Projeto Python / Java Spring-Boot utilizando RabbitMQ

a) TEORIA:

O RabbitMQ é um sistema de mensagens amplamente usado que atua como um intermediário de mensagens para enviar e receber dados entre aplicações.

1. Arquitetura Básica do RabbitMQ

- **Broker de Mensagens:** O RabbitMQ funciona como um broker de mensagens, que recebe mensagens de produtores e as encaminha para consumidores. Ele atua como um intermediário entre diferentes partes de um sistema que precisam se comunicar.

2. Conceitos Fundamentais

a. Produtores e Consumidores

- **Produtor:** Uma aplicação ou componente que envia mensagens para o RabbitMQ. No nosso exemplo uma aplicação **Python**
- **Consumidor:** Uma aplicação ou componente que recebe mensagens do RabbitMQ. No nosso exemplo uma aplicação **Java Spring-Boot**

b. Exchanges

- **Exchange:** Um componente do RabbitMQ que roteia mensagens para uma ou mais filas com base em regras de roteamento. Existem diferentes tipos de exchanges:
    - **Direct Exchange:** Roteia mensagens para filas com base em uma chave de roteamento exata.
    - **Topic Exchange:** Roteia mensagens para filas com base em padrões de chave de roteamento.
    - **Fanout Exchange:** Roteia mensagens para todas as filas associadas, sem considerar a chave de roteamento.
    - **Headers Exchange:** Roteia mensagens com base nos cabeçalhos de mensagem em vez da chave de roteamento.

c. Filas

- **Fila:** Um buffer de mensagens armazenadas que aguardam para serem processadas pelos consumidores. As filas são onde as mensagens são armazenadas até serem consumidas.

d. Bindings

- **Binding:** A conexão entre uma exchange e uma fila. Define como e para onde as mensagens devem ser roteadas.

e. Routing

- **Routing:** O processo pelo qual uma exchange determina para qual(s) fila(s) as mensagens devem ser enviadas com base na chave de roteamento e nas regras definidas.

3. Fluxo de Mensagens

1. **Produtor Envia Mensagem:**
   - O produtor **(Ex Python)** envia uma mensagem para uma exchange no RabbitMQ.
2. **Exchange Roteia Mensagem:**
   - A exchange roteia a mensagem para uma ou mais filas com base nas regras de roteamento e nos bindings definidos.
3. **Fila Armazena Mensagem:**
   - A mensagem é armazenada na fila até que um consumidor esteja pronto para processá-la.
4. **Consumidor Recebe Mensagem:**
   - O consumidor (**Ex Java Spring-Boot** se conecta à fila, recupera a mensagem e a processa.
5. **Confirmação e Exclusão:**
   - Após o processamento, o consumidor pode confirmar que a mensagem foi recebida e processada com sucesso. O RabbitMQ então remove a mensagem da fila.

4. Características e Funcionalidades

- **Persistência:** Mensagens podem ser configuradas para serem persistentes, o que significa que serão salvas em disco e não serão perdidas se o RabbitMQ for reiniciado.
- **Transações:** RabbitMQ suporta transações, permitindo que mensagens sejam enviadas e confirmadas de forma atômica.
- **Confirmação de Mensagens:** O RabbitMQ pode ser configurado para garantir que as mensagens sejam confirmadas após o processamento, garantindo que não sejam perdidas.
- **Escalabilidade:** RabbitMQ pode ser escalado horizontalmente, distribuindo a carga de trabalho entre vários nós.
- **Gerenciamento e Monitoramento:** O RabbitMQ oferece uma interface de gerenciamento web que permite monitorar o estado do broker, verificar filas, exchanges e muito mais.

b) PRÁTICA: Python e RabbitMQ

Estrutura do Projeto Python no VSCode: (Criar pasta python-producer)

python-producer/
├── producer.py
└── requirements.txt

1) ARQUIVO requirements.txt

pika

2) ARQUIVO producer.py

No terminal execute: pip install pika (referente a biblioteca RabbitMQ)

```
import pika
import time

# Conectar ao RabbitMQ
connection =
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declarar a fila
channel.queue_declare(queue='hello')

for i in range(1, 11):
    message = f"Mensagem{i}"
    channel.basic_publish(exchange='',
                          routing_key='hello',
                          body=message)
    print(f" [x] Sent '{message}'")
    time.sleep(10)  # Espera 10 segundos

connection.close()
```

- pika: É uma biblioteca Python para interação com o RabbitMQ.

- time: Módulo padrão do Python que fornece várias funções relacionadas ao tempo, incluindo sleep, que é usada para pausar a execução do programa por um número especificado de segundos.
- def send_message(channel, message): Define uma função chamada send_message que aceita dois parâmetros: channel: O canal através do qual a mensagem será enviada; message: A mensagem a ser enviada.
- channel.basic_publish(exchange='', routing_key='hello', body=message): Publica a mensagem no RabbitMQ.
    - exchange='': Especifica o exchange padrão.
    - routing_key='hello': A chave de roteamento é o nome da fila (hello) onde a mensagem será enviada.
    - body=message: O corpo da mensagem a ser enviada.
- print(f" [x] Sent '{message}'"): Imprime uma mensagem no console indicando que a mensagem foi enviada.
- def main(): Define a função principal do script.
- connection=pika.BlockingConnection(pika.ConnectionParameters(host='localhost')): Estabelece uma conexão com o RabbitMQ.

- pika.BlockingConnection(pika.ConnectionParameters(host='localhost')): Conecta ao RabbitMQ que está rodando no host localhost.
- channel = connection.channel(): Cria um canal através do qual as operações de mensagem serão realizadas.
- channel.queue_declare(queue='hello'): Declara uma fila chamada hello. Se a fila não existir, ela será criada.
- for i in range(10): Loop que itera 10 vezes para enviar 10 mensagens.
    - message = f'mensagem{i + 1}'": Cria uma mensagem chamada mensagem1, mensagem2, etc.
    - send_message(channel, message): Chama a função send_message para enviar a mensagem.
    - time.sleep(10): Pausa a execução por 10 segundos antes de enviar a próxima mensagem.
- connection.close(): Fecha a conexão com o RabbitMQ.
- if __name__ == "__main__":: Verifica se o script está sendo executado diretamente (não importado como um módulo).
- main(): Chama a função main para iniciar o processo de envio de mensagens.

O script producer.py é responsável por enviar mensagens para uma fila RabbitMQ chamada hello. Ele se conecta ao RabbitMQ, declara a fila se ela não existir, e envia 10 mensagens (uma a cada 10 segundos). Cada mensagem é enviada através do canal estabelecido e uma mensagem de confirmação é impressa no console. Ao final do envio das mensagens, a conexão com o RabbitMQ é fechada.

<u>OBS</u>: Não execute ainda o projeto Python, pois como não temos o rabbitMQ instalado localmente, executaremos um container Docker após o projeto estar concluído. Assim utilizaremos o mesmo container para o projeto Python e Java Spring-Boot.

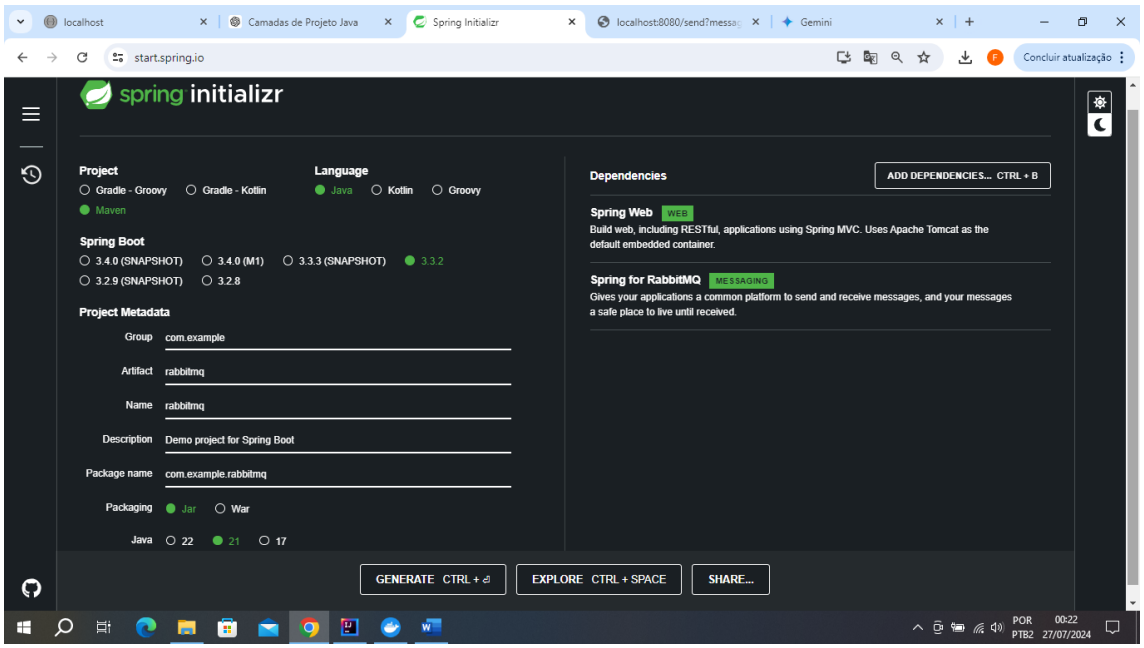
c) PRÁTICA: Spring-Boot e RabbitMQ

Estrutura do Projeto Java Spring-Boot no Intellij (Criar pasta java-consumer)

java-consumer/

```
├── docker-compose.yml/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           └── rabbitmq/
│   │   │               ├── RabbitmqApplication.java
│   │   │               ├── RabbitmqConfig.java
│   │   │               ├── Consumer.java
│   │   │               └── MessageController.java
│   │   └── resources/
│   │       └── application.properties
└── pom.xml
```

3) SPRING INITIALIZR



Começamos baixando o nosso arquivo spring boot. Veja na foto que usaremos: maven, java, spring-boot 3.3.2, Artificat: java-rabbitMQ, jar e java21.

Como dependências: Spring web e Spring for RabbitMQ

4) EXTRAIR O ARQUIVO E ABRIR NA IDEA Intellij

5) DOCKER – java-rabbitMQ

Como não tenho instalado o RabbitMQ, utilizarei um container Docker, e para isso devemos criar um arquivo: docker-compose.yml dentro do projeto Java Spring-Boot (dentro da pasta principal do projeto)

docker-compose.yml

```yaml
version: '3'
services:
  rabbitmq:
    image: "rabbitmq:3-management"
    ports:
      - "5672:5672"
      - "15672:15672"
```

OBS: Não é necessário colocar nada específico no arquivo application.properties do projeto Java para se conectar ao RabbitMQ rodando em um container Docker, contanto que o RabbitMQ esteja configurado para escutar na porta padrão (5672) e seu container esteja corretamente mapeado para a máquina local. No entanto, você pode precisar ajustar o host se o RabbitMQ estiver rodando em uma rede diferente ou se precisar configurar autenticação.

Aqui está um exemplo simplificado do que você pode precisar:

application.properties

```
spring.application.name=java-rabbitMQ

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Feito isso, já podemos rodar o Docker criado no terminal através do comando:

docker-compose up

6) JAVA – SPRING BOOT

Já temos criada a classe main JavaRabbitMqApplication, e precisaremos criar mais 3(três) classes para este exemplo: JavaRabbitMQConfig, Consumer e MessageController.

JavaRabbitMqApplication (já criada pelo Spring-Boot)

```
package com.example.java_rabbitMQ;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class JavaRabbitMqApplication {

  public static void main(String[] args) {
   SpringApplication.run(JavaRabbitMqApplication.class, args);
  }

}
```

JavaRabbitMQConfig.java

```
package com.example.java_rabbitMQ;

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class JavaRabbitMQConfig {

  public static final String QUEUE_NAME = "hello";

  @Bean
  public Queue helloQueue() {
    return new Queue(QUEUE_NAME, false);
  }
}
```

A classe JavaRabbitMQConfig é usada para configurar o RabbitMQ em uma aplicação Spring Boot. Ela define como a aplicação se conecta ao broker RabbitMQ, como as filas, exchanges e bindings são configurados, e outras propriedades necessárias para o funcionamento do RabbitMQ.

A classe RabbitmqConfig é uma configuração Spring que define um bean para uma fila do RabbitMQ. A anotação @Configuration marca a classe como uma fonte de definições de beans, e o método helloQueue() é anotado com @Bean, indicando que ele cria um bean gerenciado pelo Spring. Este bean é uma fila do RabbitMQ com o nome definido pela constante QUEUE_NAME. O parâmetro false passado ao construtor da fila indica que a fila não é durável.

Em resumo, essa configuração define uma fila chamada "hello" que pode ser usada para comunicação entre produtores e consumidores no RabbitMQ.

Consumer.java

```java
package com.example.java_rabbitMQ;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
public class Consumer {

  private final List<String> messages = new ArrayList<>();

  @RabbitListener(queues = JavaRabbitMQConfig.QUEUE_NAME)
  public void receiveMessage(String message) {
    System.out.println(" [x] Received '" + message + "'");
    messages.add(message);
  }

  public List<String> getMessages() {
    return messages;
  }
}
```

A classe Consumer é um componente de serviço no Spring Boot que consome mensagens da fila RabbitMQ. Aqui está o que acontece em detalhes:

Importa as classes necessárias:

- RabbitListener: Annotation usada para marcar métodos que devem ser invocados quando uma mensagem é recebida de uma fila RabbitMQ.
- Service: Indica que a classe é um serviço Spring, um tipo de componente que contém a lógica de negócio.
- ArrayList e List: Utilizadas para armazenar as mensagens recebidas.
- @RabbitListener(queues = RabbitmqConfig.QUEUE_NAME): Esta anotação configura o método receiveMessage como um ouvinte de mensagens da fila especificada (RabbitmqConfig.QUEUE_NAME).

- RabbitmqConfig.QUEUE_NAME é uma constante que contém o nome da fila. No caso deste exemplo, essa constante está definida na classe RabbitmqConfig como "hello".

public void receiveMessage(String message): Método que será chamado toda vez que uma mensagem for recebida na fila especificada.

- O parâmetro String message é a mensagem recebida.

- Dentro do método, a mensagem recebida é impressa no console e adicionada à lista messages.

A classe Consumer é um componente de serviço Spring que atua como um consumidor de mensagens RabbitMQ. Quando uma mensagem é recebida da fila especificada, o método receiveMessage é invocado, adicionando a mensagem a uma lista interna e imprimindo-a no console. O método getMessages permite que outros componentes do aplicativo acessem as mensagens armazenadas.

MessageController.java

```java
package com.example.java_rabbitMQ;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class MessageController {

  @Autowired
  private Consumer consumer;

  @GetMapping("/messages")
  public List<String> getMessages() {
    return consumer.getMessages();
  }
}
```

Importa as classes necessárias:

- Autowired: Annotation usada para injetar automaticamente dependências.
- GetMapping: Annotation usada para mapear solicitações HTTP GET para métodos de manipulador específicos.
- RequestParam: Annotation usada para extrair parâmetros de consulta dos pedidos HTTP.
- RestController: Annotation que marca a classe como um controlador onde cada método retorna um objeto de domínio em vez de uma visão.
- List: Interface utilizada para definir uma lista de mensagens.
- @Autowired: Esta anotação permite que o Spring injete automaticamente uma instância do bean Consumer na classe MessageController. O Spring cuidará da criação e gerenciamento da instância do Consumer.
- private Consumer consumer: Declara uma variável consumer do tipo Consumer que será injetada pelo Spring.
- @GetMapping("/messages"): Esta anotação mapeia solicitações HTTP GET para o método getMessages. Quando uma solicitação GET é feita para o endpoint /messages, este método será invocado.

- public List<String> getMessages(): Declara um método público chamado getMessages que retorna uma lista de strings (List<String>).
- return consumer.getMessages(): O método chama getMessages() na instância consumer injetada e retorna a lista de mensagens armazenadas no Consumer.

A classe MessageController é um controlador REST do Spring que expõe um endpoint HTTP GET (/messages). Quando esse endpoint é acessado, o método getMessages é chamado, que por sua vez, retorna a lista de mensagens recebidas pelo Consumer. A dependência Consumer é injetada automaticamente pelo Spring usando a anotação @Autowired. Este controlador permite que você visualize as mensagens que foram consumidas da fila RabbitMQ.

7) Teste da Aplicação

Certifique-se que o container Docker está rodando (java-rabbitmq). Se não tiver, rode o container Docker : **docker-compose up (ctrl + c → para o container STOP)**

Execute o arquivo producer.py no VSCode (Run Python File) – Veja no próprio teminal do VSCode as mensagens sendo enviadas para a fila do RabbitMQ

Execute o arquivo JavaRabbitMQApplication no terminal do Intellij (main) – Veja no próprio terminal do intellij as mensagens consumidas pela aplicação Java do RabbitMQ.

Outras formas de visualização das mensagens consumidas pela aplicação Java Spring-Boot:

1)Abra a seguinte pagina no google chrome (ou outro de sua preferencia):

http://localhost:8080/messages

2)Teste via Postman

No postman, new → http

GET http://localhost:8080/messages

SEND

Voce deverá ver no response (body) uma lista com todas as mensagens consumidas pela aplicação Java Spring-Boot

3)Usando o RabbitMQ Management Interface

http://localhost:15672

username: guest

password: guest

Aqui você poderá visualizar dashboards, queues(filas), criar filas (add queue), criar Exchange (add new Exchange)

Verificar Mensagens em uma Fila:

- Vá para a aba **"Queues"** e clique na fila de interesse.
- Na página da fila, você verá uma seção para **"Get messages"**. Aqui, você pode clicar em **"Get messages"** para visualizar as mensagens na fila.

Adicionar Mensagens Manualmente:

- Na página da fila, você pode adicionar mensagens manualmente usando o formulário de **"Publish Message"**.
- Preencha os campos necessários e clique em **"Publish Message"** para enviar uma mensagem para a fila.

No projeto colocamos o arquivo Docker-compose.yml dentro do projeto java e rodamos no intellij. Mas este também poderia ter ficado isolado, já que implementa tanto o projeto Python como o Java SpringBoot. Se quisermos alterar, a estrutura do projeto ficaria assim.

/rabbitmq-docker

├── docker-compose.yml

/python-producer

├── producer.py

├── requirements.txt

/java-consumer

├── src

│ ├── main

│ │ ├── java

│ │ │ └── com

```
│   │   │       └── example
│   │   │           └── rabbitmq
│   │   │               ├── RabbitmqApplication.java
│   │   │               ├── RabbitmqConfig.java
│   │   │               ├── Consumer.java
│   │   │               └── MessageController.java
│   │   └── resources
│   │       └── application.properties
├── pom.xml
```

E bastaria abrir o terminal na pasta do arquivo docker-compose.yml e executar o comando docker-compose up.