**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.

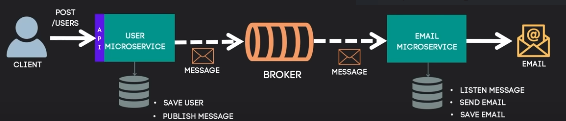
SMTP: SMTP Gmail.

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.

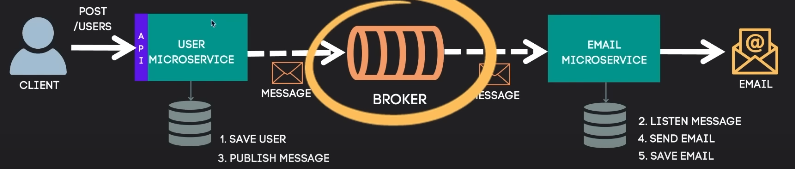
- E o **ddl-auto** é para que todos os mapeamentos que a gente for fazer, eles sejam convertidos em tabelas e colunas, tanto para a criação quanto para a remoção dos mesmos, isso reflita automaticamente na base de dados.

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

Implementando o endpoint **“save users”**



Explicando o fluxo do microserviços que vamos fazer até agora:

Um cliente ele vai enviar um POST para uma API no User Microservice, ele vai receber esse usuário e vai salvar na base de dados.

Para implementarmos isso nós fazemos os seguintes passos:

**- Criamos o package Models e iniciamos a classe UserModel.**

Declaramos que ele é uma Entidade e que a sua tabela se chama “TB\_USERS”, aonde vai ser armazenado todos os válidos usuários cadastrados.

Passamos seus atributos, como: ID, nome e email. O ID vai ser do tipo UUID, vai ser gerado automaticamente e vai ser a chave primária da entidade.

Geramos também o Getters e Setters de cada atributo.

**- Criamos o package Repositories e iniciamos a interface UserRepository.**

Ele nós ajuda a utilizarmos e acessarmos todas as facilidade e métodos que o Spring Data JPA nos oferece, passando somente a Entidade/Model que vamos utilizar e o identificador da entidade.

public interface UserRepository extends JpaRepository<UserModel, UUID> {}

**- Criamos o package Controllers e iniciamos a classe UserController.**

É aonde nós vamos criar o endpoint para que o cliente envie um POST passando um novo usuário e esse microservice vai receber o usuário vai salvar na sua base de dados.

O nosso UserController vai ser um Bean do Spring do tipo **@RestController**, para mostrar ao Spring que ele vai ser um Bean gerenciado por ele.

Iniciamos o nosso POST. O cliente vai acessar para enviar um novo usuário para ser cadastrado.

@PostMapping("/users")  
public ResponseEntity<UserModel> saveUser(@RequestBody @Valid UserRecordDto userRecordDto){  
 var userModel = new UserModel();  
 BeanUtils.*copyProperties*(userRecordDto, userModel);  
 return ResponseEntity.*status*(HttpStatus.*CREATED*).body(userService.save(userModel));  
}

- Utilizamos a notation **@PostMapping** com a URI **“/users”**.

- O método **saveUser** é do tipo ResponseEntity, onde vamos retornar um UserModel contendo todos os atributos.

Como parâmetros, ele vai receber no corpo da requisição os dados que vierem do userRecordDto, que vai ser um JSON do POST. E para garantir as validações feitas no Record, é utilizado essa notation **@Valid** no Controller.

- Recebemos um user da classe UserRecordDto e iniciamos um objeto da classe UserModel.

Como esse userRecordDto é um JSON, seus atributos precisam se transformar em atributos de objeto Java, ou seja, nós copiamos as propriedades dele e passamos para esse objeto da classe UserModel que criamos logo acima, assim nós conseguimos manipular essas informações.

Ou seja, quem vai ser salvo na base de dados vai ser nosso UserModel e não JSON que vem do POST, porém pegamos essas informações/propriedades que vem do POST e transformamos em objeto Java, em um userModel por exemplo, e utilizamos ela para subirmos na base de dados.

Para utilizarmos nossos métodos feitos no Service, vamos criar um ponto de injeção para ele.

final UserService userService;  
  
public UserController(UserService userService) {  
 this.userService = userService;  
}

**- Criamos o package dtos e iniciamos o UserRecordDto do tipo Record.**

Aqui nós vamos mapear os atributos que recebemos do nosso POST, como nós queremos receber ele.

Para nós fazermos o mapeamento desses atributos que recebemos do POST, no Record todos os atributos vão estar presentes como parâmetros. Incluimos também a validação.

public record UserRecordDto(@NotBlank String name,  
 @NotBlank @Email String email) {  
  
}

- Aqui nós vamos receber o name só se ele não estiver vazio/nulo, recebemos também só o e-mail se ele não estiver vazio/nulo e validado como um e-mail. Essas notations são da dependência Validation.

Para pegarmos esse userModel e salvarmos na base de dados, acessando o repositório que já criamos.

**- Criamos o package Services e iniciamos a classe UserService.**

Ele vai também vai ser um Bean do Spring do tipo **@Service.**

Vamos conectar o Service ao nosso Repository, que tem acesso a toda a estrutura do JPA. Para isso, nós vamos um ponto de injeção userRepository dentro do Service.

Podemos fazer das seguintes formas:

- Utilizando o **@Autowired** e iniciamos o userRepository ou via construtor, que foi o que eu utilizei e que a IDE que utilizei sugere também.

final UserRepository userRepository;  
  
public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
}

**Após isso, nós criamos o NOSSO método save(), para que ele seja utilizado no nosso controller.**

@Transactional  
public UserModel save(UserModel userModel) {  
 return userRepository.save(userModel);  
}

- Ele vai ter como retorno um UserModel, por isso que ele é desse tipo. Ele vai receber o userModel e vai retornar já utilizando o userRepository com o save do JPA passando o userModel.  
Aqui nós utilizamos o save do JPA, mas podemos utilizar outros métodos prontos para nos auxiliar na implementação.

Ou seja, o save() que utilizamos quando vamos salvar o userModel que era um userRecordDto, nós utilizamos o NOSSO save() do Service.

return ResponseEntity.*status*(HttpStatus.*CREATED*).body(userService.save(userModel));

Por enquanto nosso save() vai fazer somente isso, porém posteriormente ele vai publicar uma mensagem no nosso canal de mensagens via comandos para o Broker. Por isso vamos deixar a notation **@Transactional**, para garantirmos o rollback, pois se algum das operações falha ou dá algum tipo de erro, ele realiza o rollback e tudo volta ao normal.

**Criando conexão com rabbitMQ na CloudMQP e configurações**