhận Command và thực thi logic
  3. Thay đổi trạng thái thực hiện, thường thông qua 1 cơ sở dữ liệu dành cho việc ghi

II. Query(truy vấn)

Định nghĩa:

- Query là các yêu cầu đọc(read), dùng để truy vấn và trả về dữ liệu từ hệ thống
- Query không thay đổi trạng thái của hệ thống

Đặc điểm:

- Mục tiêu chính của Query là lấy dữ liệu từ hệ thống để hiển thị hoặc sử dụng trong các logic khác
- Không thay đổi trạng thái

Ví dụ: Lấy thông tin người dùng: GetUserByIdQuery

Luồng xử lý của Query:

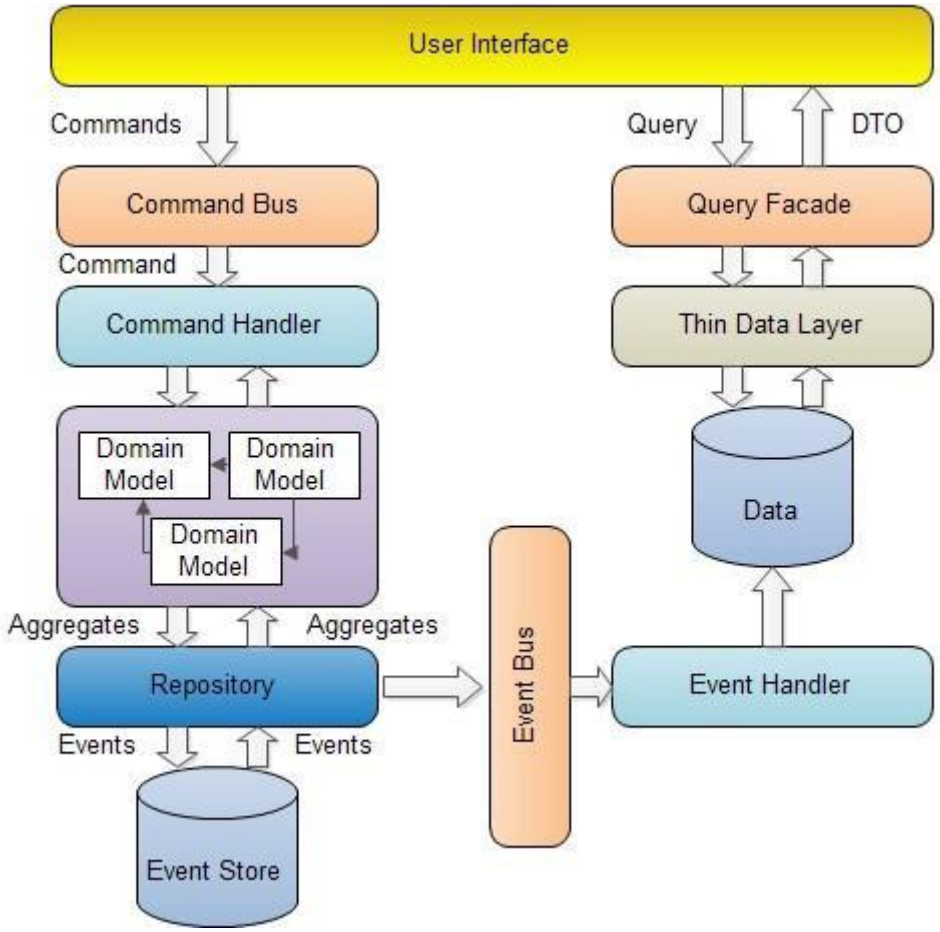
1. Người dùng hoặc dịch vụ gửi yêu cầu Query
2. QueryHandler nhận Query và truy vấn dữ liệu từ cơ sở dữ liệu đọc(read database)
3. Trả về kết quả cho người gọi

III. Sự phân tách Command và Querry trong CQRS

1. Chuyên biệt hóa trách nhiệm:
    - Command xử lý logic ghi phức tạp, bao gồm xác thực, kiểm tra điêu kiện, và áp dụng thay đổi
    - Query tối ưu cho hiệu suất đọc, có thể sử dụng các bảng tối ưu hóa hoặc các mô hình dữ liệu riêng biệt

IV. Các thành phần trong CQRS

1. Command
- Yêu cầu ghi
- Được xử lý bởi Command Handler
2. Query
- Yêu cầu đọc
- Được xử lý bởi Query Handler
3. Event
- Được sinh ra từ Command khi có thay đổi trạng thái và có thể được xử lý bởi các thành phần khác để đồng bộ hóa dữ liệu hoặc tạo thông báo



- Event Sourcing

Event Sourcing là một mẫu thiết kế (design pattern) trong hệ thống phần mềm, trong đó trạng thái của ứng dụng không được lưu trực tiếp dưới dạng các đối tượng hoặc dữ liệu cuối cùng mà thay vào đó được tái tạo từ một chuỗi các sự kiện (events). Các sự kiện này đại diện cho các thay đổi trạng thái đã xảy ra trong hệ thống theo thời gian.

1. Cách hoạt động:

Trong Event Sourcing:

- Sự kiện(Event): Mỗi thay đổi trạng thái của hệ thống được biểu diễn dưới dạng một sự kiện và được lưu trữ một cách bất biến trong Event Store

- Tái tạo trạng thái: Trạng thái hiện tại của ứng dụng không được lưu trữ trực tiếp mà được xây dựng lại bằng cách áp dụng lần lượt các sự kiện từ Event Store
- Không có ghi đè: Dữ liệu không bị mất đi; mọi thay đổi trạng thái đều được lưu dưới dạng sự kiện

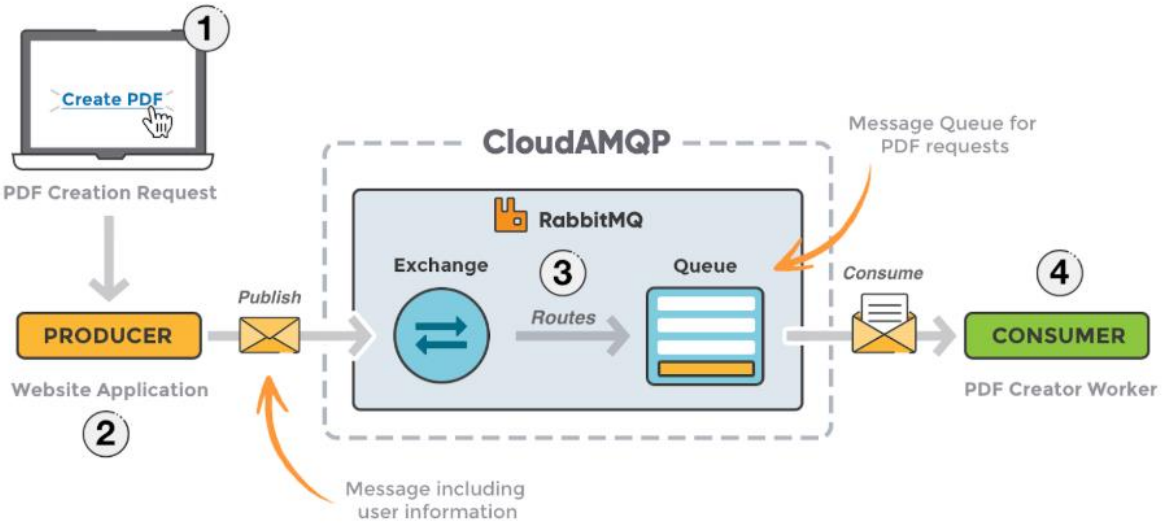
2. Thành phần chính
    - Event: Đại diện cho một hành động hoặc thay đổi đã xảy ra trong hệ thống
    - Event Store: Một cơ sở dữ liệu chuyên dụng để lưu trữ các sự kiện theo thời gian, có thể là cơ sở dữ liệu quan hệ, NoSQL, hoặc các giải pháp lưu trữ tùy chỉnh
    - Command: Một hành động yêu cầu thay đổi trạng thái, sau đó sinh ra một hoặc nhiều sự kiện
    - Event Handler: Thành phần chính xử lý các sự kiện
    - Read Model
3. Quy trình hoạt động
    - Command gửi yêu cầu: người dùng hoặc hệ thống gửi một command
    - Sinh sự kiện: Command được xử lý, kiểm tra logic nghiệp vụ, và sinh ra một hoặc nhiều sự kiện
    - Lưu trữ sự kiện: Sự kiện được lưu trữ trong Event Store
    - Xử lý sự kiện: Các Event Handler lắng nghe sự kiện và thực hiện các hành động liên quan , như cập nhật cơ sở dữ liệu đọc hoặc thông báo đến người dùng
    - Tái tạo trạng thái(khi cần): Khi cần trạng thái hiện tại của một đối tượng, hệ thống áp dụng lần lượt tất cả các sự kiện liên quan từ Event Store để tái tạo trạng thái

- Message Queue

Message queue là một cơ chế giao tiếp giữa các ứng dụng hoặc thành phần của hệ thống thông qua việc gửi và nhận thông điệp. Các thông điệp này được đặt trong một hàng đợi (queue) và chờ xử lý theo thứ tự, giúp các ứng dụng giao tiếp một cách **không đồng bộ**

Các công nghệ phổ biến: **Apache Kafka**, RabbitMQ, Amazon SQS, ActiveMQ, RocketMQ, MSMQ

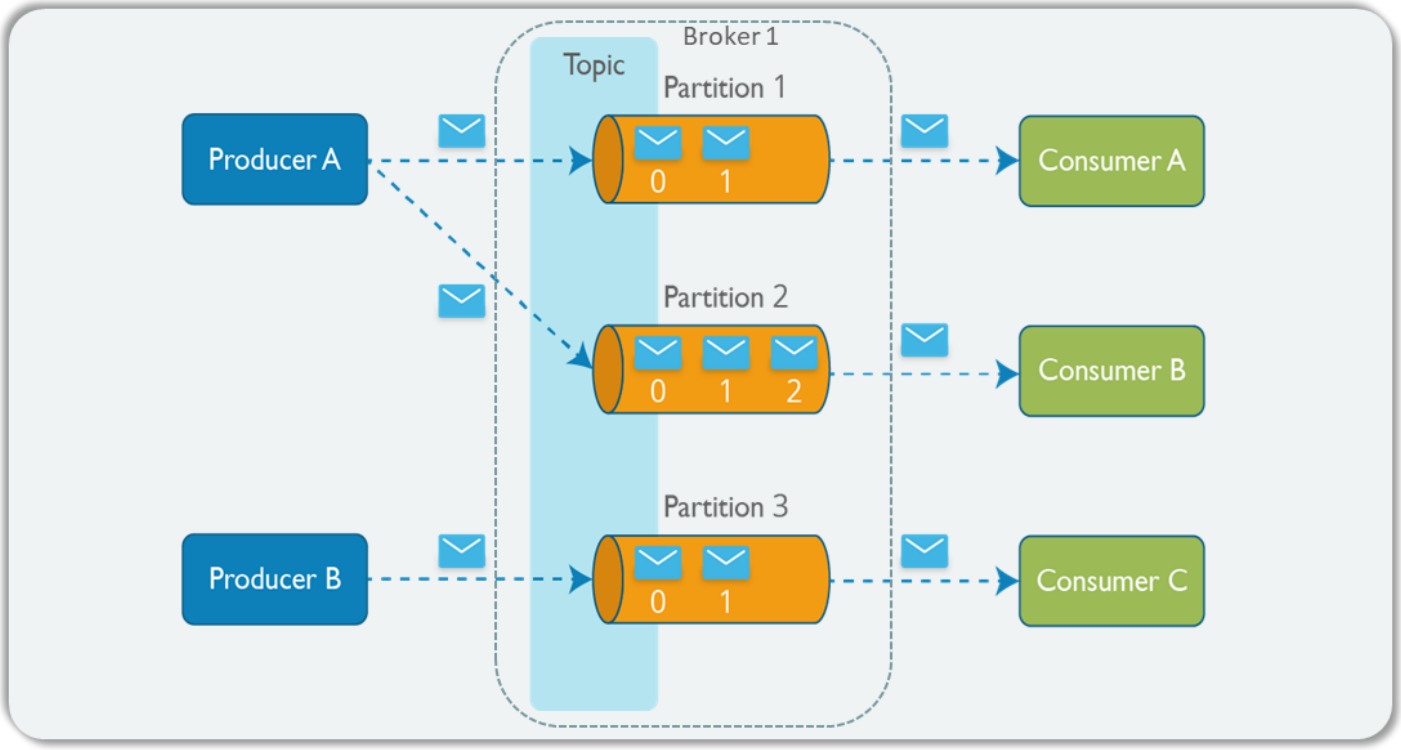
Usecase của Message Queue



Tại sao phải sử dụng Message Queue

Ưu điểm về Message Queue

- Dễ scaling hệ thống:
- Phân tán hệ thống
- Đảm bảo duration/recovery:
- Hỗ trợ rate limit, batch process

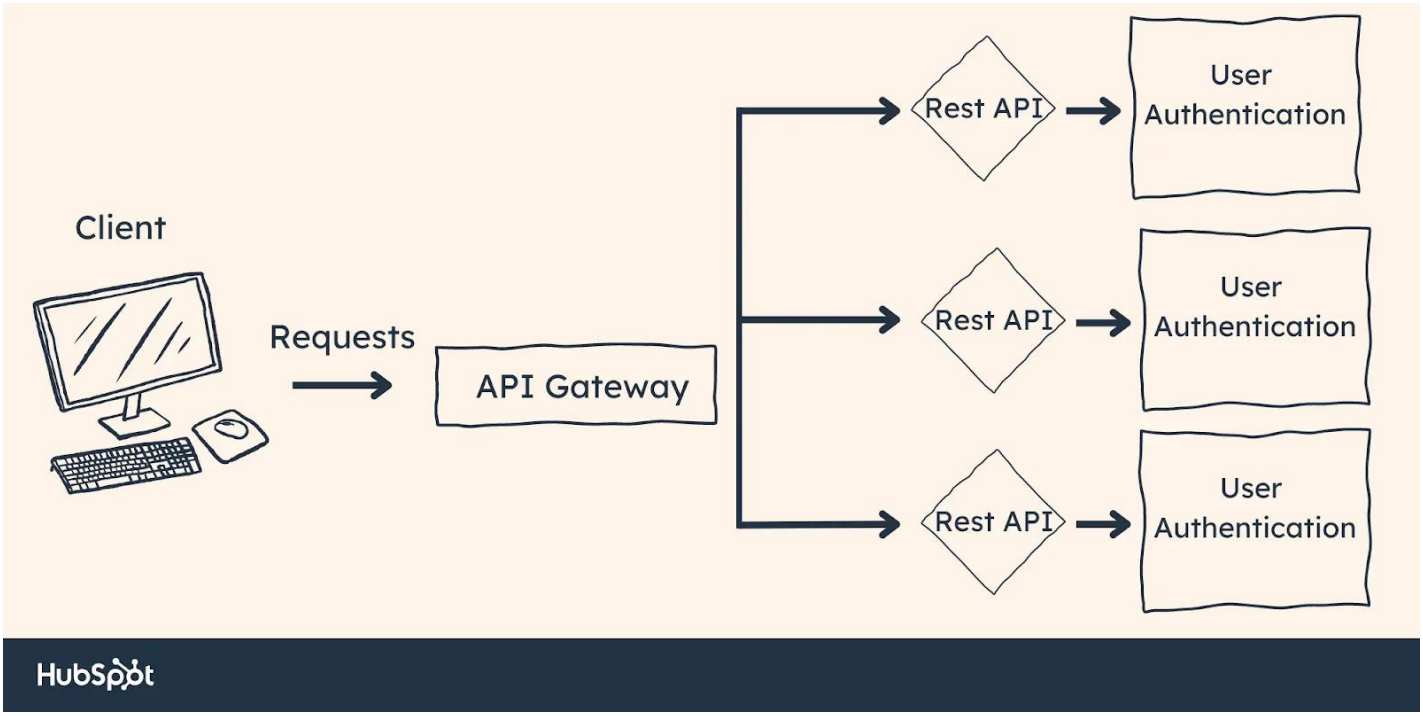


Kiến trúc KafKa

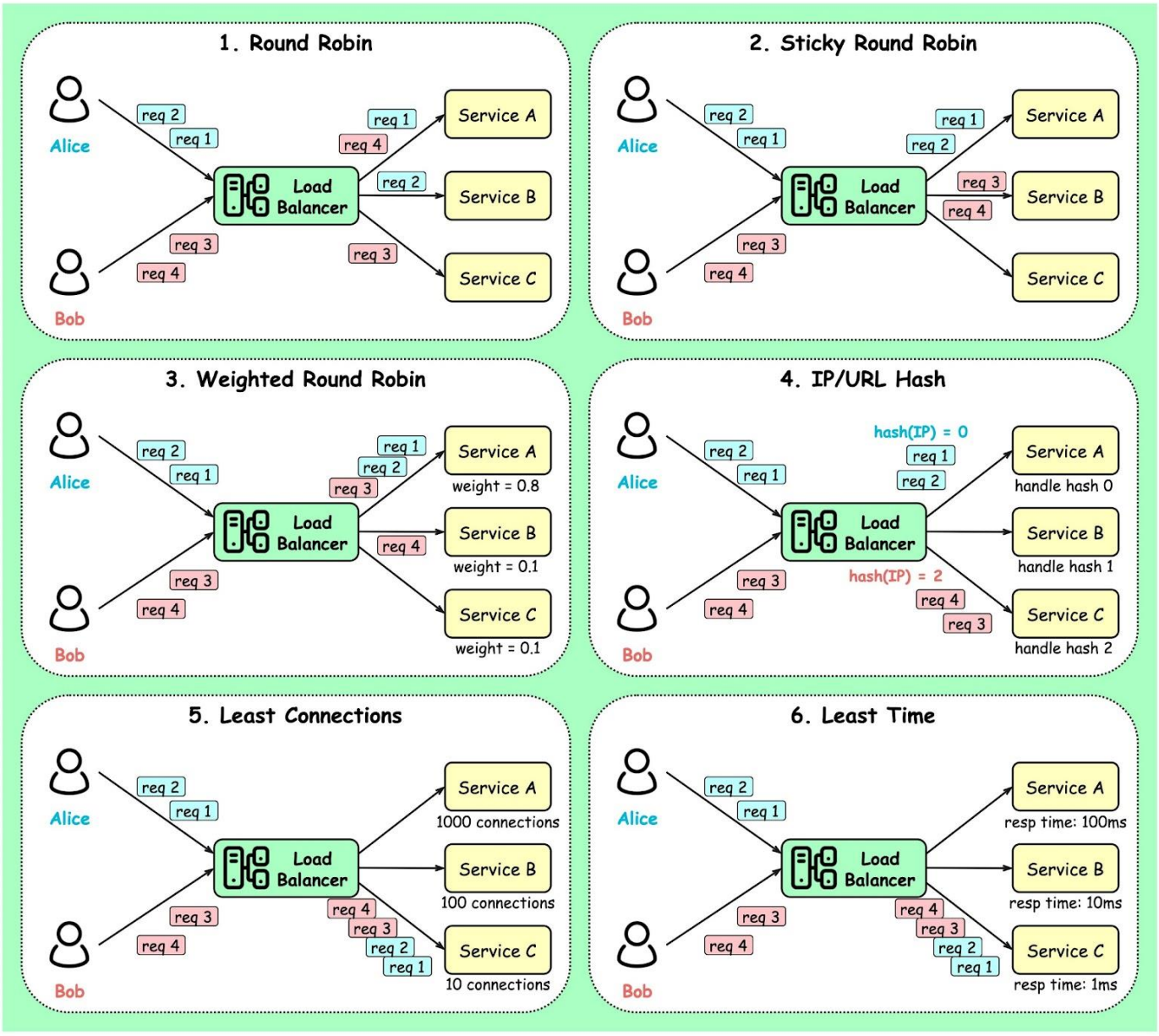
- **Producer**: Một producer có thể là bất kỳ ứng dụng nào có chức năng public message vào một topic
- **Messages**: Messages đơn thuần là byte array và developer có thể sử dụng chúng để lưu bất kỳ object với bất kỳ format nào – thông thường là String, JSON
- **Topic**: Một topic là một category hoặc feed name nơi mà record được publish
- **Partitions**: Các topic được chia nhỏ vào các đoạn khác nhau, các đoạn này được gọi là partition
- **Consumer**: Một consumer có thể là bất kỳ ứng dụng nào có chức năng subscribe vào một topic và tiêu thụ các tin nhắn
- **Broker**: Kafka cluster là một set các server, mỗi một set này được gọi là 1 broker

- Load Balancing và các thuật toán

Load balancing (cân bằng tải) là một kỹ thuật hoặc cơ chế phân phối lưu lượng mạng hoặc yêu cầu xử lý đến nhiều máy chủ, tài nguyên hoặc nút khác nhau để đảm bảo rằng không có một máy chủ nào bị quá tải. Mục tiêu chính của load balancing là tối ưu hóa tài nguyên, tăng hiệu suất, đảm bảo tính sẵn sàng cao và giảm thời gian phản hồi. Mô phỏng trong hình bên dưới:

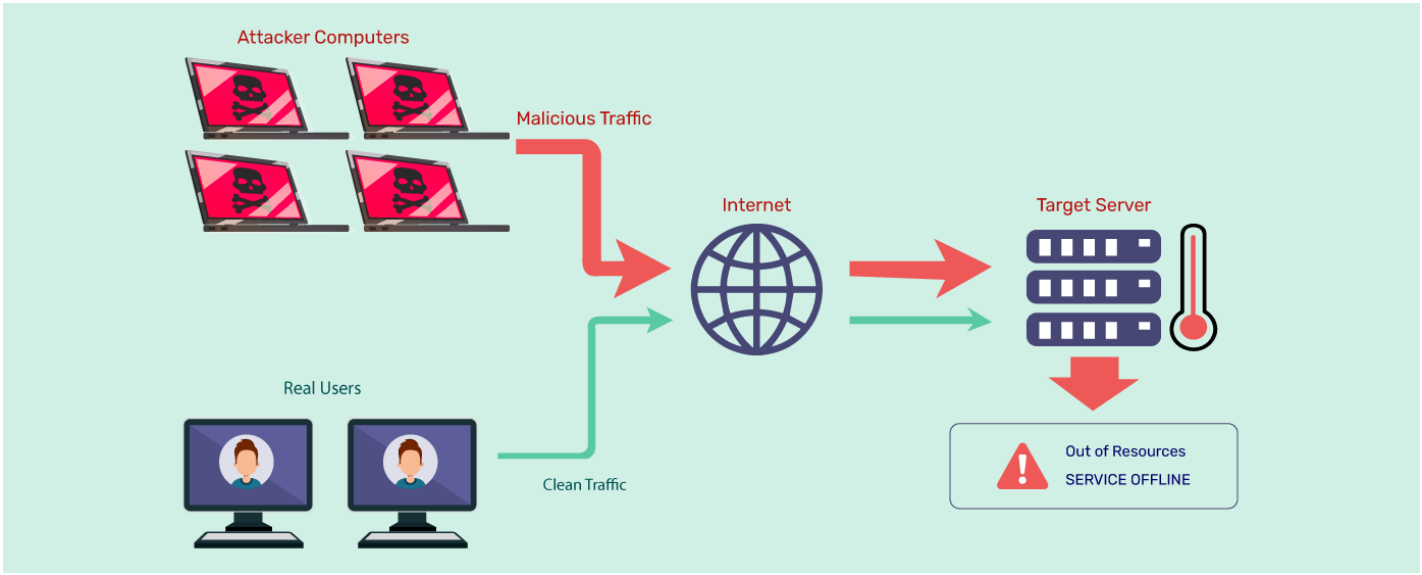


Các thuật toán Loadbalancing(phổ biến nhất là Round Robin)



- Rate Limiting và kiến trúc thường gặp

Rate Limit tức là hạn chế (limit) số lượng request gửi/nhận (rate) đến hệ thống. nói thì nghe đơn giản vậy thôi. Trên thực tế, người ta phải sử dụng 1 số thuật toán để đảm bảo chạy nhanh, chính xác mà lại ít tốn bộ nhớ. Giả sử như hệ thống của chúng ta nhận được hàng nghìn request nhưng mà trong số đó chỉ xử lí được trăm request/s chẳng hạn, và số request còn lại thì bị lỗi (do CPU hệ thống đang quá tải không thể xử lí được). Ví dụ: Mỗi người dùng chỉ cho phép nhập sai thẻ credit 3 lần trong 1 ngày (tránh tình trạng Brute force thông tin thẻ credit card)

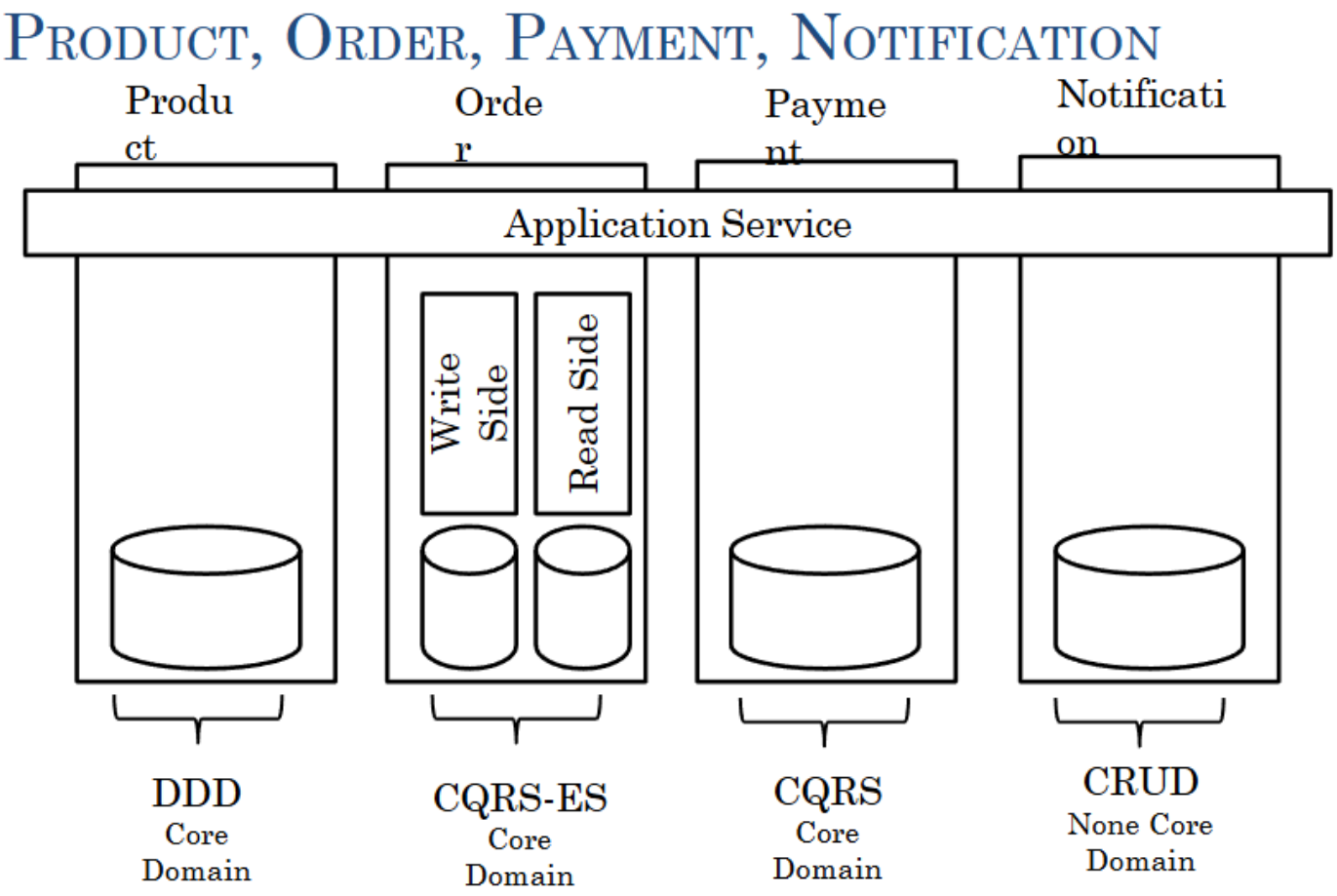


Tấn công DDoS

Các thuật toán thường gặp:

- Thuật toán Token Bucket
- Thuật toán Leaky Bucket
- Thuật toán Fixed Window Counter
- Thuật toán Sliding Window Logs

- Mối liên hệ giữa DDD, CQRS, CRUD trong 1 mô hình microservice:



V. Ứng dụng vào Project

1. Hệ thống API

I. Book Service API

Functionality	Method	Path
Get book details	GET	/api/v1/book/{id}
Add book	POST	/api/v1/books
Update book	PUT	/api/v1/books/{id}
Delete book	DELETE	/api/v1/books/{id}

II. Book Borrowing Service API

Functionality	Method	Path
Get book borrowing by employee	GET	/api/v1/borrowing/{id}
Add a new borrowing	POST	/api/v1/borrowing

Update a book return	PUT	/api/v1/borrowing/{employeeID}/{bookID}

III. Employee Service API

Functionality	Method	Path
Get employee details	GET	/api/v1/employees/{ID}
Get borrowed books for employee	GET	/api/v1/employees/{ID}/books
Add new employee	POST	/api/v1/employees
Remove employee	DELETE	/api/v1/employee/{ID}