Desafio da disciplina: Sistema de Pedido Online para Restaurante

Itens

1. Problema a resolver 1
2. Objetivo 1
3. Requisitos para a entrega/avaliação 1
   3.1. Como será feita a avaliação: 1
4. Requisitos Funcionais 2
   4.1. Gerenciamento de usuários: 2
   4.2. Criar Pedido 2
   4.3. Consultas 2
   4.4. Processamento de Pagamento 2
   4.5. Pagamento Pendente 2
   4.6. Reprocessamento Automático 2
   4.7. Atualização Automática de Status 2
5. Requisitos Não Funcionais 3
   5.1. Arquitetura em Múltiplos Serviços 3
   5.2. Segurança com Spring Security + JWT 3
   5.3. Comunicação Assíncrona com Kafka 3
   5.4. Resiliência (Resilience4j) 3
   5.5. Boas Práticas de Arquitetura 3

Boas-vindas ao **Tech Challenge da fase 3**! Este desafio é fundamental para consolidar os conhecimentos obtidos ao longo da fase. Nosso desafio focará em arquitetura distribuída, segurança de aplicações e comunicação entre serviços, de forma síncrona e assíncrona.

Seguindo com a proposta dos projetos anteriores, este projeto agora envolve a criação de uma funcionalidade de criação de pedidos online para o restaurante, e processar os pagamentos

1. Problema a resolver

Você deve implementar um *conjunto de serviços* em **Java**, usando **Spring Boot ou Quarkus**, para suportar o fluxo completo de pedidos online de um restaurante, incluindo criação de pedidos, comunicação com serviço de pagamento externo, controle de status e tratamento de falhas com resiliência.

2. Objetivo

O objetivo é criar um sistema de pedidos online para um restaurante, que permita criar pedidos, processar pagamentos e controlar o status dos pedidos. O sistema deve ser resiliente e seguro, capaz de lidar com falhas e timeouts. Na descrição de cada um dos requisitos há informações adicionais relativas a cada um dos itens.

3. Requisitos para a entrega/avaliação

- Aplicação funcionando adequadamente com todos os serviços implementados e funcionando corretamente;
- Arquivo para testes dos endpoints (bruno, postman, insomnia, etc.)
- Arquivo `compose.yml` para subir, em um único comando, **todos** os serviços necessários para o funcionamento do sistema. *Haverá um repositório com o arquivo inicial.*
- Documentação que detalhe:
  - O desenho da arquitetura, usando pelo menos um destes: diagrama de componentes, diagrama de sequência ou C4 model.
  - Descrição do fluxo principal de funcionamento.
  - Identificação clara dos pontos de resiliência.
- Repositório com o código-fonte de todos os componentes que foram usados ou construídos.
- Video, **de no máximo 10 minutos** apresentando:
  - **todas** as funcionalidades solicitadas;
  - a arquitetura escolhida, e porque foi escolhida;

3.1. Como será feita a avaliação:

- Análise da documentação, se comunica corretamente tudo o que foi solicitado;
- Análise do vídeo, se comunica corretamente tudo o que foi solicitado;
- Revisão do código e teste de endpoints.

4. Requisitos Funcionais

4.1. Gerenciamento de usuários:

- Criar cliente
- Autenticar cliente

4.2. Criar Pedido

O sistema deve permitir criar um pedido contendo:

- Dados do cliente (ID extraído do token JWT).
- Dados do restaurante.
- Lista de itens, contendo:
  - id do produto
  - nome
  - quantidade
  - preço

Com o pedido recebido, o serviço irá efetuar o cálculo e retornar para o cliente com um ID de pedido e o valor total, e pedir para confirmar o pedido.

Importante! Não é necessário criar CRUD para os itens de pedido. Pode existir uma lista prévida de itens que podem ser parte de um pedido.

4.3. Consultas

- Consultar pedido por ID.
- Consultar todos os pedidos associados ao cliente autenticado.

4.4. Processamento de Pagamento

- Existira um serviço **processamento-pagamento-externo** fornecido pelos professores como uma imagem, e que implementa um serviço *eventualmente disponível* que deverá ser considerado como um serviço externo, que faz o processamento de pagamentos. Ao criar um pedido, este deve chamar um serviço que se comunica com o **processamento-pagamento-externo**.
- Essa API sempre devolverá pagamento autorizado **quando estiver disponível**.

4.5. Pagamento Pendente

Se o **pagamento-service** estiver indisponível (falha, timeout, circuito aberto):

- O pedido **não deve falhar**.
- O pedido deve ser marcado como **PENDENTE_PAGAMENTO**.
- O pedido deve ser enviado para uma fila de pendências.

4.6. Reprocessamento Automático

Quando o **pagamento-service** voltar a funcionar:

- Os pedidos pendentes devem ser reprocessados automaticamente.
- Caso o pagamento seja confirmado, o status deve ser atualizado para **PAGO**.

4.7. Atualização Automática de Status

Após confirmação de pagamento:

- O pedido deve ser atualizado automaticamente.
- (Opcional) Seguir fluxo para serviços de produção ou notificação.

5. Requisitos Não Funcionais

5.1. Arquitetura em Múltiplos Serviços

A arquitetura deve estar separada em mais de um serviço, e deve conter no mínimo:

- serviços de autenticação (ver item abaixo)
- `pedido-service`
- `pagamento-service`, que faz a comunicação com o serviço externo de pagamento.
- (Opcional) `restaurante-service`, que recebe um aviso de que o pedido foi confirmado e começa a preparar o pedido
- (Opcional) `api-gateway`

Inclua um diagrama de componentes, de sequência ou C4 model.

5.2. Segurança com Spring Security + JWT

- Endpoint de login gerando token JWT.
- Perfis de acesso (cliente, admin).
- Endpoints de pedido protegidos com token obrigatório.
- O ID do cliente deve vir do token.

5.3. Comunicação Assíncrona com Kafka

Eventos obrigatórios:

- `pedido.criado`
- `pagamento.aprovado`
- `pagamento.pendente` (caso o serviço esteja indisponível)

Fluxo mínimo:

- `pedido-service` publica `pedido.criado`.
- `pagamento-service` consome o evento, e tenta processar usndo **processamento-pagamento-externo**
- Caso OK, atualizar o pedido conforme evento
- Caso Falha
  - publicar `pagamento.pendente`.
  - Worker consome pendências e reprocessa.
  - `pedido-service` atualiza status conforme eventos.

5.4. Resiliência (Resilience4j)

A chamada ao pagamento-service deve conter:

- Circuit Breaker
- Retry
- Timeout
- Fallback que:
  - Marca pedido como **PENDENTE_PAGAMENTO**
  - Envia para tópico Kafka `pagamento.pendente`

5.5. Boas Práticas de Arquitetura

Seguir camadas ou Clean/Hexagonal Architecture:

- controller
- service / use cases
- domain *(onde devem ficar as regras de negócio)*
- infra