Arquitetura e Estrutura do Projeto

O sistema BtgLedger adota uma arquitetura distribuída, separando as responsabilidades entre a interface de usuário (Frontend), a interface de programação de aplicações (Backend API), o processamento assíncrono (Worker) e a orquestração de infraestrutura.

Backend (MINI_LEDGER_API)

```
/src
│
├── /BtgLedger.API
(Camada de Apresentação HTTP: Contém os Controllers,
como o AccountsController, e os DTOs de Request e Response.)
│
├── /BtgLedger.Domain
(Núcleo de Regras de Negócio: Implementa o padrão Strategy
para transações (Credit, Debit, Refund) e segrega as interfaces
em Commands e Queries (CQRS).)
│
├── /BtgLedger.Infrastructure
(Camada de Dados e Integração: Gerencia a persistência com
Entity Framework (AppDbContext), Migrations e a
comunicação com mensageria através do RabbitMqService.)
│
└── /BtgLedger.Worker
(Processamento em Background: Serviço dedicado a consumir
e processar mensagens de forma assíncrona vindas do RabbitMQ.)

/tests
└── /BtgLedger.Tests
(Qualidade e Confiabilidade: Contém os testes unitários
focados nas regras de domínio, como AccountTests e
TransactionStrategyFactoryTests.)
```

Infraestrutura e DevOps (Docker e Kubernetes)

```
/Raiz do Backend
├── docker-compose.yml
(Orquestração local do ambiente, subindo o banco de dados,
RabbitMQ e a API em conjunto.)
├── Dockerfile.API
(Receita de construção da imagem Docker para a API.)
└── Dockerfile.Worker
(Receita de construção da imagem Docker para o Worker.)

/.github/workflows
└── ci-cd.yml
(Pipeline de automação para Integração e
Entrega Contínuas (CI/CD).)

/k8s
├── api-deployment.yaml
(Manifesto Kubernetes para o deploy e escalonamento da API.)
├── infra-deployment.yaml
(Manifesto Kubernetes para os serviços de infraestrutura,
como o RabbitMQ e o Banco de Dados.)
└── worker-deployment.yaml
(Manifesto Kubernetes para o deploy do Worker assíncrono.)
```

Frontend (MINI_LEDGER_APP)

```
/btg-ledger-web
├── /src
(Contém o código-fonte da aplicação React.)
├── tsconfig.json
(Configurações do TypeScript, garantindo segurança de tipos.)
└── vite.config.ts
(Utilização do Vite como empacotador (bundler)
para um desenvolvimento e build mais rápidos.)
```

Camada 1: Visão geral do sistema

O que é o BTG Ledger?

O BTG Ledger é um núcleo de processamento financeiro (livro-razão) full-stack, distribuído e estruturado de forma Cloud Native. O sistema atua como o motor de backend primário responsável por gerenciar o ciclo de vida de contas bancárias, o fluxo de transações com alta segurança (créditos e débitos) e a autenticação multifator (2FA) dos clientes.

O Problema enfrentado e a solução arquitetural

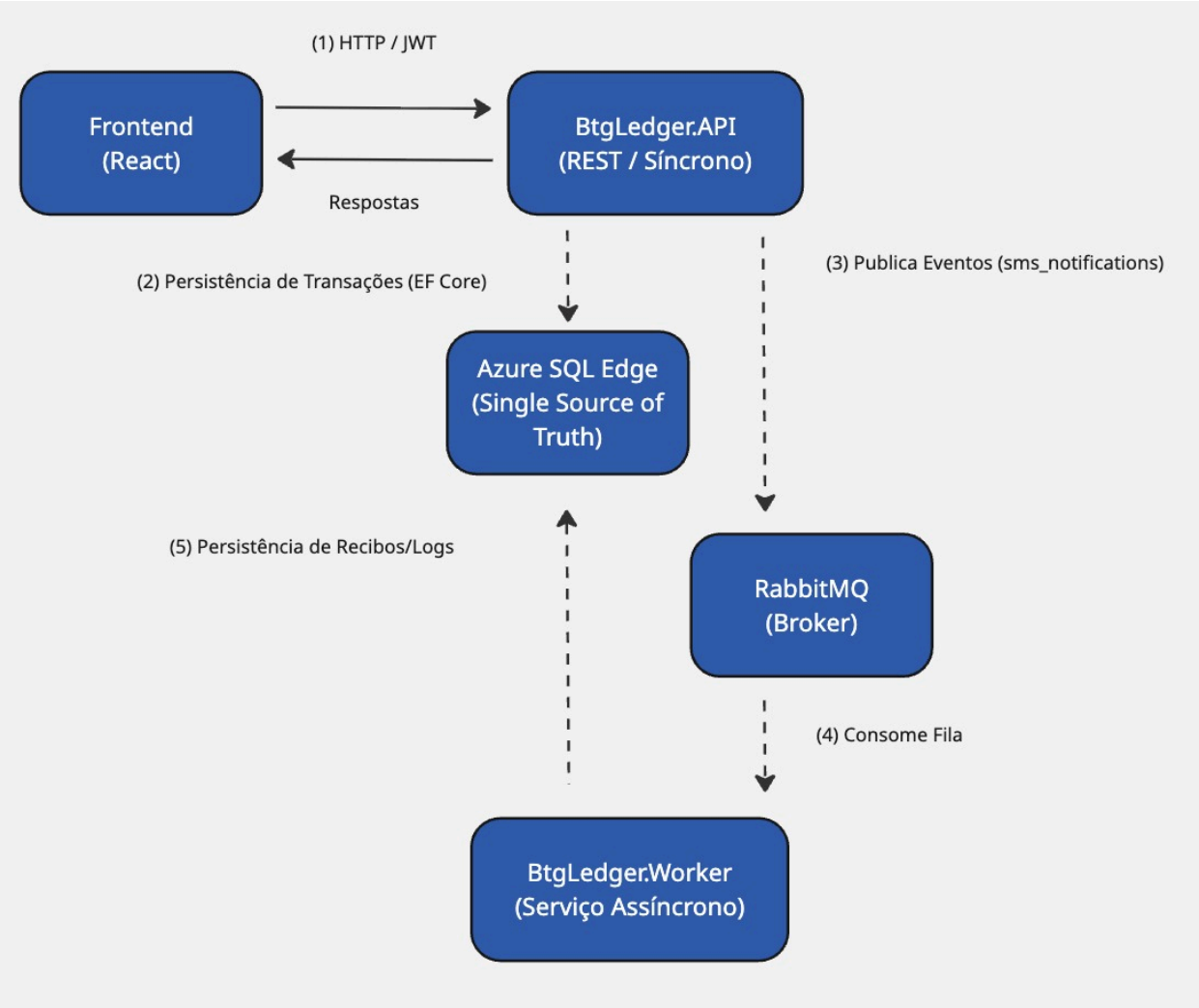
Historicamente, sistemas financeiros tradicionais enfrentam problemas críticos de esgotamento de recursos e gargalos de I/O (Input/Output). Durante a concepção do BTG Ledger, o maior desafio técnico mapeado foi o risco de bloqueio da thread (linha de execução) da API em operações dependentes de terceiros.

- O Cenário de Falha: Em um fluxo de login síncrono, se a API tentasse enviar o SMS de autenticação e a operadora de telefonia demorasse 10 segundos para responder, a API ficaria travada aguardando. Em um momento de pico de acessos, isso levaria ao esgotamento das threads e à queda do servidor.
- A Solução Aplicada: O sistema resolve esse gargalo implementando uma Arquitetura Orientada a Eventos (Event-Driven Architecture). A API valida o login e, em vez de enviar o SMS, delega a tarefa publicando um evento no Message Broker (RabbitMQ). Isso libera a conexão do usuário em milissegundos. O processamento pesado e o envio real da notificação são absorvidos de forma totalmente assíncrona por um microsserviço isolado (Worker), garantindo que o núcleo financeiro da API nunca seja interrompido.

Quem deveria usar?

- Desenvolvedores Frontend/Mobile: Consumindo os endpoints da API para construir interfaces de banking (como o painel em React.js já integrado).
- Engenheiros de Infraestrutura e SREs: Responsáveis por escalar os Pods no Kubernetes de forma independente (ex: subir mais Workers se a fila de SMS crescer).

Desenho simples da arquitetura



Resumo do Fluxo:

1. Requisição: O Frontend envia uma requisição HTTP autenticada via JWT para a aplicação BtgLedger.API (ponto de entrada síncrono).
2. Persistência Core: A API valida a regra de negócio e salva o estado da transação na base relacional Azure SQL Edge (Single Source of Truth) via EF Core.
3. Publicação de Evento: A API avisa o RabbitMQ (Broker) publicando um evento na fila de notificações (ex: necessidade de envio de SMS).
4. Consumo Assíncrono: O BtgLedger.Worker consome a fila do RabbitMQ de forma contínua e executa a tarefa externa pesada.
5. Atualização de Estado: Após finalizar, o Worker pode atualizar o banco de dados com o recibo ou log do processamento.

Camada 2: Como rodar o projeto

O princípio desta camada é a reprodutibilidade imediata. Todo o ecossistema (Banco de Dados, Mensageria, API e Worker) foi 100% "dockerizado". O objetivo é garantir

que o projeto rode em qualquer máquina exatamente como rodaria em produção, sem poluir o sistema operacional local com instalações pesadas.

Pré-requisitos

Para rodar a stack completa, será preciso de:

- Docker Desktop.
- Git.
- .NET 10 SDK (Apenas se quiser rodar os testes unitários ou debugar o código C# fora do Docker).
- Node.js 18+

Nota para usuários de Mac (Apple Silicon / M1+): O projeto utiliza a imagem mcr.microsoft.com/azure-sql-edge para o banco de dados, o que garante compatibilidade nativa com a arquitetura ARM64, sem emulação pesada.

Desafios de setup e soluções arquiteturais

Durante o desenvolvimento da infraestrutura local, esbarrei em desafios de hardware e de ciclo de vida de contêineres que exigiram soluções específicas:

- Desafio de Hardware (Arquitetura ARM64 vs. Banco de Dados): O desenvolvimento ocorreu em um ambiente macOS com processador Apple Silicon (chip M-series, arquitetura ARM64). As imagens Docker oficiais do Microsoft SQL Server são otimizadas para x86/AMD64. Tentar rodar o SQL Server tradicional via emulação (Rosetta) no Mac causava consumo excessivo de recursos, lentidão na inicialização e quebras inesperadas do contêiner.
    - A Solução: Para manter a paridade com o ecossistema corporativo Microsoft sem sacrificar a performance local, substituí a imagem pela mcr.microsoft.com/azure-sql-edge. O Azure SQL Edge compartilha a mesma engine do SQL Server, mas possui suporte nativo para ARM64 , garantindo inicializações quase instantâneas.
- Desafio de Migrations em Contêineres Efêmeros: Em uma arquitetura Cloud Native (especialmente rodando Kubernetes), os Pods podem ser destruídos e recriados a qualquer momento. Executar comandos manuais de migração (como dotnet ef database update) não é escalável nem seguro.
    - A Solução (Code-First automatizado): Adotei a execução automática no startup da aplicação. A API foi configurada para que, antes de aceitar requisições HTTP, injete o contexto do banco e execute

db.Database.Migrate(). Isso torna o deploy autossuficiente: o próprio software garante que o banco possui as tabelas necessárias.

Variáveis de ambiente

Você não precisa configurar nenhum arquivo .env para rodar o ambiente de desenvolvimento. O arquivo docker-compose.yml e os manifestos do Kubernetes já injetam as variáveis padrão automaticamente:

- ConnectionStrings__DefaultConnection: Conecta a API e o Worker ao Azure SQL Edge.
- RABBITMQ_HOST: Aponta os serviços para o contêiner do Message Broker.*

Subindo a stack

A maneira mais à prova de falhas de rodar o BTG Ledger localmente é usando o Docker Compose. Ele cria a rede virtual e orquestra a ordem de inicialização.

Executando Localmente (Docker Compose)

1. Clone e inicie o ambiente:

```
git clone https://github.com/SEU_USUARIO/btg-ledger.git
cd btg-ledger
docker-compose up -d --build
```

2. Suba o frontend

Em um novo terminal, inicie a interface visual:

```
cd src/Frontend
npm install
npm run dev
```

Acesse a aplicação em http://localhost:5173. O frontend já está configurado para bater na API na porta 5088.

3. Validação de Sucesso: Verifique se tudo está de pé.

- API Swagger: Acesse http://localhost:5088/swagger para ver a documentação interativa dos endpoints.
- RabbitMQ Management: Acesse http://localhost:15672 (Usuário: guest, Senha: guest) para ver as filas.

4. Lendo o PIN de 2FA

Como não estamos enviando SMS de verdade para não gerar custos de telefonia, o PIN de Autenticação em Duas Etapas é capturado e "impresso" pelo Worker Service.

Quando você tentar fazer o Login no React, a tela vai pedir o PIN. Para pegá-lo, leia os logs do contêiner do Worker:

```
docker logs -f btg-ledger-worker-1
```

(O PIN de 6 dígitos aparecerá instantaneamente neste terminal assim que você clicar em "Entrar" no frontend).

6. Como Rodar os Testes

A qualidade é garantida por testes automatizados (Unitários e de Integração). Para rodar a suíte de testes e validar se as suas alterações quebraram alguma regra de domínio, execute na raiz do projeto:

```
dotnet test BtgLedger.slnx
```

O console retornará a quantidade de testes que passaram, falharam ou foram ignorados. O pipeline de CI/CD do GitHub Actions rodará exatamente este comando antes de aceitar qualquer Merge Request.

7. Desligando o Ambiente

Para parar a execução e limpar os contêineres, redes e volumes da sua máquina:

```
docker-compose down
```

Executando via Kubernetes (simulação de produção)

Para validar escalonamento, balanceamento de carga e auto-healing, o sistema pode ser executado via K8s.

1. Garanta que o Docker Compose está desligado (docker-compose down).
2. Aplique a base (Banco e Mensageria) e as Aplicações:

```
kubectl apply -f k8s/infra-deployment.yaml
kubectl apply -f k8s/api-deployment.yaml
kubectl apply -f k8s/worker-deployment.yaml
```

3. Libere a porta da API para acesso local via Port-Forwarding:

```
kubectl port-forward svc/btg-ledger-api-service 5088:5088
```

4. Para capturar o PIN do 2FA no Kubernetes, escute os logs usando a Label do Worker (evitando problemas com nomes dinâmicos de Pods):

```
kubectl logs -f -l app=btg-ledger-worker
```

Camada 3: Como usar (Contratos da API e Fluxos)

O BTG Ledger expõe uma API RESTful síncrona para o cliente, atuando como a fronteira entre o mundo externo (Frontend) e as regras de negócio. Abaixo estão detalhados os desafios de segurança enfrentados ao desenhar esta camada, os fluxos esperados na interface gráfica e os contratos estritos dos endpoints.

1. Desafios Enfrentados na Exposição da API

Expor um sistema financeiro à web traz desafios críticos de design e segurança. Durante a construção dos Controllers, os seguintes problemas foram mapeados e resolvidos:

- Vazamento de Domínio e Over-posting: Um erro comum é receber ou devolver Entidades do banco de dados (como a classe Account) diretamente nas requisições HTTP, expondo a estrutura interna do banco e permitindo injeção de dados.
    - Solução: Implementação estrita do padrão de DTOs (Data Transfer Objects). A API recebe classes burras (como CreateAccountRequest), valida a estrutura e só então mapeia para o Domínio.

- Gerenciamento de Estado do 2FA (Gargalo de I/O): Salvar o PIN temporário de 6 dígitos no SQL Server adicionaria um gargalo de I/O desnecessário e o problema de limpar esses códigos expirados.
    - Solução: Utilização do IMemoryCache nativo do ASP.NET Core. O PIN é guardado diretamente na memória RAM do servidor atrelado ao ID da conta, com um Time-To-Live (TTL) estrito de 5 minutos. Se não for validado, o .NET destrói o dado silenciosamente.
- Autenticação e "Passe-Livre" Seguro: Como autorizar transferências sem exigir a senha a cada clique?
    - Solução: Autenticação Stateless baseada em JSON Web Tokens (JWT). As rotas financeiras foram blindadas com o atributo [Authorize]. Qualquer requisição sem um token válido é interceptada retornando 401 Unauthorized antes de tocar no código de negócio.

2. O Fluxo visual

Se você estiver testando o sistema pela interface gráfica, o comportamento esperado é este:

1. Cadastro: O usuário cria uma conta fornecendo dados básicos. O sistema gera automaticamente um Número de Conta e Agência.
2. Login (Etapa 1): O usuário insere a Conta e a Senha. Se estiverem corretos, a tela muda para o campo de "Insira seu PIN". Neste exato milissegundo, a API jogou uma mensagem no RabbitMQ.
3. Login (Etapa 2 - O 2FA): Como não há disparo real de SMS no ambiente de dev, você deve olhar o terminal onde rodou docker logs -f btg-ledger-worker-1. Digite o PIN de 6 dígitos que apareceu lá na tela do React.
4. Dashboard: O React recebe o token JWT (invisível para o usuário) e o redireciona para o painel principal, onde o saldo é carregado e operações de Débito/Crédito são liberadas.

3. Referência da API (Endpoints Principais)

Todas as rotas baseiam-se no endpoint principal /api/accounts

A. Criação de Conta e Autenticação

Criar uma Conta Bancária

- Endpoint: POST /api/Accounts

- Descrição: Cria uma nova conta. A senha é criptografada com algoritmo BCrypt antes de ser salva, e a agência/número são gerados automaticamente.
- Request Body:

```
{
  "ownerName": "Renan Lima",
  "initialBalance": 1000.00,
  "password": "SenhaSuperSegura123!",
  "phoneNumber": "21999999999"
}
```

- Response (201 Created): Retorna o objeto da conta com o ID gerado.

Solicitar Login (Gera o PIN)

- Endpoint: POST /api/Accounts/login
- Descrição: Valida a agência, conta e senha. Se corretos, gera um PIN no cache de memória e enfileira o envio do SMS no RabbitMQ.
- Request Body:

```
{
  "agency": "0001",
  "number": "123456",
  "password": "SenhaSuperSegura123!"
}
```

- Response (200 OK): { "message": "PIN de validação enviado por SMS.", "accountId": "a1b2c3d4-e5f6-7a8b-9c0d-1e2f3a4b5c6d", "requires2FA": true }
- Response (401 Unauthorized): { "error": "Agência, conta ou senha inválidos." }

Validar PIN (Retorna o JWT)

- Endpoint: POST /api/Accounts/validate-pin
- Descrição: Valida o PIN temporário atrelado ao ID da conta. Se correto, emite o token JWT com validade de 2 horas.
- Request Body:

```
{
  "accountId": "a1b2c3d4-e5f6-7a8b-9c0d-1e2f3a4b5c6d",
  "pin": "849201"
}
```

- Response (200 OK): { "token": "eyJhbG...", "accountId": "a1b2c3d4-e5f6-7a8b-9c0d-1e2f3a4b5c6d" }
- Response (401 Unauthorized): { "error": "PIN inválido ou expirado." }

B. Operações Financeiras

Consultar Dados da Conta

- Endpoint: GET /api/accounts/{id}
- Descrição: Retorna os dados fundamentais e o saldo atual da conta.
- Response (200 OK): { "id": "a1b2...", "ownerName": "Renan Lima", "balance": 1000.00, "agency": "0001", "number": "123456" }
- Response (404 Not Found): "Account not found."

Realizar uma Transação

- Endpoint: POST /api/accounts/{id}/transaction
- Segurança: Requer cabeçalho HTTP Authorization: Bearer .
- Descrição: Processa um débito ou crédito. A regra de domínio blinda a operação (ex: impede débito se o saldo for menor que o valor).
- Request Body:

```
{
  "amount": 150.75,
  "type": "Debit"
}
```

- Response (200 OK): { "message": "Transaction processed successfully." }
- Response (400 Bad Request): Retorna a mensagem de erro da camada de domínio caso a regra seja violada.

4. Dicionário de Códigos de Erro HTTP

A API respeita a semântica do protocolo HTTP:

- 200 OK / 201 Created: Operação concluída com sucesso.
- 400 Bad Request: Violação de regra de negócio (ex: saldo insuficiente) capturada via bloco try/catch.
- 401 Unauthorized: Credenciais incorretas, PIN expirado ou falta do token JWT nas rotas protegidas.

- 404 Not Found: Conta não localizada na base de dados.

Camada 4: Decisões Técnicas e Desafios de Engenharia

O BTG Ledger não foi construído apenas para "funcionar", mas para suportar a carga, o rigor de segurança e a alta disponibilidade exigidos por uma instituição financeira. Abaixo está documentado as escolhas arquiteturais fundamentais e os problemas que elas resolveram.

1. Por que Clean Architecture e Domain-Driven Design (DDD)?

Sistemas financeiros não podem ter suas regras de negócio acopladas a frameworks voláteis.

- O Problema: Em arquiteturas tradicionais (monolíticas ou em três camadas simples), é comum ver a lógica de transferência bancária misturada com consultas SQL ou regras de interface, gerando o chamado "Modelo Anêmico".
- A Decisão: Foi adotado a Clean Architecture para blindar o coração do software (BtgLedger.Domain). O Domínio não sabe nada sobre RabbitMQ, SQL Server ou APIs. Para evitar adulterações indevidas de estado, foi aplicado DDD com forte encapsulamento (uso de private setters). O saldo só é modificado através de métodos específicos de negócio (como ProcessDebit), garantindo a consistência da Entidade.

2. O Conflito: Entity Framework Core vs. Modelo Rico (DDD)

A escolha natural para persistência no .NET é o EF Core, mas ele trouxe um desafio de encapsulamento.

- O Problema: O EF Core tradicionalmente exige construtores vazios e setters públicos para popular os dados lidos do banco. Abrir esses setters destruiria a proteção do nosso Domínio. Usar Data Annotations (como [Table]) poluiria as Entidades puras com dependências da Microsoft.
- A Decisão: Utilização estrita da Fluent API via interface IEntityTypeConfiguration dentro da camada de Infraestrutura. O EF Core moderno usa Reflection nos bastidores para preencher as propriedades com private setters, mantendo o nosso Domínio 100% agnóstico e imutável.

3. O Desafio de Hardware e Banco de Dados (ARM64 vs. x86)

Para garantir que a stack rodasse perfeitamente no ambiente local de desenvolvimento antes de ir para a nuvem.

- O Problema: O desenvolvimento foi feito em um Mac com processador Apple Silicon (arquitetura ARM64). As imagens oficiais do Microsoft SQL Server no Docker são otimizadas para x86/AMD64. Rodá-las via emulação (Rosetta) causava lentidão extrema na inicialização e quebras de contêiner.
- A Decisão: Foi substituído a imagem tradicional pela imagem do Azure SQL Edge (mcr.microsoft.com/azure-sql-edge). Ele compartilha a mesma engine do SQL Server, mas possui suporte nativo para ARM64, permitindo inicializações quase instantâneas e garantindo paridade total com o ecossistema corporativo da Microsoft.

4. Segurança e 2FA: BCrypt e MemoryCache

Lidar com dinheiro exige defesas contra vulnerabilidades comuns de autenticação.

- O Problema da Senha: Salvar senhas em texto puro ou usar hashes antigos (como MD5) abre margem para ataques de força bruta ou Rainbow Tables.
    - A Decisão da Senha: Foi implementado o BCrypt, um algoritmo propositalmente lento que gera Salts dinâmicos, tornando a base de dados altamente resistente a ataques.
- O Problema do 2FA: Salvar o PIN temporário de 6 dígitos no SQL Server adicionaria gargalo de I/O desnecessário e o custo de gerenciar exclusões após a expiração.
    - Decisão do 2FA: Foi utilizado o IMemoryCache nativo do .NET. O PIN vive diretamente na memória RAM do servidor com um Time-To-Live (TTL) rigoroso de 5 minutos. Se o tempo esgotar, o próprio .NET destrói o dado silenciosamente, otimizando recursos e fechando a janela de ataque.

5. Resiliência: Arquitetura Orientada a Eventos (RabbitMQ)

Um sistema bancário não pode parar porque um serviço de terceiros ficou lento.

- O Problema: Se a API enviasse o SMS do 2FA de forma síncrona durante o login e a operadora de telefonia demorasse a responder, a thread do servidor ficaria bloqueada. Em picos de tráfego, isso esgotaria os recursos e derrubaria o núcleo financeiro.

- A Decisão: Adoção do padrão Event-Driven Architecture. A API aprova o login, publica o evento sms_notifications no RabbitMQ em milissegundos e libera a conexão do usuário. O envio real é delegado a um microsserviço assíncrono isolado (o Worker). Isso garante Escalabilidade Independente: podemos subir mais instâncias apenas do Worker se a fila de notificações crescer.

6. Orquestração: Por que Kubernetes?

- O Problema: O Docker Compose resolve o "funciona na minha máquina", mas não tem inteligência para recriar contêineres que travam (auto-healing) ou redistribuir o tráfego de forma dinâmica em produção.
- A Decisão: Foi projetado os manifestos para implantação no Kubernetes (K8s). Os Deployments garantem que o número exato de réplicas esteja sempre rodando (resiliência), enquanto os Services atuam como balanceadores de carga internos para o Frontend não precisar mapear IPs voláteis da API.

Camada 5: Estrutura do Código e Modelo Mental

A solução BtgLedger.slnx foi estruturada sob os rigorosos princípios da Clean Architecture e do Domain-Driven Design (DDD). O sistema é dividido em camadas estritas, aplicando o Princípio da Responsabilidade Única (SRP - SOLID) em nível de arquitetura.

O fluxo de dependência aponta sempre para o centro (o Domínio). As camadas externas dependem das internas, mas o Domínio não depende de ninguém.

O Mapa de Diretórios

Abaixo está a representação da arquitetura de pastas e as responsabilidades de cada módulo:

1. src/BtgLedger.Domain/

- Responsabilidade: Contém 100% das regras de negócio bancárias. É o módulo mais protegido do sistema. Não sabe absolutamente nada sobre bancos de dados, frameworks web, APIs ou mensageria.
- O que você encontra aqui:
  - Entidades (Entities): Classes fundamentais de negócio como Account e Transaction. Elas utilizam private setters para evitar o "Modelo Anêmico" e garantir que o saldo nunca seja adulterado fora das regras.

- Contratos (Interfaces): Definições como IAccountRepository. O Domínio diz: "Preciso de algo que salve uma conta". Ele não se importa se será no SQL Server ou em um arquivo de texto.

2. src/BtgLedger.Infrastructure/

- Responsabilidade: Sabe como conectar a aplicação ao mundo real (ler e gravar dados, enviar mensagens). É onde as tecnologias específicas são encapsuladas.
- O que você encontra aqui:
  - Mapeamento e Persistência (EF Core): Classes de configuração que usam a Fluent API para mapear as Entidades do Domínio para o banco de dados relacional sem violar os private setters. A implementação concreta do AccountRepository vive aqui.
  - Mensageria (RabbitMQ): A implementação concreta do publicador de eventos (MessageBusService) que enviará os dados para a fila.
  - Migrations: O histórico de evolução do esquema do banco de dados (Code-First).

3. src/BtgLedger.API/

- Responsabilidade: É o mecanismo de entrega (Delivery Mechanism). Atua como a porta de entrada para requisições HTTP, delegando o trabalho pesado para o Domínio ou Infraestrutura.
- O que você encontra aqui:
  - Controllers: Como o AccountsController, que processa a entrada e saída.
  - DTOs (Data Transfer Objects): Classes burras usadas para receber dados estritos e evitar vulnerabilidades de Over-posting ou vazamento do Domínio.
  - Segurança e Cache: Configurações do JWT (Bearer Auth) e o gerenciamento de estado do PIN de 2FA utilizando o IMemoryCache.
  - Injeção de Dependência: O arquivo Program.cs onde o contêiner de IoC do .NET é instruído a resolver as interfaces do Domínio entregando as classes concretas da Infraestrutura.

4. src/BtgLedger.Worker/

- Responsabilidade: É um microsserviço completamente autônomo, rodando em segundo plano. Ele não possui interface ou rotas web.

- O que você encontra aqui:
  - Consumers: Classes construídas sobre o BackgroundService do .NET, focadas em escutar as filas do RabbitMQ de forma ininterrupta. Aqui ocorre o processamento simulado do envio de SMS e consolidação de logs de transações.

5. tests/BtgLedger.Tests/

- Responsabilidade: Armazenar toda a suíte de garantias do software.
- O que você encontra aqui:
  - Testes unitários para blindar as regras de negócio das Entidades e testes de integração.

Camada 6: Testes e Qualidade

A qualidade do BTG Ledger não é presumida, ela é validada continuamente a cada linha de código alterada. O sistema foi desenhado para ser inerentemente testável, graças à separação rigorosa de responsabilidades da Clean Architecture.

1. A Suíte de Testes (O que testamos?)

O projeto de testes está isolado no diretório tests/BtgLedger.Tests/.

- Testes Unitários: BtgLedger.Domain. Com as Entidades (Account, Transaction) possuem regras de negócio encapsuladas, os testes unitários garantem que:
  - Não é possível debitar um valor maior que o saldo atual.
  - Transações com valores negativos são rejeitadas instantaneamente.
  - O estado da conta só muda através dos métodos oficiais (ex: ProcessDebit).

2. Como Rodar os Testes Localmente

Nenhum código deve ser "commitado" sem antes passar pelo crivo local. Para rodar a suíte inteira e validar se as suas alterações quebraram alguma regra de domínio, abra o terminal na raiz do projeto e execute:

```
dotnet test BtgLedger.slnx
```

O console do .NET retornará imediatamente a quantidade de testes que passaram, falharam ou foram ignorados. Se houver falhas, a mensagem de erro indicará

exatamente qual regra de negócio foi violada e em qual linha.

3. A Barreira de Qualidade Contínua (CI/CD)

O teste local é importante, mas humanos esquecem. Para isso, a automação foi importante aqui.

O pipeline do GitHub Actions (.github/workflows/ci-cd.yml) atua como o guardião da ramificação principal (main). Em cada Push ou Pull Request, o robô na nuvem provisiona um servidor Linux isolado e roda exatamente o comando dotnet test BtgLedger.slnx.

A premissa é simples: Se um teste falhar, o pipeline quebra, a "bolinha fica vermelha" e o código defeituoso é bloqueado, sendo impedido de chegar ao ambiente de produção ou mesmo gerar as imagens Docker finais.

4. Protocolo para Adicionar Novos Testes

Sempre que uma nova funcionalidade for adicionada ou um bug for corrigido, um teste automatizado correspondente deve nascer junto com ele. O protocolo exige que:

- Isole o Cenário: Configure o estado inicial (ex: "Conta com saldo de 100 reais").
- Execute a Ação: Chame o método que está sendo testado (ex: "Tentar debitar 150 reais").
- Valide o Resultado: Afirme (Assert) que a exceção correta foi lançada ou que o saldo permaneceu inalterado.

Camada 7: Operações e Manutenção

O BTG Ledger foi desenhado para ser operado com o mínimo de atrito e o máximo de automação. Abaixo estão os protocolos de deploy, monitoramento e as tarefas comuns de manutenção do ecossistema Cloud Native.

1. Deploy Automatizado e CI/CD

A integração e entrega contínuas do projeto são providas nativamente via GitHub Actions.

- O Gatilho: O workflow configurado em .github/workflows/ci-cd.yml é acionado automaticamente a cada push ou pull_request na branch main.

- A Esteira de Compilação: O pipeline provisiona um robô Linux na nuvem que faz o checkout do código, restaura as dependências e compila a solução C# inteira (BtgLedger.slnx).
- Validação de Contêineres: Por fim, o pipeline valida a construção das imagens Docker da API e do Worker em um ambiente ubuntu-latest, assegurando a integridade dos Dockerfiles para deployment. Se houver qualquer falha, o deploy é bloqueado.

2. Migrações de Banco de Dados (Zero Downtime)

Esquecer de rodar scripts SQL em produção é uma das maiores causas de falha em deploys. O BTG Ledger resolve isso com automação.

- Estratégia Code-First: A evolução do esquema de banco de dados no Azure SQL Edge é gerenciada estritamente pelo Entity Framework Core (Code-First).
- Execução no Startup: O banco de dados aplica as migrações automaticamente através do comando db.Database.Migrate() durante o startup do Pod da API.
- Como funciona: O próprio software verifica se o banco de dados possui todas as tabelas necessárias; se não possuir, ele aplica as migrações instantaneamente antes de começar a aceitar requisições HTTP. Isso torna o deploy autossuficiente e livre de intervenções manuais.

3. Monitoramento e Logs (Observabilidade)

A arquitetura orientada a eventos exige rastreabilidade. Como as mensagens do RabbitMQ são processadas em segundo plano , monitorar os logs do Worker é essencial para a saúde do sistema (e para capturar o PIN do 2FA em ambiente local).

- Ambiente Docker Compose: Para acompanhar os logs de execução, utilize o comando docker logs -f btg-ledger-worker-1.
- Ambiente Kubernetes: Os Pods do K8s são efêmeros (seus nomes mudam). Para nunca perder o rastro do Worker, utilizamos Labels (etiquetas). O comando definitivo para escutar o serviço de mensageria em tempo real é: kubectl logs -f -l app=btg-ledger-worker.

4. Resiliência e Auto-Healing (Kubernetes)

O Kubernetes atua como o maestro da nossa infraestrutura, garantindo que o estado desejado da aplicação seja sempre mantido.

- Recuperação de Falhas: O sistema foi desenhado separando a topologia em Deployments e Services. O Deployment garante a resiliência e o auto-healing (autocura). Se o Pod do Worker que processa os SMS for destruído por qualquer motivo, o Kubernetes percebe a anomalia e levanta um novo Pod

instantaneamente, conectando-o de volta à fila do RabbitMQ sem perda de dados.

- Balanceamento: Os Services atuam como balanceadores de carga internos. Eles distribuem o tráfego de rede de forma inteligente entre as réplicas saudáveis disponíveis, garantindo a alta disponibilidade exigida por um núcleo financeiro.