Desafio Técnico PagueVeloz: Sistema de Processamento de Transações Financeiras

Contexto da Empresa

A PagueVeloz é uma empresa de tecnologia voltada para o setor financeiro, especializada em soluções de meios de pagamento, serviços bancários integrados e adquirência. Nosso propósito é facilitar a vida financeira de empresas e empreendedores por meio de produtos robustos, ágeis e seguros. Com uma arquitetura orientada a microsserviços e foco em escalabilidade, operamos em alto volume de transações e buscamos talentos que compartilhem nossa paixão por desempenho, arquitetura limpa e excelência técnica.

Objetivo do Desafio

O objetivo deste exercício é implementar um sistema de processamento de transações financeiras que será o núcleo transacional de uma nova plataforma de adquirência. O sistema deve lidar com operações financeiras em um ambiente onde volume, concorrência e confiabilidade são críticos.

Como o Sistema Deve Funcionar

Entrada

Seu sistema vai receber comandos de operações financeiras através de uma interface de linha de comando (CLI) ou API REST em formato JSON. Cada operação contém os seguintes campos:

Campo	Significado
`operation`	Tipo de operação: `credit`, `debit`, `reserve`, `capture`, `reversal`, `transfer`
`account_id`	Identificador único da conta
`amount`	Valor da operação em centavos (inteiro)
`currency`	Moeda da operação (ex: "BRL")
`reference_id`	Identificador único da transação para idempotência
`metadata`	Dados adicionais opcionais

Exemplo de entrada:

```
{
  "operation": "credit",
  "account_id": "ACC-001",
  "amount": 10000,
  "currency": "BRL",
  "reference_id": "TXN-001",
```

```
  "metadata": {
    "description": "Depósito inicial"
  }
}
```

Saída

Para cada operação, o sistema deve retornar um objeto JSON com o resultado da transação:

Campo	Significado
`transaction_id`	Identificador único da transação processada
`status`	Status da operação: `success`, `failed`, `pending`
`balance`	Saldo atual da conta após a operação
`reserved_balance`	Saldo reservado da conta
`available_balance`	Saldo disponível para uso
`timestamp`	Data e hora da operação
`error_message`	Mensagem de erro (se aplicável)

Exemplo de saída:

```
{
  "transaction_id": "TXN-001-PROCESSED",
  "status": "success",
  "balance": 10000,
  "reserved_balance": 0,
  "available_balance": 10000,
  "timestamp": "2025-07-07T20:05:00Z",
  "error_message": null
}
```

Regras de Negócio

Contas e Clientes

1. **Múltiplas contas por cliente**: Cada cliente pode possuir N contas
2. **Estrutura da conta**: Cada conta possui:
    - Saldo disponível
    - Saldo reservado
    - Limite de crédito
    - Status da conta (`active`, `inactive`, `blocked`)
    - Histórico de transações

Tipos de Operações

1. **Crédito (`credit`)**: Adiciona valor ao saldo da conta
2. **Débito (`debit`)**: Remove valor do saldo da conta
3. **Reserva (`reserve`)**: Move valor do saldo disponível para o saldo reservado
4. **Captura (`capture`)**: Confirma uma reserva, removendo do saldo reservado
5. **Estorno (`reversal`)**: Reverte uma operação anterior
6. **Transferência (`transfer`)**: Move valor entre duas contas

Validações de Negócio

- Operações não podem deixar o saldo disponível negativo
- Limite de crédito deve ser respeitado
- Operações de débito consideram saldo disponível + limite de crédito
- Reservas só podem ser feitas com saldo disponível
- Capturas só podem ser feitas com saldo reservado suficiente

Controle de Concorrência

- Operações concorrentes na mesma conta devem ser bloqueadas
- Implementar locks otimistas ou pessimistas para garantir consistência
- Transações devem ser atômicas

Resiliência e Eventos

- Cada operação deve gerar eventos assíncronos
- Implementar retry com backoff exponencial para falhas
- Garantir idempotência através do `reference_id`
- Suporte a rollback em caso de falhas

Exemplos de Casos de Uso

Caso #1: Operações Básicas de Crédito e Débito

Cenário: Cliente com conta nova realizando operações simples

Operação	Valor	Saldo Resultante	Status	Explicação
`credit`	R$ 1000,00	R$ 1000,00	`success`	Depósito inicial
`debit`	R$ 200,00	R$ 800,00	`success`	Débito aprovado
`debit`	R$ 900,00	R$ 800,00	`failed`	Saldo insuficiente

Entrada:

```
[
  {"operation": "credit", "account_id": "ACC-001", "amount": 100000, "currency": "BRL", "
  {"operation": "debit", "account_id": "ACC-001", "amount": 20000, "currency": "BRL", "re
  {"operation": "debit", "account_id": "ACC-001", "amount": 90000, "currency": "BRL", "re
]
```

Saída:

```
[
  {"transaction_id": "TXN-001-PROCESSED", "status": "success", "balance": 100000, "availa
  {"transaction_id": "TXN-002-PROCESSED", "status": "success", "balance": 80000, "availab
  {"transaction_id": "TXN-003-PROCESSED", "status": "failed", "balance": 80000, "availabl
]
```

Caso #2: Operações com Limite de Crédito

Cenário: Cliente com limite de crédito de R$ 500,00

Operação	Valor	Saldo	Limite	Status	Explicação
`credit`	R$ 300,00	R$ 300,00	R$ 500,00	`success`	Depósito inicial
`debit`	R$ 600,00	-R$ 300,00	R$ 500,00	`success`	Usou limite de crédito
`debit`	R$ 300,00	-R$ 300,00	R$ 500,00	`failed`	Excedeu limite

Entrada:

```
[
  {"operation": "credit", "account_id": "ACC-002", "amount": 30000, "currency": "BRL", "a
  {"operation": "debit", "account_id": "ACC-002", "amount": 60000, "currency": "BRL", "re
  {"operation": "debit", "account_id": "ACC-002", "amount": 30000, "currency": "BRL", "re
]
```

Caso #3: Reserva e Captura

Cenário: Operação de reserva seguida de captura

Operação	Valor	Saldo Disponível	Saldo Reservado	Status	Explicação
`credit`	R$ 1000,00	R$ 1000,00	R$ 0,00	`success`	Depósito inicial
`reserve`	R$ 300,00	R$ 700,00	R$ 300,00	`success`	Reserva para pagamento
`capture`	R$ 300,00	R$ 700,00	R$ 0,00	`success`	Captura da reserva

Caso #4: Transferência Entre Contas

Cenário: Transferência entre duas contas do mesmo cliente

Operação	Conta Origem	Conta Destino	Valor	Status	Explicação
`transfer`	ACC-001	ACC-002	R$ 500,00	`success`	Transferência aprovada

Caso #5: Operações com Falha e Retry

Cenário: Simulação de falha na publicação de eventos

Operação	Tentativa	Status	Explicação
`credit`	1	`failed`	Falha na publicação do evento
`credit`	2	`failed`	Retry com backoff
`credit`	3	`success`	Sucesso após retry

Requisitos Técnicos

Linguagem e Framework

- **C# .NET 9** obrigatório
- Uso eficiente de **async/await**
- Aplicação de princípios **SOLID** e **OOP**

Arquitetura

- **Clean Architecture**, **DDD** ou **Onion Architecture**
- Código estruturado para facilitar divisão futura em microsserviços
- Separação clara de responsabilidades
- Inversão de dependências

Persistência

- Modelagem relacional adequada
- Suporte a transações distribuídas
- Controle de concorrência eficiente

Resiliência

- Implementar retry com backoff exponencial
- Fallback strategies para falhas
- Circuit breaker pattern (diferencial)
- Idempotência garantida

Processamento Assíncrono

- Publicação de eventos assíncronos
- Processamento de comandos em background
- Consistência eventual

Observabilidade

- Logs estruturados
- Métricas de performance
- Rastreamento de transações
- Health checks

Critérios de Avaliação

Qualidades Valorizadas

1. **Simplicidade**: Projeto pequeno e de fácil entendimento
2. **Elegância**: Facilidade de manutenção e estrutura bem organizada
3. **Operacional**: Resolução completa do problema e capacidade de extensão

Aspectos Técnicos Avaliados

- Uso adequado de transparência referencial
- Qualidade dos testes unitários e de integração
- Documentação adequada
- Tratamento de concorrência
- Modelagem de domínio
- Estratégias de teste
- Cobertura de testes consistente

Diferenciais

- Uso de **Docker** para containerização
- Métricas de performance implementadas
- Observabilidade e eventos de negócio
- Deploy em nuvem ou container
- Implementação de padrões avançados (CQRS, Event Sourcing)

Entregáveis

Obrigatórios

1. **Projeto público no Git** com histórico de commits
2. **README detalhado** com:
   - Explicação das decisões técnicas e arquiteturais
   - Justificativa para uso de frameworks/bibliotecas
   - Instruções completas de compilação e execução
   - Instruções para execução dos testes
   - Exemplos de uso da API
3. **Código C# .NET 9** com cobertura de testes
4. **Testes unitários e de integração** abrangentes
5. **Documentação da API** (OpenAPI/Swagger)

Opcionais

- **Docker Compose** para execução do ambiente
- **Scripts de deployment**
- **Monitoring e métricas**
- **Performance benchmarks**

Instruções de Execução

Pré-requisitos

- .NET 9 SDK instalado
- SQL Server ou PostgreSQL (ou banco em memória para testes)
- Docker (opcional, mas recomendado)

Exemplo de Execução

```
# Compilar o projeto
dotnet build

# Executar os testes
dotnet test

# Executar a aplicação
dotnet run --project src/PagueVeloz.TransactionProcessor

# Executar via Docker
docker-compose up
```

Exemplo de Uso da API

```
# Criar uma conta
curl -X POST http://localhost:5000/api/accounts \
  -H "Content-Type: application/json" \
  -d '{"client_id": "CLI-001", "initial_balance": 0, "credit_limit": 50000}'

# Realizar um crédito
curl -X POST http://localhost:5000/api/transactions \
  -H "Content-Type: application/json" \
  -d '{"operation": "credit", "account_id": "ACC-001", "amount": 100000, "currency": "BRL
```

Considerações Importantes

Controle de Concorrência

- Implementar locks adequados para operações na mesma conta
- Considerar deadlocks e timeouts
- Otimizar para alta concorrência

Tratamento de Erros

- Não assumir que todas as operações serão bem-sucedidas
- Implementar logging detalhado de erros
- Retornar códigos de erro apropriados

Segurança

- Validação rigorosa de entrada
- Prevenção contra ataques de injeção
- Auditoria completa de todas as operações

Performance

- Otimização de consultas ao banco
- Uso eficiente de memória
- Processamento assíncrono quando apropriado

Notas Finais

Este desafio simula um ambiente real de alta disponibilidade e volume de transações. Esperamos que você demonstre não apenas conhecimento técnico, mas também capacidade de tomar decisões arquiteturais apropriadas para um sistema financeiro crítico.

O código será avaliado por profissionais experientes da PagueVeloz, então certifique-se de que sua solução seja robusta, bem documentada e pronta para produção.

Importante: Remoção de Informações Pessoais

⚠ **IMPORTANTE**: Por favor, remova toda informação que possa identificá-lo nos arquivos do desafio antes de enviar a solução. Atenção especial para:

- Arquivos da solução como código, testes, namespaces, binários, comentários e nomes dos arquivos
- Comentários automáticos que seu editor de código pode ter adicionado
- Documentação do código como annotations, metadata e README.md
- Informações de autoria do código e configuração do versionador de código

Se você planeja utilizar git como sistema de controle de versões, execute o seguinte comando na raiz do repositório para exportar a solução anonimizada:

```
git archive --format=zip --output=./pagueveloz-challenge.zip HEAD
```

Boa sorte com o desafio!