**Projeto Microservice**

**Tecnologias utilizadas:**

Linguagem: Java 17.

Gerenciador de dependências: Maven.

Ecossistema/Framework: Spring Framework.

- Spring Boot: Iniciar os dois Microservices de negócio.

- Spring Web: Para criar os endpoints.

- Spring Data JPA: Responsável por fazer a modelagem e as ligações com as base de dados.

- Spring Validation: Para fazermos validações iniciais na entrada da API.

- Spring AMQP: Para trabalharmos com o protocolo de mensageria, realizar essa comunicação assíncrona entre eles.

- Spring Mail: Para enviarmos o e-mail para os usuários que forem se cadastrar na plataforma.

Banco de dados (Base de dados por microservices): PostgreSQL.

Broker: RabbitMQ.

Cloud: Cloud AMQP.

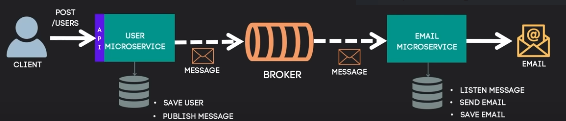
SMTP: SMTP Gmail.

Para colocarmos o nosso conhecimento teórico na prática, vamos implementar dois microserviços de negócio (User Microservice e Email Microservice).

Mas mesmo os microserviços sendo independentes entre si e possuir bases de dados por microserviço, elas vão precisar se comunicar de alguma forma, para isso nós vamos implementar também todo um fluxo de comunicação assíncrona entre eles, lembrando que tem vários outros modelos e tipos de comunicações entre microserviços, e quando vamos utilizar essa comunicação assíncrona não bloqueante via mensageria, também temos diversas maneiras, utilizando de comandos, eventos, mas neste caso vamos utilizar comandos, onde o ”User Microservice” vai produzir uma mensagem, enviado para outro microservice que ele faça uma determinada ação.

Ou seja, o fluxo do RabbitMQ vai ser o seguinte:

- Um cliente vai enviar um POST para cadastrar um usuário no sistema, salvando na sua base de dados, e logo após vai publicar uma mensagem com a intenção de um comando para um canal de mensagens (Broker), para realizar então essa comunicação assíncrona e não bloqueante. Vamos iniciar o ”Email Microservice”, que vai estar esperando essas mensagens serem enviadas para que elas então possam ser consumidas, ou seja, ele também vai estar conectado ao canal de mensagens, e quando essas mensagens forem chegando no Broker, ele vai fazer o respectivo roteamento para o “Email Microservice” consumir essas mensagens e realizar então a ação que o “User Microservice” está enviando. Logo após o “Email Microservice” consumir essa mensagem, ele vai enviar um email de boas vindas para o usuário que acabou de ser cadastrado na plataforma e ele vai salvar o email para que a gente tenha esses dados salvos na arquitetura.



Para implementarmos e fazemos o Broker, nós vamos utilizar o RabbitMQ.

Mas antes disso vamos ver como funciona o fluxo do RabbitMQ.

Já que o “User Microservice” vai gerar/produzir uma mensagem, ele vai ser denominado “Producer”, e vai enviar para o Broker. O RabbitMQ ele é formato pelas estruturas: Exchange e Queues, que variam muito de Broker para Broker.  
Ou seja, o Producer envia/produz uma mensagem para o Broker e quem recebe essa mensagem vai ser o Exchange, responsável por analisar a mensagem e todas as suas informações, e a partir disso vai fazer o devido roteamento para as respectivas Queues, dessa forma essa mensagem pode ser enviada para uma fila, para mais ou nenhuma, dependendo do tipo de Exchange que vamos utilizar.

Após isso, o “Email Microservice” vai estar conectado a essa fila e vai sempre consumir essas mensagens, por isso é denominado “Consumer”.

Vamos utilizar do CloudAMQP para realizar toda a configuração e monitoramento do nosso RabbitMQ/Broker em Cloud.

**Criação dos microserviços com Spring Boot**

**- User**

Java 17, Spring Boot 3.3.2, Jar, Maven.

**Dependências utilizadas**: Spring Web, Spring Data JPA, PostgreSQL Driver, Validation, Spring for RabbitMQ.

**Base de dados no PostgreSQL**: ms-user

**Configuração e conexão com a base de dados:**

**OBS: Vamos utilizar bases de dados por microservices**

server.port=8081  
  
spring.datasource.url= jdbc:postgresql://localhost:5432/ms-user  
spring.datasource.username=postgres  
spring.datasource.password=victormacedo  
spring.jpa.hibernate.ddl-auto=update

- Definir qual vai ser a porta que esse microserviço vai estar disponível, para que não haja conflitos, pois se não definirmos, ele vai levar em consideração a porta default 8080.

- Em relação ao Spring DataSource, é para realizarmos a conexão com a base de dados, passando sua URL, o username do admin e a senha para conexão.

- E o **ddl-auto** é para que todos os mapeamentos que a gente for fazer, eles sejam convertidos em tabelas e colunas, tanto para a criação quanto para a remoção dos mesmos, isso reflita automaticamente na base de dados.

Podemos notar que não fizemos nenhuma configuração em relação ao nosso Servlet Container (Tomcat), pois quando utilizamos do Spring Boot, ele já traz embutido um Servlet Container embutido, o que facilita a o Start dessa aplicação.

**Fazemos a mesma coisa com o nosso microserviço Email e sua base de dados ms-email.  
OBS: As configurações continuam as mesmas, mudando somente sua base de dados, a server port e implementando mais uma dependência (Java Mail Sender, para realizar toda a conexão do SMTP do Gmail).**

server.port=8082  
  
spring.datasource.url= jdbc:postgresql://localhost:5432/ms-email  
spring.datasource.username=postgres  
spring.datasource.password=victormacedo  
spring.jpa.hibernate.ddl-auto=update