Solution Architecture

and Software Design

Techinical Challenge

by Flavio dos santos

Sumário

1. Introdução

1.1. Objetivo do Documento

Este documento tem como objetivo apresentar a proposta de arquitetura elaborada para atender ao desafio técnico de desenvolvimento de uma solução distribuída baseada em microserviços. Através deste material, são descritos as decisões arquiteturais adotadas, tecnologias selecionadas, requisitos funcionais e não funcionais, além de estratégias de segurança, resiliência e implantação.

O conteúdo aqui reunido também serve como apoio para avaliação técnica da solução, destacando seus diferenciais e oportunidades de evolução.

1.2. Contexto do Desafio

O desafio técnico propõe a criação de uma plataforma de controle financeiro voltada para comerciantes, permitindo o registro de **lançamentos** e a **consolidação** de saldos diários. A solução deve ser composta por microserviços independentes, expostos por meio de uma **API Gateway**, e estruturada de forma a garantir modularidade, escalabilidade e segurança.

A arquitetura deve ainda considerar boas práticas de separação de responsabilidade, uso de containers (**Docker**), e ser viável para execução local via **WSL2**, simulando um ambiente realista de produção.

1.3. Escopo da Solução Proposta

A solução contempla o desenvolvimento de três principais microserviços: **API Gateway**, **Serviço de Lançamento** e **Serviço de Consolidação**. Cada serviço é hospedado de forma isolada, comunica-se por meio de API REST e persiste seus dados em bancos **PostgreSQL** independentes. O sistema oferece um ponto de entrada único via **Gateway** e expõe uma interface web para integração do usuário.

Foi projetada com foco na simplicidade, mas preparada para suportar evolução, com balanceamento de carga, cache com **Redis** e observabilidade. O escopo cobre a estrutura básica de implantação, comunicação e persistência dos dados, priorizando clareza arquitetural e aderência aos requisitos do desafio.

1.4. Repositório do Projeto

O código-fonte completo da solução está disponível publicamente no GitHub:

Repositório: https://github.com/flavio-santos-ti/Challenge.SolutionArchitecture

O repositório contém:

- Estrutura organizada da solução com separação por microserviços;
- Scripts de inicialização (**Docker** e **SQL**);
- Arquitetura baseada em .NET 5 e PostgreSQL;
- Documentação complementar no README.md
- Automatização com shell scripts (`up-all.sh`, `down-all.sh`, etc).

O avaliador poderá:

- Clonar o projeto para execução local;
- Validar a arquitetura e o funcionamento da proposta;
- Acompanhar eventuais melhorias ou atualizações.

1.5. Origem e Contexto do Desafio

Este documento e a solução arquitetura proposta foram elaborados em resposta ao desafio técnico descrito no arquivo **DESAFIO ARQ DE SOLUCOES.pdf**, que estabeleceu os requisitos funcionais e não funcionais para o desenvolvimento do sistema.

O documento original do desafio está disponível no diretório **pdf** do repositório do projeto, garantindo total rastreabilidade entre os requisitos solicitados e as decisões arquiteturais aqui documentadas.

2. Requisitos do Negócio

2.1. Requisitos Fornecidos no Case

O desafio propõe a criação de uma solução para controle financeiro de um pequeno comerciante, com dois serviços principais:

- Serviço de Lançamentos: Registro de transações(créditos e débitos) diários.
- Serviço de Consolidação: Calcula a e persiste o saldo consolidado do dia.

Os requisitos obrigatórios são:

- Mapeamento de domínios e capacidade de negócio;
- Identificação de requisitos funcionais e não funcionais;
- Definição da arquitetura alvo;
- Documentação das decisões técnicas;
- Projeto publicado no GitHub com instruções de execução e testes.

Os requisitos diferenciais são:

- Arquitetura de transição (caso legado);
- Estimativas de custo;
- Estratégias de monitoramento, observabilidade e segurança.

Observação crítica:

O serviço de lançamentos deve operar independentemente do serviço de consolidação. Este último deve suporta 50 requisições/segundo com no máximo 5% de perda.

2.2. Requisitos Funcionais

O sistema deverá atender aos seguintes requisitos funcionais:

- **Registrar transações financeiras**: O microserviço de lançamentos deverá permitir o cadastro de transações com data, tipo (débito ou crédito) e valor.
- **Consultar transações por data**: Deverá ser possível obter todas as transações cadastradas em uma data específica por meio do microserviço de lançamentos.
- **Gerar saldo diário**: O microserviço de consolidação deverá calcular o saldo diário com base nas transações lançadas para determinada data, separando valores de crédito e débito.

- **Listar saldos consolidados**: O microserviço de consolidação deverá permitir a consulta de saldos diários consolidados por data.
- **Roteamento via API Gateway**: Toda comunicação com os microserviços deverá ser feita por meio de uma API Gateway implementado com YARP.
- **Retorno estrutura**: Todas as respostas dos microserviços deverão seguir um padrão uniforme com os campos: `isSuccess`, `message`, `statusCode` e `data`.

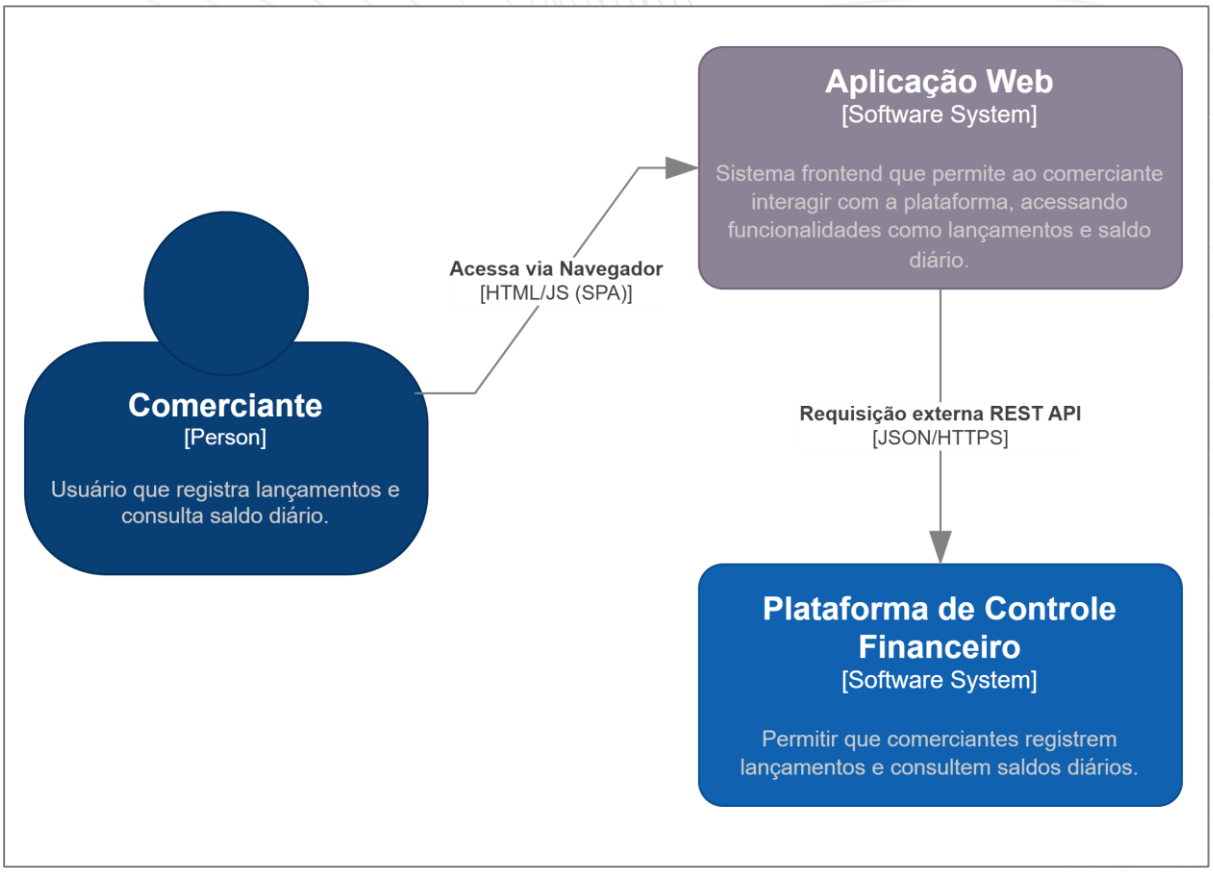
2.3. Requisitos Não Funcionais

O sistema deverá atender aos seguintes requisitos não funcionais:

- **Escalabilidade**: A arquitetura em microserviços permitirá o escalonamento individual das APIs de lançamentos, consolidação e gateway, conforme a demanda.
- **Manutenibilidade**: A estrutura modular e a separação de responsabilidade facilitarão a manutenção e evolução do sistema.
- **Portabilidade**: A Aplicação deverá ser executada em ambiente conteinerizado, com uso de Docker, viabilizando a replicação em diferentes ambientes.
- **Disponibilidade**: Os microserviços deverão permanecer acessíveis enquanto seus containers estiverem ativos, respeitando as boas práticas de inicialização e saúde.
- **Tempo de resposta**: As APIs deverão responder em tempo razoável (preferencialmente inferior a 500ms em consultas diretas), garantindo fluidez no uso.
- **Padrão de retorno**: As respostas deverão seguir um contrato unificado, facilitando a tratativa no frontend ou por sistemas terceiros.
- **Ambiente isolado**: O ambiente de execução deverá funcionar dentro de uma rede Docker customizada (Challenge-net) para simular isolamento e comunicação entre serviços.
- **Padronização**: O projeto deverá seguir convenções consistentes de nomenclatura, versionamento e estrutura de arquivos.

3. Arquitetura de Solução (C4 – Nível 1)

3.1. Diagrama – Contexto do Sistema



3.2. Atores e Sistemas Externos

O principal ator do sistema é o **Comerciante**, responsável por registrar os lançamentos financeiros diários e consultar os saldos consolidados. Essa interação é feita por meio da **Aplicação Web**, que serve como sistema frontend (SPA – Single Page Application), acessado via navegador.

A **Aplicação Web** é um sistema externo que se comunica com a **Plataforma de Controle Financeiro** por meio de requisições REST (JOSON/HTTPS). Esse modelo permite desacoplar a interface do núcleo da solução, viabilizando diferentes canais de interação no futuro.

O sistema foi projetado para permitir integrações futuras com terceiros, como serviços de autenticação, contabilidade ou plataformas financeiras, embora atualmente não dependa dessas integrações para operar.

3.3. Decisões Arquiteturais Estratégicas

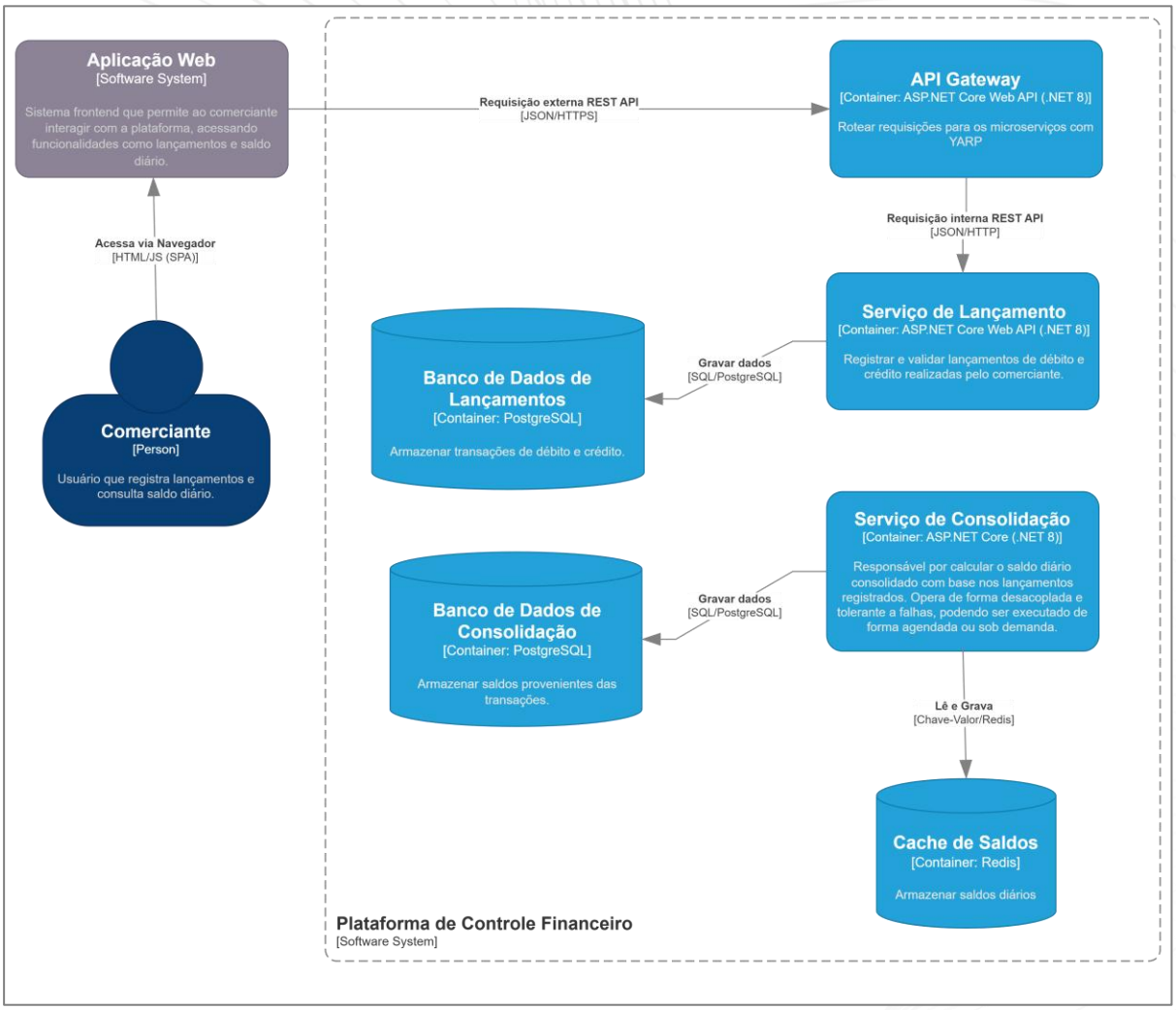
A arquitetura foi baseada na decomposição por **microserviços independentes**, promovendo isolamento de responsabilidades e escalabilidade. O **API Gateway** centraliza o roteamento das requisições, padronizando a comunicação entre os sistemas externos e os microserviços internos.

Cada serviço é executado em container isolado, facilitando o provisionamento via **Docker** e a orquestração por meio de scripts automatizados. A escolha por SPA no frontend visa oferecer melhor experiência ao usuário final e separação clara entre interface e lógica de negócio.

Além disso, a arquitetura suporta cenários evolutivos, como autenticação via OAuth, Logging distribuído e integração com mensageria, favorecendo manutenibilidade e extensibilidade no longo prazo.

4. Arquitetura de Software (C4 – Nível 2)

4.1. Diagrama – Visão de Containers



4.2. Atores e Sistemas Externos

O ator principal do sistema é o **Comerciante**, que utiliza a plataforma para registrar transações financeiras (débitos e créditos) e consultar o saldo diário consolidado. Essa interação ocorre por meio de um navegador web acessando uma **Aplicação Web** construída como uma **SPA** (Single Page Application).

Essa aplicação representa um sistema externo que consome a plataforma de forma desacoplada, permitindo futura substituição ou evolução do frontend sem impacto direto na lógica de negócios. A comunicação entre a aplicação web e a plataforma é realizada através de chamadas **REST via HTTPS**, garantindo segurança no transporte das informações.

4.3. Tecnologias Utilizadas

A arquitetura do sistema é baseada em contêineres **Docker** orquestrados manualmente para fins demonstrativos. A solução faz uso das seguintes tecnologias.

- **ASP.NET Core 8.0**: Desenvolvimento dos microserviços com suporte a API REST.
- **PostgreSQL**: Armazenamento relacional das transações e saldos consolidados.
- **Redis**: Cache de saldos para otimizar leituras frequentes e reduzir a carga sobre o banco de dados.
- **YARP (Yet Another Reverse Proxy)**: Utilizado no API Gateway para retear requisições de forma eficiente para os microserviços internos. O **YARP** pode ser configurado para aplicar políticas de roteamento e balanceamento de carga, como round-robin, least request ou stick sessions.

Essa Stack garante escalabilidade horizontal, performance e separação clara de responsabilidades

4.4. Comunicações entre os Containers

A comunicação entre os contêineres ocorre por meio de uma **rede Docker** customizada (Challenge-net), garantindo isolamento de rede e resolução de nomes por **DNS** interno. O **API Gateway** centraliza todas as requisições provenientes da **Aplicação Web** e as distribui para os respectivos microserviços:

- As chamadas relacionadas a transações são roteadas para o **Serviço de Lançamentos**.
- As chamadas relacionadas à consolidação de saldos são direcionadas ao **Serviço de Consolidação**.

A comunicação entre os serviços ocorre via **HTTP (REST/JSON)**, e os dados são persistidos em bancos **PostgreSQL** distintos por contexto.

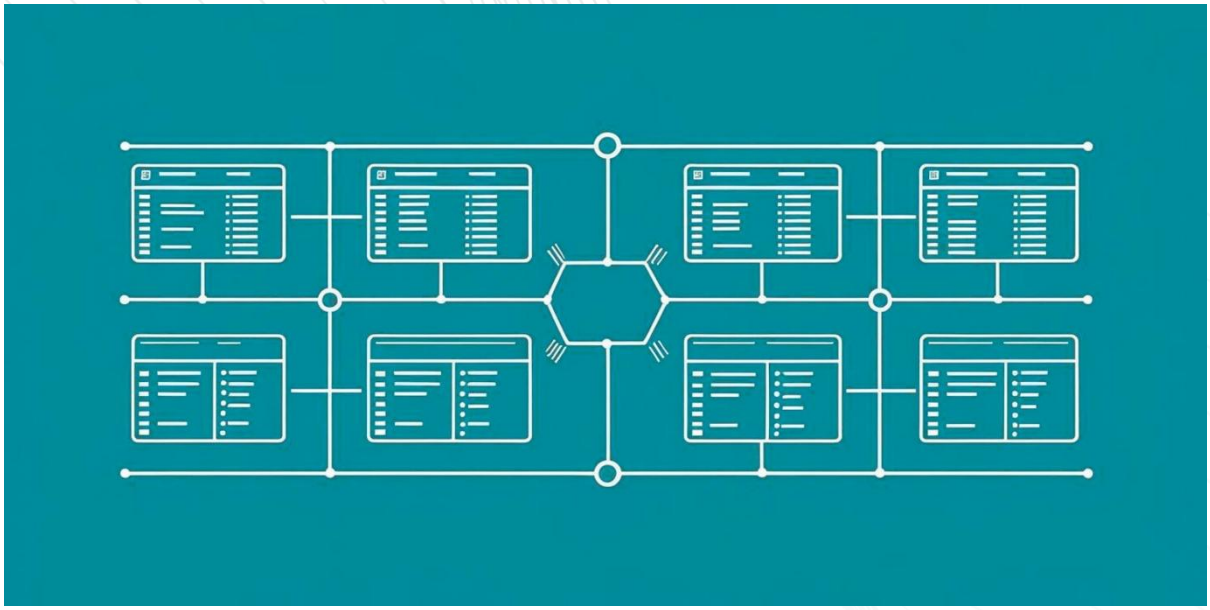
A comunicação entre os serviços ocorre via **HTTP (REST/JSON)**, e os dados são persistidos em bancos **PostgreSQL** distintos por contexto. O **Serviço de Consolidação**, além de persistir dados em seu banco próprio, também interage com o Redis para leitura e gravação de saldos, otimizando as respostas para o Comerciante.

O uso do JWT permite que o API Gateway valide o token antes de encaminhar as requisições, reforçando a segurança sem acoplar autenticação diretamente nos microserviços.

5. Modelo de dados

5.1. Visão Geral do Modelo Relacional

O modelo relacional proposto tem como objetivo refletir de forma clara, consistente e otimizada as principais necessidades de persistência de dados da solução. Ele foi estruturado para atender aos requisitos de controle de lançamentos financeiros(transações) e à condição dos saldos diários, conforme definido nos requisitos funcionais do desafio.



A modelagem segue os seguintes princípios:

- **Simplicidade e objetividade**: Todas as tabelas e campos foram definidos com base em regras de negócio claras, evitando abstrações desnecessárias.
- **Separação de responsabilidades**: Transações financeiras e saldos consolidados foram organizados em tabelas distintas, garantindo independência entre operações transacionais e agregações históricas.
- **Flexibilidade para crescimento:** A modelagem foi pensada para facilitar a inclusão futura de novas entidades, como comerciantes, categorias ou logs de auditoria.
- **Consistência temporal:** Os campos relacionados a data e hora foram definidos com precisão, diferenciando o momento da ocorrência do fato financeiro (`occurred_at`) do momento de persistência (`created_at` ou `generated_at`).
- **Eficiência de leitura:** A estrutura da tabela `daily_balances` permite acesso direto a informações consolidadas sem recalcular dados históricos, otimizando desempenho em relatórios e visualizações.

Este modelo representa uma base sólida e extensível para a persistência de dados da aplicação, respeitando as boas práticas de modelagem relacional, normalização e clareza semântica.

5.2. Tabela de Transações

A tabela `transactions` é responsável por armazenar os lançamentos financeiros individuais que ocorrem no sistema, funcionando como a base transacional para o controle do fluxo de caixa diário do comerciante.

Ela representa cada movimentação financeira registrada, seja de débito ou crédito, permitindo a persistência, rastreabilidade a auditoria dessas operações com precisão.

Tabela: transactions		
Nome	**Tipo**	**Definição**
`id`	`uuid`	Identificador único gerado pelo próprio backend.
`occurred_at`	`timestamp`	Data e hora em que a transação ocorreu de fato.
`amount`	`numeric(14,2)`	Valor monetário da transação.
`type`	`varchar(10)`	Tipo da transação: `debit` ou `credit`.
`description`	`varchar(255)`	Descrição textual opcional da transação.
`created_at`	`timestamp`	Data e hora no banco de dados.

A modelagem foi construída de forma a atender aos seguintes objetivos:

- Registrar com precisão o momento em que a transação ocorreu (`occurred_at`), separando o momento em que foi salva no banco (`created_at`).
- Identificar claramente o valor e tipo da transação (`amount`, `type`), com validação de integridade via `CHECK`.
- Permitir consultas analíticas ou operacionais com base em descrições textuais (`description`), como por exemplo, recebimento de venda via pix, pagamento de taxa de serviço ou estorno de cobrança duplicada, enriquecendo o histórico e facilitando a interpretação semântica das movimentações.
- Garantir unicidade, escalabilidade e compatibilidade com ambientes distribuídos por meio de um identificador global (`id` como `uuid`).

Essa tabela é fundamental para viabilizar o funcionamento do serviço de lançamento e serve de base para a consolidação dos saldos diários, conforme requerido pelo desafio técnico.

5.3. Tabela de Saldos Diários

A tabela `daily_balances` tem como objetivo armazenar o saldo consolidado de lançamentos financeiros por dia, servindo como base para o relatório de fluxo de caixa diário solicitado no desafio.

Essa consolidação é derivada da tabela transactions e permite acesso rápido e eficiente ao saldo diário, sem necessidade de recalcular toda a base de transações a cada consulta.

Tabela: daily_balances		
Nome	**Tipo**	**Definição**
`id`	`uuid`	Identificador único gerado pelo próprio backend.
`reference_date`	`date`	Data de referência do saldo consolidado.
`total_credit`	`numeric(14,2)`	Soma dos créditos registrados no dia.
`total_debit`	`numeric(14,2)`	Soma dos débitos registrados no dia.
`balance`	`numeric(14,2)`	Saldo do dia (`total_credit` – `total_debit`).
`generated_at`	`timestamp`	Data e hora no banco de dados.

A estrutura foi projetada para atender aos seguintes propósitos:

- Registrar a data de referência do saldo (`reference_date`), garantindo uma entrada por dia consolidado.
- Separar o total de créditos e débitos do dia (`total_credit`, `total_debit`), possibilitando análises mais detalhadas e relatórios segmentados.
- Armazenar o saldo do dia (`balance`) com o resultado líquido entre créditos e débitos, facilitando o consumo direto pela aplicação ou visualização pelo usuário.
- Controlar a rastreabilidade do processo de consolidação (`generated_at`), permitindo auditar quando o cálculo foi executado.
- Manter uma chave primária global (`id`) para garantir integridade e facilitar rastreamento em integrações futuras.

Essa tabela otimiza o desempenho de leitura e entrega do saldo diário, além de isolar a lógica de agregação e servir como histórico consolidado para consulta, exportação ou visualização.

6. Considerações de Infraestrutura

6.1. Visão de Implantação

A implantação da solução foi pensada para ser simples, replicável e compatível com múltiplos ambientes. Todo o ecossistema é orquestrado por meio de containers **Docker**, com rede personalizada (**challenge-net**) para isolar a comunicação entre serviços, mantendo segurança e clareza de dependências.

Os **microserviços**, **API Gateway**, **Serviço de Lançamento** e **Serviço de Consolidação**, são hospedados em containers separados, permitindo escalabilidade independente. O frontend (**Aplicação Web**) se conecta exclusivamente ao gateway, que roteia as requisições conforme regras definidas no **YARP**.

Scripts `*.sh` automatizam o provisionamento completo do ambiente (subida de **containers**, criação de volumes e inicialização dos bancos), permitindo fácil execução local e viabilizando pipelines de **CI/CD** em um ambiente corporativo.

6.2. Camada de Cache

Foi implementada uma camada de cache utilizando **Redis** com o objetivo de otimizar leituras frequentes e reduzir a carga sobre os bancos de dados relacionais.

Essa estratégia foi aplicada principalmente no **Serviço de Consolidação**, que persiste os saldos consolidados no **Redis** após o processamento e os consulta diretamente para exibir saldos já calculados ao comerciante. Isso reduz significativamente o tempo de resposta em cenários de alta demanda, como dashboards ou visualizações repetidas.

O cache é acessado via chave composta com base na data de referência (`daily_balance:{yyyy-MM-dd}`), garantindo rápida localização e coerência com os dados consolidados.

6.3. Persistência

A persistência foi preparada por contexto, adotando bancos **PostgreSQL** independentes para os serviços de **Lançamento** e **Consolidação**, promovendo isolamento de dados e simplificando a governança de schema.

O **Serviço de Lançamento** grava transações financeiras individuais (crédito e débito) com rastreabilidade e integridade. O **Serviço de Consolidação** consome essas transações e persiste os saldos diários consolidados em uma estrutura otimizada para leitura analítica. O uso de **UUID** como chave primária em todas as tabelas reforça a escalabilidade horizontal e a compatibilidade com arquiteturas distribuídas.

Além disso, as tabelas foram modeladas com foco em clareza semântica e desempenho, respeitando princípios de normalização e separação de responsabilidades.

7. Segurança e Isolamento

7.1. Papel do API Gateway

O **API Gateway** atua como a única porta de entrada para os microserviços da plataforma centralizando o tráfego, roteando requisições e aplicando políticas transversais como Logging, autenticação e balanceamento de carga. Sua presença permite que os microserviços permaneçam isolados da exposição direta, reforçando a segurança, simplificando o consumo e facilitando futuras mudanças de rotas sem impactar o cliente.

7.2. Autenticação e Autorização

Para garantir o acesso seguro aos dados e operações, a plataforma foi projetada para suportar autenticação baseada em **JWT** (JSON Web Token). O token é validado pelo **API Gateway**, que rejeita acessos não autenticados antes de repassar a requisição ao microserviço responsável. A autorização é aplicada conforme o escopo do token, assegurando que usuários só acessem recursos permitidos.

7.3. Comunicação Interna vs Externa

A comunicação externa, entre a **Aplicação Web** e a **Plataforma**, é realizada via **HTTPS** com payloads **JSON**, garantindo segurança e interoperabilidade. Já a comunicação interna entre os microserviços ocorre de forma isolada dentro da rede **Docker** privada (**challenge-net**), também via **HTTP/JSON**, sem exposição pública. Essa separação melhora o isolamento de rede e reduz a superfície de ataque.

8. Escalabilidade e Resiliência

8.1. Estratégia de Alta Demanda e Cache

Para suportar cenários de alta demanda, a arquitetura implementa uma camada de cache utilizando **Redis**. O cache é utilizado especificamente para armazenar os saldos diários consolidados, evitando consultas repetidas ao banco de dados e melhorando a performance das requisições. Essa estratégia contribui para a escalabilidade horizontal da solução da latência em consultas frequentes.

8.2. Desacoplamento entre Serviços

A solução foi projetada com base na arquitetura de microserviços, garantindo independência entre os componentes. O **Serviço de Lançamento** e o **Serviço de Consolidação** operam de forma autônoma, cada um com seu banco de dados isolado, o que facilita a manutenção, implantação individual e escalabilidade de forma independente.

8.3. Considerações de Tolerância a Falhas

A arquitetura contempla estratégias para tolerância a falhas, como execução assíncrona da consolidação dos saldos e uso do **Redis** como cache para recuperação rápida. Além disso, a separação de responsabilidades entre os serviços e a presença do **gateway** permitem isolar

falhas e mitigar impactos em cadeia. Isso assegura maior resiliência operacional e disponibilidade mesmo em casos de degradação parcial dos componentes.

9. Conclusão e Possíveis Evoluções

9.1. Resumo da Solução Proposta

A solução foi estruturada com base em microserviços, visando modularidade, escalabilidade e facilidade de manutenção. Um **API Gateway** centraliza as requisições e realiza o roteamento para os serviços de **Lançamento** e **Consolidação**, responsáveis por registrar transações e gerar saldos diários, respectivamente. O uso de cache com **Redis** otimiza a leitura de dados consolidados e melhora o desempenho. A **Aplicação Web** consome os serviços de forma segura e desacoplada via **SPA**.

9.2. Limitações e Pontos de Atenção

Apesar da arquitetura escalável e modular, o ambiente atual ainda depende de execução local via **Docker** e não contempla mecanismos de observabilidade completos (logs centralizados, métricas ou tracing). Além disso, a comunicação entre serviços e bancos é síncrona, o que pode limitar a resiliência em casos de alto volume ou falhas temporárias. O processo de consolidação ainda opera de forma simplificada, sem agendador robusto.

9.3. Melhorias Futuras Recomendadas

Para evolução da solução, recomenda-se implementar autenticação via **JWT** com Refresh token, adicionar métricas e rastreamento distribuído, integrar com serviços de mensageria como **RabbitMQ** para desacoplamento de eventos e automatizar o agendamento da consolidação com um orquestrador como **Hangfire** ou **Quartz**. A implementação contínua e testes automatizados também são aspectos que podem ser incorporados para aumentar a maturidade da arquitetura.

Também é recomendada a implementação efetiva da camada de cache com **Redis** para otimizar a leitura dos saldos consolidados. Apesar de estar prevista na arquitetura, sua configuração não foi realizada nesta entrega devido à limitação de tempo para o desafio.

Por fim, destaca-se a importância de fortalecer os pilares de **observabilidade**. Mesmo em ambiente local com o **WSL2**, é possível integrar logs centralizados como por exemplo **Serilog** e **Grafana Loki** expondo métricas básicas como **Prometheus**, criando uma base sólida para diagnóstico, rastreamento e análise de performance dos microserviços.