projeto debito credito

Diagram owner	Bruno Zapateiro Silva
Last date updated	Sep 10, 2024
On this page	

📖 Visão Geral

Este projeto implementa um sistema de transações financeiras distribuído usando dois microserviços: **account-service** e **transaction-service**. O sistema foi desenhado para processar transações financeiras de forma assíncrona e segura, utilizando tópicos para comunicação entre os serviços.

📐 Arquitetura

O sistema segue uma arquitetura de microserviços baseada em mensagens, onde os serviços se comunicam de maneira assíncrona por meio de tópicos Kafka de mensageria. Cada microserviço tem uma responsabilidade bem definida:

Account-Service

O **account-service** é responsável pelas seguintes funcionalidades:

- Criação de contas.
- Execução de débitos e créditos nas contas associadas a uma transação.
- Publicar num tópico Kafka informando o status da transação (sucesso ou falha).

Esse serviço possui uma base de dados PostgreSQL para persistir os dados da conta.

Transaction-Service

O **transaction-service** é responsável pelas seguintes funcionalidades:

- Recebimento de requisições de transações.
- Persistência das transações no banco de dados.
- Postagem em um tópico para o **account-service** processar as transações.
- Atualização do status da transação com base no retorno do **account-service**.
- Consulta o resultado das transações.

Esse serviço possui uma base de dados NoSQL com MongoDB, para persistir o histórico das transações.

Sequência

1. **Client** → **Transaction-Service** : Recebe requisição para transação. `POST /transaction/start`
2. **Transaction-Service** → **Tópico** : Postar transação no tópico Kafka de Transação e persistir na base de dados.
3. **Tópico** → **Account-Service** : Consumir transação no tópico Kafka de Transação e executa-la.
4. **Account-Service** → **Tópico** : Postar resultado da transação no tópico kafka de Resultado de Transação.
5. **Transaction-Service** → **Tópico** : Consumir status da transação no tópico e atualizar na base de dados.

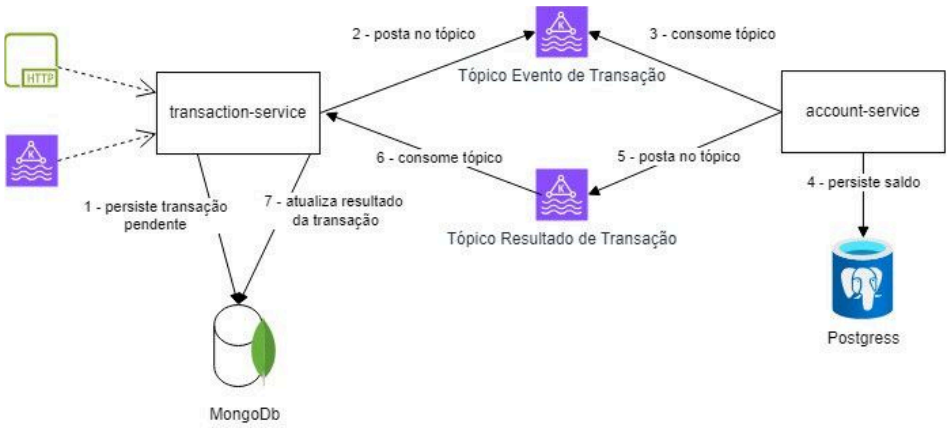
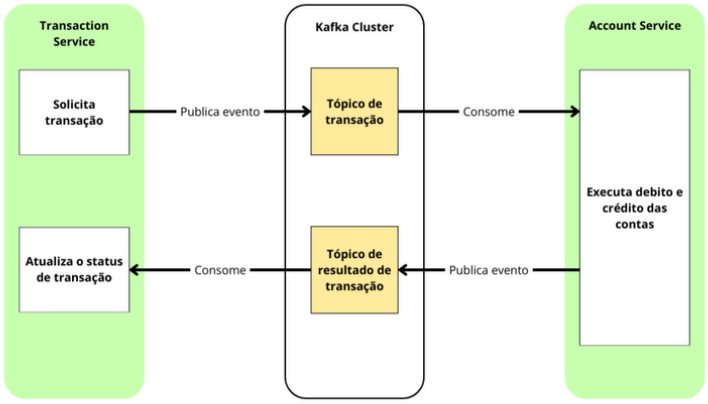


Diagrama de Arquitetura - Transação



Fluxo de Transação

🏦 Fluxo de Criação de Conta

1. O cliente faz uma requisição HTTP `POST /account/create` para o serviço, enviando os dados necessários para a criação da conta.
2. O **account-service** valida os dados fornecidos, como por exemplo uam verificação de maioridade.

3. Persiste a conta na base Postgress, com o saldo zero.
4. Em caso de sucesso, o serviço retorna um status `201 - Created` com os detalhes da nova conta, incluindo o ID da conta e o saldo inicial.

`POST /account/create`

```
{
  "name": "João",
  "cpf": "12345678900",
  "birthdate": "2000-05-15",
  "email": "email@example.com",
  "phone": "11912345678"
}
```

💰 Fluxo de Débito e Crédito

No **account-service**, o fluxo de débito e crédito é gerido por meio de bloqueios utilizando o PostgreSQL, especificamente a função `pg_try_advisory_lock`. Este método evita que a mesma conta seja modificada simultaneamente por múltiplas transações, garantindo a integridade dos dados

1. **Requisição de Transação**
   a. O **account-service** consome uma mensagem do tópico que indica uma nova transação (débito ou crédito).
2. **Identificação da Conta**
   a. O serviço identifica a conta envolvida na transação (conta de origem ou conta de destino) a partir dos IDs fornecidos na transação.
3. **Tentativa de Bloqueio da Conta (`pg_try_advisory_lock`)**
   a. O serviço tenta adquirir um lock na conta utilizando a função `pg_try_advisory_lock`, que tenta obter um bloqueio exclusivo no nível do banco de dado, A chave do bloqueio é derivada do hash do atributo `account_id`.
   b. Se o bloqueio for adquirido com sucesso, a transação continua.
   c. Se o bloqueio não for adquirido (porque a conta já está bloqueada por outra operação), o serviço pode retornar um erro e irá reprocessar a transação conforme a política de retentativas do consumidor do tópico de transações.
4. **Validação do Saldo (Para Débitos)**
   a. Se a transação for um débito, o serviço verifica se a conta de origem possui saldo suficiente. Se o saldo for insuficiente a transação falha e o serviço irá publicar um evento no tópico de resultado de transação, informando a falha e o motivo.
5. **Execução da Transação**
   a. Com o bloqueio em vigor, o serviço realizar o débito e o credito das respectivas contas.
6. **Finalização**
   a. Após a execução da transação, o serviço libera o bloqueio utilizando a função `pg_advisory_unlock`, e irá publicar um evento no tópico de resultado de transação, informando o sucesso da transação
   b. Se algum erro acontecer o serviço irá publicar um evento no tópico de resultado de transação, informando a falha e o motivo

Utilizar a técnica de `pg_try_advisory_lock` garante que apenas uma transação por vez possa modificar uma conta específica, proporcionando um controle robusto sobre o acesso concorrente e mantendo a integridade dos dados em cenários de alta concorrência.

❓ Decisões

PostgreSQL

Para o armazenamento dos dados das contas, eu escolhi o postgreSQL pela sua facilidade de integração com inúmeras bibliotecas, capacidade enorme de escalabilidade, e por suas funcionalidades avançadas de gerenciamento de transações, como por exemplo o

pg_try_advisory_lock, que foi utilizado no projeto.

MongoDB

Escolhi um banco de dados NoSQL para armazenar as transações, pensando na flexibilidade e na e na escalabilidade desse serviço, pois ele poderia receber como por exemplo uma consulta de extrato.
Essa base precisa de uma **alta performance de leitura**, e não necessita de um esquema fixo.

Apache Kafka

O Kafka foi escolhido porque permite que os serviços se comuniquem de forma assíncrona e desacoplada. Isso melhora a escalabilidade e flexibilidade da arquitetura. Ele é altamente escalável, **capaz de lidar com milhões de mensagens por segundo**, e isso é essencial para esse projeto.

Arquitetura dos Serviços

Para a arquitetura dos serviços, eu não segui ao pé da letra um pattern de Arquitetura Hexagonal ou Clean Architecture, porém apliquei o conceito de **isolar a camada de domínio** de qualquer dependência externa. Com o uso do DDD ou outros patterns orientados a eventos, a escalabilidade dessas aplicações fica muito mais simples.

🚩 Melhorias e Limitações

Circuit Breaker

Um ponto de melhoria é adicionar um **circuit braker** entre os serviços de transação e de conta, para que possa ser implementado tanto transações de assíncrona quanto síncrona, e ter também respostas mais rápidas em caso de instabilidade ou lentidão no serviço de conta. Com isso implementado, imagino uma tentativa de transação síncrona, e dependendo de qual é a situação do `account-service`, essa mesma transação poderia ser postada num tópico e passaria a acontecer de forma assíncrona.

Histórico de Saldos

Outro ponto muito importante para a aplicação, é uma tabela com todos os saldos anterior e após uma transação, para que em casos de inconsistência ou alguma grande falha no fluxo de uma operação de débito ou crédito, possamos fazer o rollback do saldo.