001

Giới thiệu Challenge 10: Tập trung vào việc xây dựng microservices dựa trên sự kiện (event-driven).

Vấn đề Temporal Coupling (ràng buộc thời gian):

- Xảy ra khi một microservice cần phản hồi ngay lập tức từ một microservice khác để tiếp tục xử lý.
- Điều này làm tăng sự phụ thuộc và có thể gây chậm trễ toàn hệ thống nếu một microservice bị chậm hoặc không phản hồi.

So sánh:

- Loose Coupling: Tách biệt logic giữa các microservice để dễ phát triển, triển khai, mở rộng.
- Temporal Coupling: Một microservice bị chặn vì phải chờ microservice khác phản hồi.
- Synchronous vs Asynchronous Communication:
  - Synchronous: Giao tiếp đồng bộ như REST API — có thể gây temporal coupling.
    - Imperative: Thread chờ phản hồi.
    - Reactive: Thread được trả về pool nhưng vẫn là synchronous.
  - Asynchronous: Giao tiếp bất đồng bộ giúp tránh temporal coupling.
- Giới thiệu Event-Driven Microservices:
  - Giao tiếp qua các sự kiện (event) như trong ví dụ về đặt hàng trong Amazon — khi order được xác nhận, gửi sự kiện cho delivery service mà không chờ phản hồi.
  - Lợi ích: Tăng hiệu suất, giảm phụ thuộc thời gian giữa các service.
  - Cách xây dựng:

- ◆ Áp dụng kiến trúc hướng sự kiện.
- ◆ Sử dụng công nghệ như **Spring Cloud Function**, **Spring Cloud Stream**.
- ◆ Dùng các event broker để gửi và nhận event.

002

Event-driven microservices đang được áp dụng để xây dựng hệ thống microservice.

Có hai mô hình sự kiện chính:

Pub/Sub (Publisher-Subscriber):

- Các producer gửi sự kiện đến tất cả subscriber đã đăng ký.
- Sự kiện không thể phát lại.
- Các subscriber đến sau không nhận được sự kiện trước đó.

Event Streaming:

- Producer ghi sự kiện theo thứ tự vào một log (luồng sự kiện).
- Consumer có thể đọc bất kỳ phần nào trong luồng, cho phép phát lại sự kiện.
- Linh hoạt hơn, phù hợp khi cần xử lý lại dữ liệu cũ.

Lựa chọn mô hình tùy theo nhu cầu kinh doanh:

- Nếu không cần phát lại sự kiện → dùng Pub/Sub.
- Nếu cần khả năng truy xuất lại dữ kiện cũ → dùng Event Streaming.
- Với event streaming, có thể cấu hình vô hiệu hóa phát lại nếu cần.

Công cụ tương ứng:

- Pub/Sub → sử dụng RabbitMQ.
- Event Streaming → sử dụng Apache Kafka.

Trong khoá học, cả hai công cụ sẽ được giảng dạy riêng biệt:

- Phần đầu: RabbitMQ.
- Phần sau: Apache Kafka.

003

Tình huống ban đầu:

- Hệ thống gồm các microservices: **Accounts**, **Cards**, **Loans**.
- Hiện tại đang sử dụng **giao tiếp đồng bộ (synchronous)** do yêu cầu phản hồi nhanh cho người dùng.

Mục tiêu mới:

- Giới thiệu **giao tiếp bất đồng bộ (asynchronous)** bằng mô hình **Pub/Sub** để xử lý **gửi thông báo (email/SMS)** sau khi tạo tài khoản.

Kịch bản được xây dựng:

1. **Người dùng tạo tài khoản** thông qua Accounts microservice.
2. Microservice này sẽ:
    - Tạo tài khoản và lưu vào DB.
    - Gửi phản hồi ngay cho người dùng (giao tiếp **đồng bộ**).
    - Gửi một sự kiện (event) đến **event broker** (ví dụ: RabbitMQ).
3. **Message microservice** sẽ:
    - Nhận sự kiện từ hàng đợi trong event broker.
    - Gửi email/SMS cho người dùng.

    - Gửi lại một event phản hồi đến broker để báo **đã gửi thông báo thành công**.
4. **Accounts microservice** (đã đăng ký lắng nghe sự kiện này):
    - Nhận phản hồi.
    - Cập nhật DB rằng đã gửi thông báo cho người dùng (**giao tiếp bất đồng bộ**).

Lợi ích:

- Các microservice hoạt động **độc lập**, không bị phụ thuộc lẫn nhau.
- Hệ thống **ổn định hơn** khi một dịch vụ bị chậm hoặc ngừng hoạt động.
- Có thể mở rộng dễ dàng bằng cách thêm subscriber mới.

Công nghệ sử dụng:

- **Event broker:** RabbitMQ.
- **Mô hình:** Pub/Sub.
- **Các microservices:** accounts, message.

004

RabbitMQ là một **message broker mã nguồn mở**, tuân theo giao thức **AMQP (Advanced Message Queuing Protocol)**, cho phép giao tiếp bất đồng bộ giữa các ứng dụng.

Các khái niệm quan trọng được giới thiệu:

- **Producer (hoặc Publisher)**: dịch vụ gửi thông điệp.
- **Consumer (hoặc Subscriber)**: dịch vụ nhận và xử lý thông điệp.

- **Message Broker**: trung gian truyền tải thông điệp giữa producer và consumer.
- **Exchange và Queue**: mô hình hoạt động của AMQP. Producer gửi message đến exchange, exchange dựa vào rule để định tuyến tới queue, và từ đó consumer đọc message.

RabbitMQ từng không hỗ trợ phát lại (replay) sự kiện như **Apache Kafka**, nhưng hiện tại đã bổ sung khả năng này, dù Kafka vẫn phổ biến hơn trong các kịch bản stream dữ liệu.

Ngoài ra, giảng viên nhấn mạnh:

- Một broker có thể xử lý nhiều producer và consumer.
- Một consumer có thể nhận thông điệp từ nhiều queue, và một queue cũng có thể có nhiều consumer.
- RabbitMQ hỗ trợ đa ngôn ngữ (ví dụ: microservice có thể viết bằng Python, Java...).

005

Mục tiêu chính

- Xây dựng một **message microservice** để **nhận tin nhắn từ RabbitMQ** và **gửi SMS/email** đến người dùng cuối.
- **Không** sử dụng phương pháp truyền thống với @RestController, @GetMapping, @PostMapping, mà **sử dụng Spring Cloud Function** để viết business logic dưới dạng function.

Spring Cloud Function là gì?

- Một công cụ cho phép **tách riêng logic nghiệp vụ khỏi tầng hạ tầng**.
- Hỗ trợ ba loại function cơ bản của Java 8:

- **Supplier**: không nhận đầu vào, nhưng trả về đầu ra.
- **Function**: nhận đầu vào và trả về đầu ra (giống như bộ xử lý).
- **Consumer**: nhận đầu vào nhưng không trả về đầu ra.

Lợi ích khi dùng Spring Cloud Function

- **Đơn giản hóa** việc viết logic nghiệp vụ (chỉ dùng các hàm thuần túy).
- Dễ dàng **tái sử dụng** logic ở nhiều môi trường:
  - REST API.
  - Stream processors (Kafka, RabbitMQ…).
  - Serverless (AWS Lambda, Azure Functions...).
- Hỗ trợ viết theo **phong cách lập trình phản ứng (reactive)**, **tường minh (imperative)** hoặc kết hợp.
- Có thể **kết hợp nhiều function** lại để tạo logic phức tạp.
- **Triển khai linh hoạt**, chỉ cần thay đổi cấu hình YAML.

Tính ứng dụng cao

- Hỗ trợ tích hợp với các dịch vụ serverless lớn như **AWS Lambda**, **Azure**, **Apache OpenWhisk**.
- Business logic viết bằng function có thể:
  - Chạy dưới dạng REST API.
  - Chạy như một stream processor.
  - Đóng gói độc lập triển khai ở nhiều môi trường khác nhau.

Kết luận

Sử dụng Spring Cloud Function giúp:

- Giảm phụ thuộc vào framework cụ thể.
- Tăng khả năng mở rộng và tái sử dụng.
- Phù hợp tuyệt đối với **mô hình hướng sự kiện (event-driven)** như RabbitMQ hoặc Kafka.

006

Khởi tạo project tại start.spring.io:

- Chọn Maven, ngôn ngữ Java, Spring Boot version ổn định (ví dụ: 3.1.2).
- Group: com.eazybytes, Artifact và Name: message.
- Mô tả: Microservice hỗ trợ gửi tin nhắn cho hệ thống EazyBank.
- Packaging: Jar, Java version: 17.

Thêm dependency:

- Thêm Spring Cloud Function để triển khai business logic dạng hàm, có thể chạy độc lập hoặc trên các nền tảng serverless.

Tải và cấu hình dự án:

- Tải project về máy.
- Tạo folder section13 bằng cách copy từ section12, xóa thư mục .idea.
- Dán project message vào thư mục mới và mở bằng IntelliJ IDEA.

Tạo DTO (Data Transfer Object):

- Tạo record AccountsMessageDto trong package com.eazybytes.message.dto.

- Record chứa các trường: accountNumber, customerName, email, mobileNumber, dùng để nhận dữ liệu từ accounts microservice.

Tạo package xử lý logic:

- Tạo package com.eazybytes.message.functions.
- Tạo class MessageFunctions và thêm annotation @Configuration.
- Business logic sẽ được viết trong class này dưới dạng các function.

007

Tạo lớp MessageFunctions và khai báo biến logger để ghi log.

Xây dựng hàm email sử dụng Function<AccountsMessageDto, AccountsMessageDto>, trong đó:

- Dữ liệu đầu vào và đầu ra đều là AccountsMessageDto.
- Viết một lambda expression đơn giản để log thông tin và trả về dữ liệu đầu vào.
- Hàm này được đánh dấu bằng annotation @Bean.

Tạo hàm sms tương tự nhưng:

- Trả về long (số tài khoản) thay vì AccountsMessageDto.
- Dữ liệu đầu vào vẫn là AccountsMessageDto.
- Lấy giá trị accountNumber từ DTO và trả về.

Giải thích mục tiêu thiết kế:

- Hai hàm email và sms được tách riêng để minh họa việc *compose* (kết hợp) nhiều hàm trong Spring Cloud Function.
- Việc email trả về AccountsMessageDto cho phép hàm sms sử dụng làm đầu vào.

- Sau khi gửi email và SMS, hệ thống trả về accountNumber để thông báo với microservice accounts rằng quá trình gửi thông tin đã hoàn tất.

Kết nối với Spring Cloud:

- Hai hàm được giám sát bởi Spring Cloud Function, sử dụng trong môi trường xử lý bất đồng bộ qua message broker như RabbitMQ.

Lưu ý bổ sung:

- Giảng viên không đi sâu vào việc thực hiện logic gửi email/SMS thật để giữ trọng tâm vào Spring Cloud Function.
- Người học cần hiểu rõ về lambda expression và functional interface trong Java 8.

008

Giới thiệu lý do dùng Spring Cloud Function:

- Thay vì viết REST API truyền thống, ta chỉ cần viết các **hàm (functions)** chứa logic nghiệp vụ.
- Bằng cách thêm spring-cloud-starter-function-web vào pom.xml, các function này tự động được expose thành REST API.

Triển khai ví dụ thực tế:

- Tạo ứng dụng MessageApplication với 2 function chính: email và sms.
- Mỗi function nhận vào đối tượng JSON chứa thông tin tài khoản (account number, name, email, mobile...).
- Kiểm tra bằng Postman: Gửi yêu cầu đến /email hoặc /sms sẽ kích hoạt function tương ứng.

Debug và xác minh hoạt động:

- Đặt breakpoint trong function và kiểm tra log console để xác minh function thực thi đúng.
- Kết quả trả về tương ứng với dữ liệu đầu vào (hoặc một phần của nó).

Tạo REST API kết hợp nhiều function:

- Sử dụng application.yml để cấu hình port mới (9010) và khai báo:
- Việc này giúp **compose** 2 function thành một logic thống nhất, có thể gọi qua REST API /emailsms.

Kiểm tra API kết hợp:

- Gửi request tới /emailsms cũng chạy được lần lượt cả email và sms, và vẫn đảm bảo hoạt động độc lập của từng function.

Ứng dụng mở rộng:

- Các function có thể tích hợp với **RabbitMQ, Kafka**, hoặc triển khai trên **AWS Lambda** để xây dựng **microservices hướng sự kiện** (event-driven microservices).

009

Bài giảng này giới thiệu **Spring Cloud Stream**, một framework giúp xây dựng các ứng dụng **event-driven** và **streaming** một cách dễ dàng và có khả năng mở rộng cao. Trước đó, người học đã được hướng dẫn sử dụng **Spring Cloud Functions** để viết logic nghiệp vụ dưới dạng hàm và có thể expose như REST API.

Spring Cloud Stream giải quyết vấn đề gì?

- Khi bạn muốn tích hợp các hàm này với **message brokers** như **RabbitMQ**, **Apache Kafka**, bạn cần Spring Cloud Stream.
- Framework này giúp tách biệt phần **logic nghiệp vụ** khỏi phần **kết nối hạ tầng** với message broker, giúp code dễ bảo trì và chuyển đổi giữa các hệ thống messaging mà không cần thay đổi code.

Lợi ích chính:

- Chỉ cần thêm dependency phù hợp (RabbitMQ, Kafka, Kinesis, v.v.) và thay đổi một vài cấu hình, không cần sửa đổi business logic.
- Hỗ trợ nhiều broker phổ biến: Kafka, RabbitMQ, Google Pub/Sub, AWS SQS/SNS, Azure Event Hubs, v.v.
- Giúp việc chuyển đổi từ RabbitMQ sang Kafka (hoặc ngược lại) diễn ra **seamless**, không cần rewrite lại toàn bộ mã nguồn.

3 thành phần chính của Spring Cloud Stream:

1. **Destination Binders:** Tạo ra kết nối với hệ thống messaging bên ngoài.
2. **Destination Bindings:** Tạo cầu nối giữa ứng dụng và hệ thống messaging, có thể là **input** (nhận message) hoặc **output** (gửi message).
3. **Message:** Định nghĩa cấu trúc dữ liệu được gửi/nhận giữa producer và consumer.

Kết luận: Spring Cloud Stream đơn giản hóa quá trình tích hợp messaging vào microservices, giảm gánh nặng cho developer, và giúp hệ thống dễ thích nghi với sự thay đổi nền tảng.

010

1. Message Microservice

- **Bước đầu**: Dừng microservice và cập nhật pom.xml.
    - **Xóa**: các dependency cũ như spring-cloud-function.
    - **Thêm mới**:
        - spring-cloud-stream

        - spring-cloud-stream-binder-rabbit (dùng RabbitMQ làm message broker)
        - Các dependency test (spring-boot-starter-test, stream-test-binder)
    - **Lý do xóa spring-cloud-function**: vì spring-cloud-stream đã tự động bao gồm nó.
- **Thêm Google Jib plugin** để chuẩn bị build Docker image.
- **Cập nhật application.yml**:
    - Cấu hình spring.cloud.stream.bindings.emailsms-in-0:
        - destination: send-communication (RabbitMQ queue)
        - group: dùng tên app để giữ tên queue cố định.
    - Cấu hình RabbitMQ:
        - Host: localhost, Port: 5672, User/Pass: guest

2. Accounts Microservice

- **Cập nhật pom.xml**:
    - Thêm các dependency giống như message service (stream, binder-rabbit)
- **Tạo DTO class**:
    - AccountsMessageDto: chứa các thông tin account như account number, name, email, mobile.
- **Mục tiêu**:
    - Khi tạo tài khoản mới, dịch vụ accounts sẽ **gửi một sự kiện** (event) đến RabbitMQ, để message service xử lý qua queue send-communication.

011

1. **Khởi tạo RabbitMQ:**
    - Chạy lệnh Docker để khởi động RabbitMQ.
    - Kiểm tra RabbitMQ container đang chạy trên Docker Desktop.
2. **Khởi động hệ thống microservices:**
    - Lần lượt khởi động các services: ConfigServer, EurekaServer, AccountsMicroservice, MessageMicroservice, GatewayServer.
    - Khởi động thêm Keycloak trong Docker làm máy chủ xác thực.
3. **Truy cập giao diện quản trị RabbitMQ:**
    - Đăng nhập với tài khoản mặc định guest/guest tại localhost:15672.
    - Kiểm tra exchange send-communication và queue send-communication.message đã được tạo đúng từ cấu hình trong application.yml.
4. **Cấu hình và giải thích binding trong Spring Cloud Stream:**
    - Spring tự động tạo input/output binding từ function emailsms.
    - Output binding (emailsms-out-0) chưa được kết nối với queue nào.
5. **Gửi yêu cầu tạo tài khoản (API) qua Postman:**
    - Sử dụng Client Credentials để lấy access token từ Keycloak.
    - Gửi API tạo tài khoản từ AccountsMicroservice.
6. **Kết quả:**
    - Message được gửi đến RabbitMQ khi tài khoản được tạo.
    - Có thể thấy spike trong giao diện RabbitMQ console.

- Log trong AccountsApplication xác nhận message đã được gửi thành công.
- Message được nhận trong MessageMicroservice.

012

1. Tình trạng hiện tại

- Hệ thống mới chỉ xử lý việc gửi sự kiện từ accounts đến message.
- Mục tiêu là thiết lập chiều ngược lại: từ message đến accounts.

2. Cấu hình gửi sự kiện từ message microservice

- Thêm **output binding** trong application.yml:
- Spring Cloud sẽ tự động tạo exchange communication-sent trong RabbitMQ.
- Không cần viết thêm code gửi thủ công nhờ Spring Cloud Functions—hệ thống tự hiểu dữ liệu trả về từ function sms (trả về accountNumber) là output.

3. Cấu hình nhận sự kiện tại accounts microservice

- Tạo class AccountsFunctions trong package com.eazybytes.accounts.functions.
- Định nghĩa một function dạng Consumer<Long> tên là updateCommunication, nhận accountNumber và cập nhật trạng thái giao tiếp trong database.

4. Cập nhật schema và entity

- Thêm cột mới communication_switch kiểu BOOLEAN trong bảng accounts.
- Cập nhật entity AccountsEntity để ánh xạ trường communicationSwitch vào DB.

5. Tạo phương thức cập nhật trạng thái giao tiếp

- Thêm phương thức mới updateCommunicationStatus(Long accountNumber) vào interface IAccountService.

- Cài đặt logic trong AccountServiceImpl.

6. Lợi ích của Spring Cloud Functions

- Giảm đáng kể việc viết code thủ công.
- Dễ dàng mở rộng, thay đổi hệ thống mà không cần viết lại nhiều logic như trong các phương pháp truyền thống.
- Dễ dàng di chuyển giữa các sản phẩm hoặc môi trường khác nhau.

013

Bài giảng trình bày một **demo đầy đủ** về giao tiếp bất đồng bộ giữa hai microservice: accounts và message, sử dụng **RabbitMQ**.

Các ứng dụng liên quan (accounts, message, gateway server) đã được **khởi động lại** sau khi có nhiều thay đổi.

Người học được nhắc **cũng nên khởi động lại** các ứng dụng để áp dụng các thay đổi mới nhất.

Truy cập RabbitMQ console để kiểm tra:

- Có **hai exchanges chính**: send communication và communication sent.
- Có một exchange cũ không dùng nữa (emailsms-out-zero) do thiếu cấu hình trước đó.

Trong phần queues:

- Tồn tại queue mới: communication-sent.accounts, nơi accounts microservice sẽ chờ nhận message từ message microservice.

Thực hiện test toàn bộ flow bằng Postman:

- Gửi request tạo tài khoản mới (số điện thoại kết thúc bằng 687).
- Sau khi gửi, tài khoản được tạo thành công (HTTP 201).
- Tuy nhiên, message chưa được gửi vì đang bị dừng tại breakpoint trong message microservice.
- Truy cập H2 console (port 8080) để xác nhận trạng thái communication hiện là null.
- Sau khi **release breakpoint**, message được gửi đi và accounts microservice nhận sự kiện để cập nhật trạng thái communication = true.
- Điều này chứng minh hệ thống **hoạt động bất đồng bộ thành công**.

Cuối cùng:

- Cấu hình thêm khả năng **event streaming** cho hệ thống EasyBank.
- Nhấn mạnh tầm quan trọng của event streaming trong việc **giảm sự phụ thuộc thời gian** giữa các microservice.
- Toàn bộ các bước được lưu trong slide để người học tham khảo lại.
- Bước tiếp theo: **test hệ thống trong Docker network**, tạo Docker images, đẩy lên Docker Hub và cập nhật file docker-compose.yml.

014

Tạo Docker images có tag s13, bao gồm cả microservice mới tên là message.

Đẩy các images lên Docker Hub để người khác có thể sử dụng.

Cập nhật Docker Compose file:

Thêm service rabbit với các cấu hình image, hostname, port, health check, và mạng.

Cập nhật tag từ s12 thành s13 cho các microservice như config server, Eureka, accounts, loans, cards, gateway.

Cập nhật accounts và message service để phụ thuộc rabbit, cấu hình biến môi trường SPRING_RABBITMQ_HOST.

Khởi động lại toàn bộ container bằng docker compose up -d sau khi dừng các container cũ, bao gồm cả Keycloak và RabbitMQ.

Truy cập RabbitMQ UI, xác minh có hai exchanges và hai queues như mong đợi.

Cấu hình lại Keycloak:

Tạo client EazyBankCallCenter-cc với grant type là client credentials.

Tạo role accounts, gán role cho client.

Kiểm thử bằng Postman:

Lấy access token thành công.

Gửi yêu cầu tạo tài khoản mới → phản hồi thành công.

Kiểm tra logs:

Trong message microservice: ghi log gửi email và SMS.

Trong accounts: log gửi và cập nhật trạng thái gửi thành công.

Cuối cùng, toàn bộ mã nguồn đã được đẩy lên GitHub tại thư mục section_13.