Case Técnico - Sistema de Processamento de Transações

Cenário:

Uma empresa de e-commerce precisa de um sistema que processe transações em tempo real. Toda vez que uma compra é realizada, os dados da compra (ID do pedido, ID do cliente, valor da compra e status do pagamento) são enviados para um sistema de processamento em tempo real, que deve registrar essas transações em um banco de dados e realizar algumas operações de validação e notificação. O sistema precisa ser escalável, tolerante a falhas e fácil de implantar em diferentes ambientes.

Objetivo:

Desenvolver uma aplicação usando Java, Kafka, Docker e um banco de dados não-relacional que receba eventos de compra de um tópico Kafka, valide os dados e armazene as informações em um banco de dados.

Requisitos Funcionais:

1. Produção de eventos Kafka:

Criar um produtor Kafka em Java que simule a geração de eventos de compra. Cada evento deve conter o seguinte payload:

```
{
  "data": {
    "order": {
      "orderId": "a35e5447-cc4b-30ff-aa14-a02a82417374",
```

```
      "customerId": "bbb522d7-cc4b-30ff-aa14-a02a82410215",
      "amount": 100.00,
      "status": "PAID", "PENDING", "FAILED",
      "date": "yyyy-MM-ddThh:mm:ss"
    },
    "timestamp": "2024-09-19T12:44:33.513-03:00"
  }
}
```

Os eventos devem ser enviados para um tópico Kafka chamado "purchase-orders".

2. Consumo de eventos Kafka:

- Criar um consumidor Kafka em Java que consuma eventos do tópico "purchase-orders".
- O consumidor deve validar o campo amount (não pode ser negativo) e o campo status.
- Caso os dados sejam inválidos, o evento deve ser rejeitado e o erro registrado.

3. Persistência dos dados:

- Se os dados forem válidos, armazenar a transação em um banco de dados não-relacional (ex.: MongoDB, DynamoDB ou Cassandra).
- A estrutura do documento deve incluir os campos: orderId, customerId, amount, status, timestamp, além de qualquer outra informação relevante para a modelagem no contexto de um banco não-relacional.
- Justificar a escolha do banco não-relacional e explicar a modelagem dos dados, considerando a escalabilidade e as consultas frequentes.

4. Observabilidade:

- Expor no Prometheus métricas de tempo de consumo, tempo de execução total, tempo de persistência no banco de dados (Total, P99 e P50) e tempo de acknowledge.
- Exportar logs no formato JSON, contendo:
  - Header: Schema
  - Payload: Mensagem
  - Metadata:
    - Offset
    - Partition
    - Timestamp
    - TimestampType
    - Topic

5. Ambiente Docker:

- Criar um ambiente Docker contendo:
  - Um container com o Kafka (pode usar uma imagem pronta, como confluentinc/cp-kafka).
  - Um container para o banco de dados não-relacional (ex.: MongoDB, DynamoDB Local).
  - Um container para a aplicação Java que consome e processa os eventos.
- A aplicação deve ser capaz de se conectar ao Kafka e ao banco de dados dentro da rede Docker.

```
docker run --name kafka-container -d confluentinc/cp-kafka

docker run --name mongo-container -d mongo
```

6. Extras (opcional):

- Adicionar container para Splunk/Fluent Bit.
- Implementar um circuit breaker.
- Preparar o sistema para consumo com lag no tópico Kafka, criando um endpoint para inserção em

lote dos eventos, simulando o consumo com lag.

- Implementar retentativas automáticas no consumidor Kafka em caso de falha de conexão com o banco de dados.

- Implementar logs estruturados para facilitar a observabilidade.

- Caso o banco escolhido suporte, implementar TTL (Time to Live) para expirar registros antigos automaticamente.

Critérios de Avaliação:

1. Testes Unitários:

- Cobertura mínima de 80%.

- Extra: Teste Funcional e Teste Mutante.

- Teste de Performance (Performance4ALL).

2. Conhecimento em Docker:

- Configuração de múltiplos containers.

- Rede entre containers e persistência de dados.

3. Habilidade em Java:

- Implementação de produtor e consumidor Kafka.

- Manipulação de erros e exceções.

- Integração com banco de dados não-relacional usando drivers ou SDKs específicos.

4. Kafka:

- Configuração e uso de tópicos.

- Capacidade de lidar com grande volume de eventos.

- Configuração de retentativas e timeout no Kafka Consumer.

5. Banco de Dados Não-Relacional:

- Modelagem eficiente dos dados no contexto não-relacional.

- Justificativa da escolha do banco e da modelagem dos dados.

6. Boas Práticas:

- Código limpo e organizado.

- Implementação de testes unitários para validação dos componentes.

Entrega:

- Deve entregar o código em um repositório Git com um arquivo README.md explicando como rodar o sistema localmente, além de uma breve explicação sobre as decisões técnicas, especialmente na escolha do banco de dados não-relacional e modelagem dos dados.

- Entregar um desenho de solução (freeform).

- Entregar uma collection (Postman ou Insomnia) para POST e GET dos cenários descritos.

Referências:

- Splunk Docker Image: https://hub.docker.com/r/splunk/splunk/

- Grafana Docker Image: https://hub.docker.com/r/grafana/grafana