FACULDADE DE TECNOLOGIA DE SOROCABA – FATEC
PROGRAMAÇÃO MULTIPLATAFORMA – 2º A – TARDE – ENSINO SUPERIOR

ADRIAN DA PAZ MARANDOLA

JOSÉ ROBERTO LISBOA DA SILVA FILHO

LUIZ GABRIEL RODRIGUES FREI

FELIPE SALCEDO RAMALHO

ISABEL ALMEIDA DA SILVA

ISW, PMA

SOROCABA – SP
2025

ADRIAN DA PAZ MARANDOLA

JOSÉ ROBERTO LISBOA DA SILVA FILHO

LUIZ GABRIEL RODRIGUES FREI

FELIPE SALCEDO RAMALHO

ISABEL ALMEIDA DA SILVA

ISW, PMA

Documentação do Projeto Binance, apresentado à Faculdade de Tecnologia de Sorocaba.

Orientador: André Cassulino Araújo Souza.

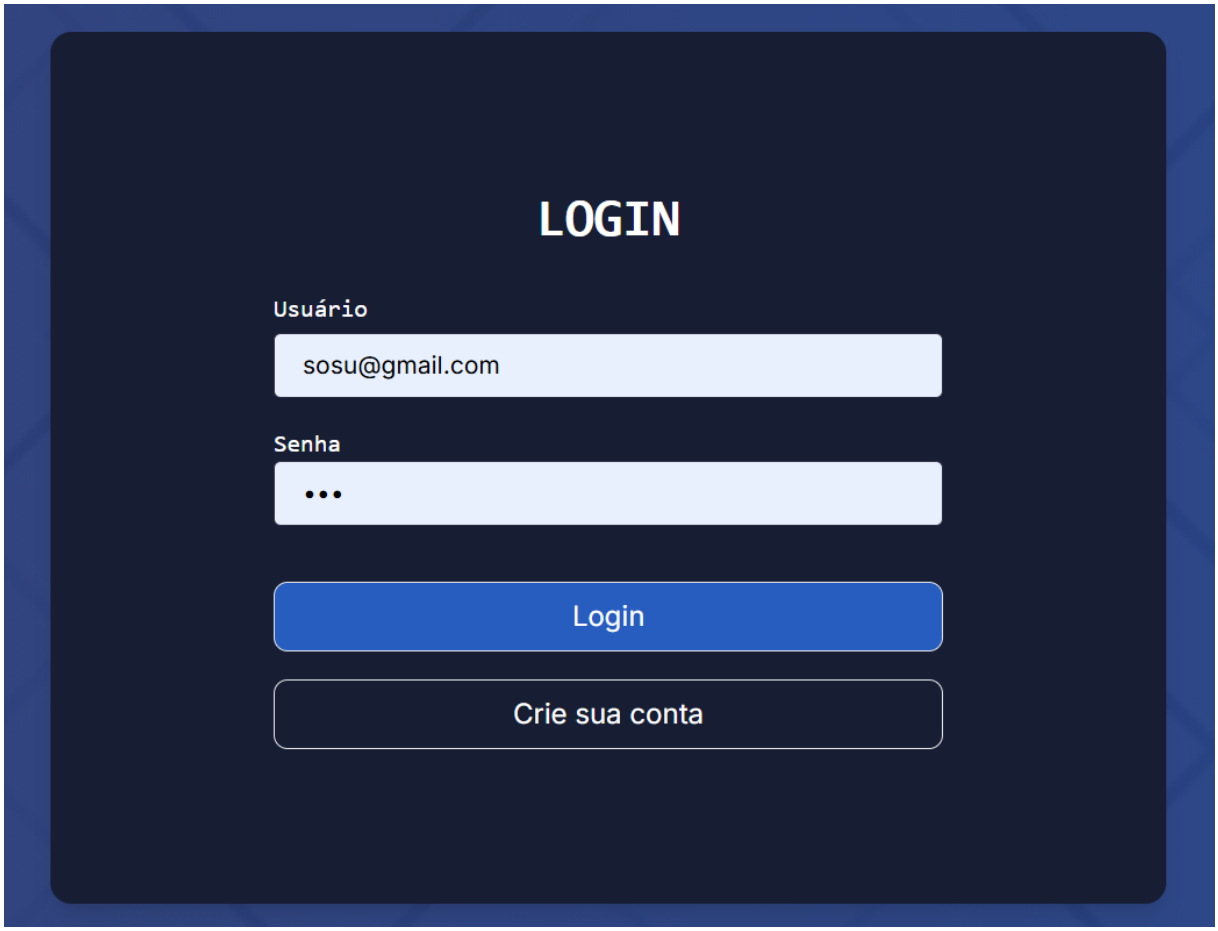
SOROCABA – SP
2025

Sumário

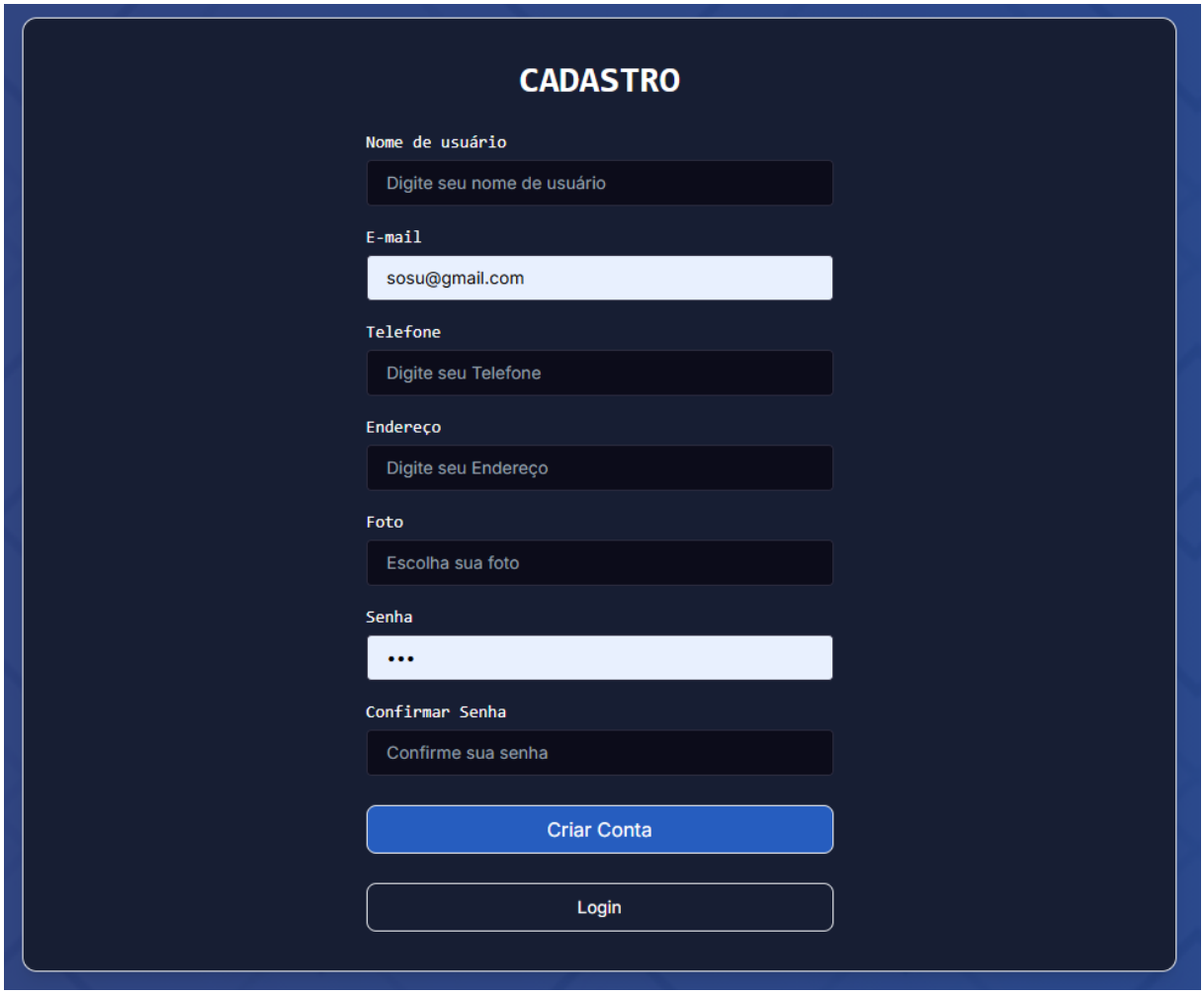
1 – Telas

Inicial

Login



Cadastro

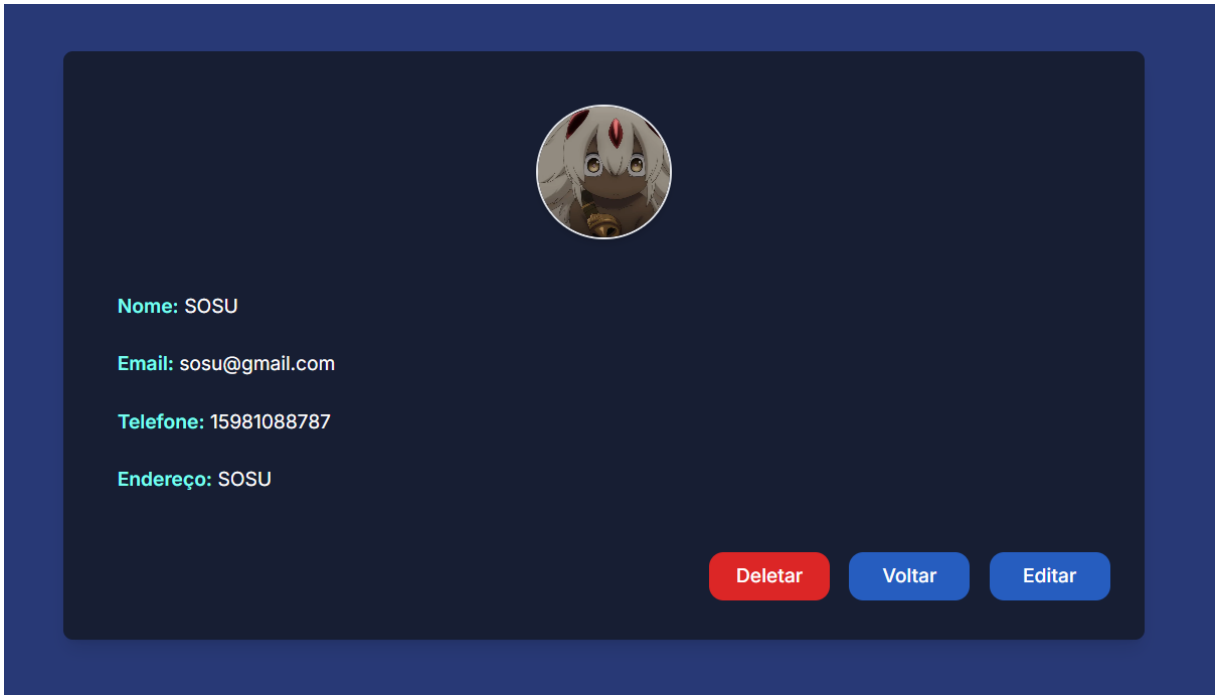

CADASTRO
Nome de usuário
Digite seu nome de usuário
E-mail
sosu@gmail.com
Telefone
Digite seu Telefone
Endereço
Digite seu Endereço
Foto
Escolha sua foto
Senha
Confirmar Senha
Confirme sua senha
Criar Conta
Login

Lista de Usuários

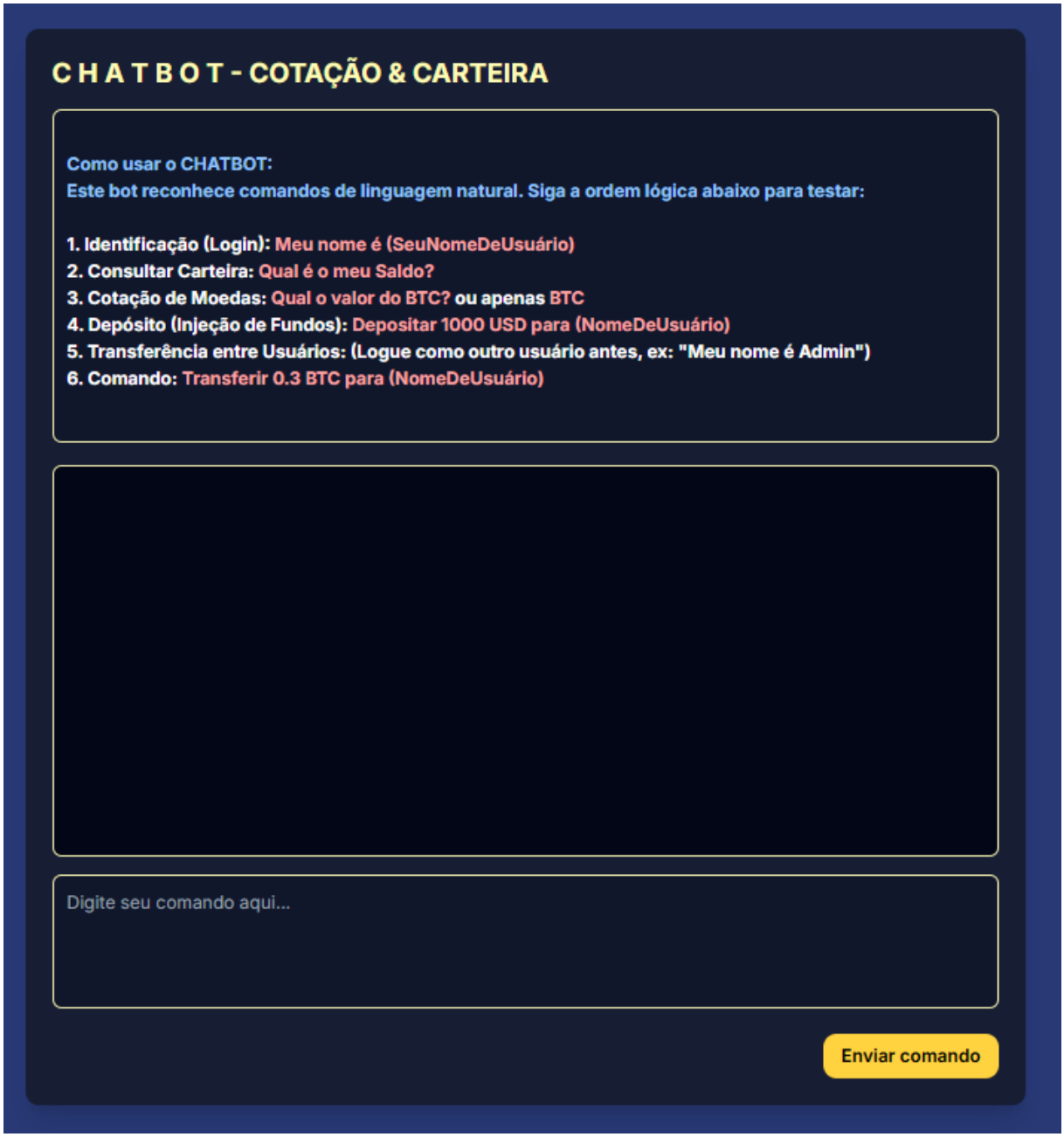
Dashboard

Lista de Moedas

Perfil



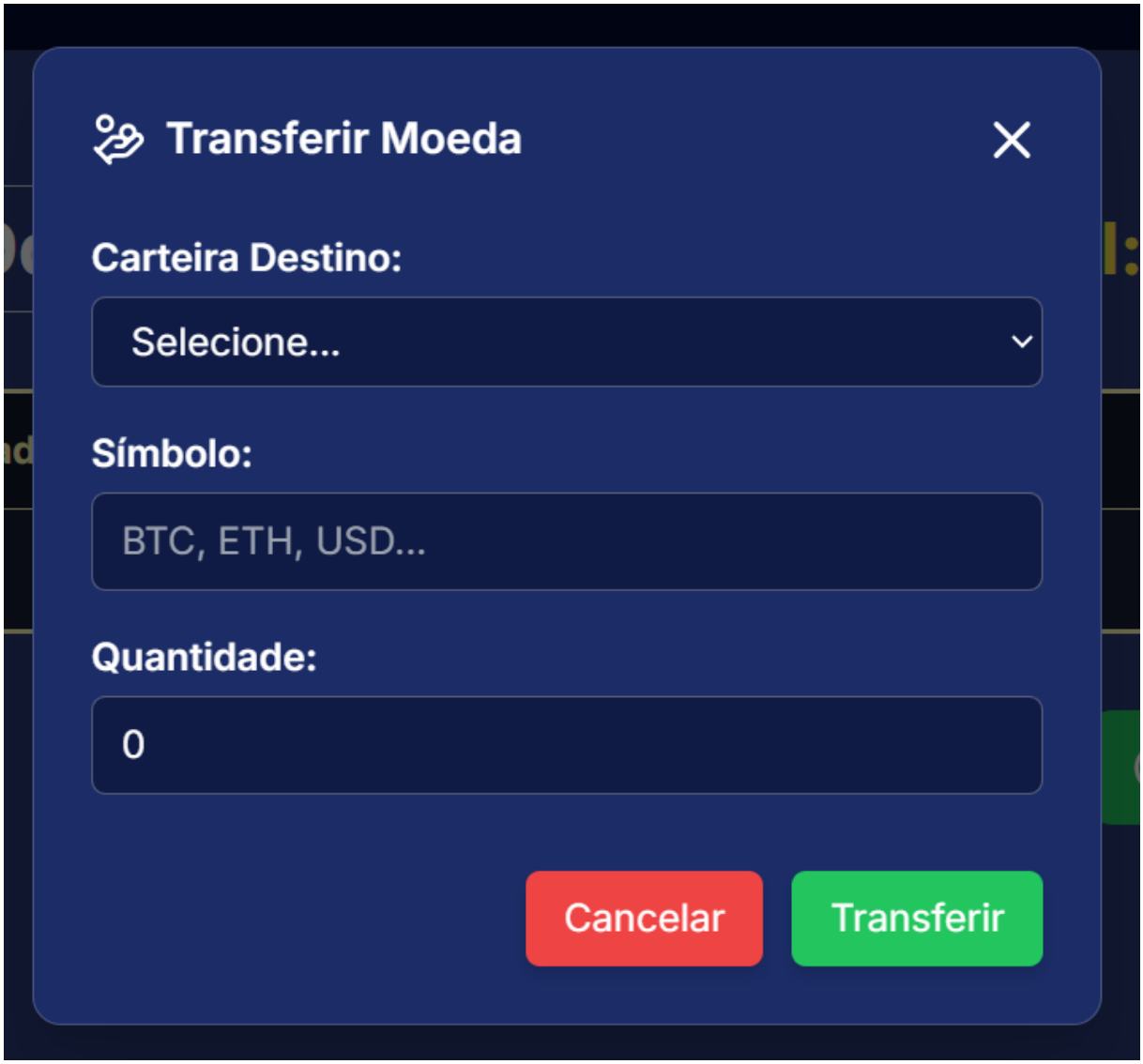
ChatBot


C H A T B O T - COTAÇÃO & CARTEIRA
Como usar o CHATBOT:
Este bot reconhece comandos de linguagem natural. Siga a ordem lógica abaixo para testar:
1. Identificação (Login): Meu nome é (SeuNomeDeUsuário)
2. Consultar Carteira: Qual é o meu Saldo?
3. Cotação de Moedas: Qual o valor do BTC? ou apenas BTC
4. Depósito (Injeção de Fundos): Depositar 1000 USD para (NomeDeUsuário)
5. Transferência entre Usuários: (Logue como outro usuário antes, ex: "Meu nome é Admin")
6. Comando: Transferir 0.3 BTC para (NomeDeUsuário)
Digite seu comando aqui...
Enviar comando

Carteira

Detalhes da Carteira

Transferir Moeda



Adicionar Moeda

2 – Requisitos Funcionais

Login

Tudo que é habitual/ Passkey via sms (desktop) via autenticação facial, digital (mobile).

Home

Exibir o saldo total vindo da conta do usuário/ botão de depósito que adiciona o valor fictício inserido na conta / aba com atualizações tempo real dos ativos mais populares vindos da API.

Trade

Dois campos de seleção de ativos onde você irá selecionar os ativos que você deseja trocar (é preciso que o usuário tenha aquele ativo), botão para efetuar a troca (é feita a comparação de valor entre os dois ativos e o usuário recebe na conta dele o equivalente aquele valor trocado, porém em formato do segundo ativo), gráfico com o valor do ativo nas últimas 24 horas a direita.

Wallets

Tela base que será alterada pelos seus filtros. Todos os filtros e a tela base da wallet (overview, spot, funding) possuem a mesma diagramação, sendo ela o valor total daquela categoria (no overview será o valor de todas as categorias), as opções de depósito (é inserido um valor fictício que passa pra conta do usuário), saque(simplesmente remove aquele valor fictício da conta do usuário), e transferência

para outra carteira(idêntico ao saque), e abaixo a lista de ativos que o usuário possuí e seus valores atuais que devem ser sempre atualizados com base nas transações.

3 – Requisitos Não Funcionais

Desempenho e Tempo de Resposta

O sistema deve ser capaz de processar ordens de negociação, com um tempo de resposta máximo de 1 segundo para exibição de dados e execução de transações.

Escalabilidade

O sistema deve ser capaz de escalar horizontalmente, permitindo a adição de servidores conforme o aumento de tráfego, para garantir o funcionamento contínuo, mesmo durante picos de alta demanda.

Segurança

O sistema deve garantir criptografia de dados sensíveis dos usuários (como senhas, informações financeiras). Além disso, implementar autenticação multifatorial (2FA) para a segurança das contas. (Isso será implementado através de reconhecimento facial/digital no mobile, no desktop será feito por SMS).

Usabilidade

O sistema deve ser responsivo, funcionando adequadamente em diferentes dispositivos (computadores, smartphones, tablets), e fornecer uma interface simples e intuitiva, com uma experiência de usuário que permita realizar operações com facilidade.

Manutenibilidade

O código deve ser bem estruturado, modular e documentado para facilitar a manutenção e as atualizações futuras. Deve-se usar práticas de codificação que permitam que desenvolvedores adicionais possam entender e expandir o sistema de forma eficiente.

Tolerância a Folhas

O sistema deve ser projetado para continuar operando normalmente, mesmo em caso de falhas em alguns componentes. Deve ser implementado um sistema de failover, que redireciona o tráfego para servidores ou sistemas secundários quando necessário.

Compatibilidade

O sistema deve ser compatível com as versões mais recentes dos navegadores mais populares (Chrome, Firefox, Safari, Edge) e dispositivos móveis, garantindo uma experiência uniforme de uso.

4 – Padronização de Commits

Tipo

Um prefixo indicando o tipo da mudança realizada.

Escopo

(Opcional) A área do código afetada pela mudança.

Mensagem

Uma descrição concisa e clara da mudança.

5 – Tipos de Commit

Feat

Nova funcionalidade ou recurso.

Fix

Correção de bug.

Docs

Mudanças na documentação.

Style

Mudanças relacionadas à formatação do código (espaços, quebras de linha etc.), sem afetar o comportamento do código.

Refactor

Refatoração de código, melhorias na estrutura, sem alteração de comportamento.

Perf

Melhoria de desempenho.

Test

Adição ou correção de testes.

Build

Mudanças no sistema de build ou dependências.

Ci

Mudanças no pipeline de integração contínua (ex.: GitHub Actions, Jenkins).

Chore

Tarefas de manutenção ou outras alterações gerais que não se encaixam em outro tipo.

Revert

Reversão de um commit anterior.

6 – Exemplos

Feat - (User - Auth) - Adicionar Autenticação com JWT

Adiciona autenticação via JWT na área de login do sistema.

Fix - (Cart) - Corrigir Erro de Atualização do Valor Total do Carrinho

Corrige um bug onde o valor total do carrinho não era atualizado corretamente após remover um item.

Docs - Atualizar Readme com Instruções de Instalação

Atualiza o Readme com os passos para configurar o ambiente de desenvolvimento.

Style - Ajustar Indentação no Arquivo de Configuração

Alinha a indentação no arquivo config.json para manter consistência no código.

Refactor - (Api) - Melhorar Estrutura das Funções de Login

Refatora a lógica de autenticação para melhorar a legibilidade e manutenibilidade.

Perf - (Database) - Otimizar Consultas SQL

Melhora o desempenho das consultas ao banco de dados, reduzindo o tempo de resposta.

Test - Adicionar Testes para Função de Validação de Dados

Adiciona novos testes unitários para a função de validação de dados.

Ci - Configurar GitHub Actions para Linting e Testes

Configura o fluxo de trabalho de integração contínua no GitHub Actions para incluir linting e execução de testes.

Chore - Atualizar Dependências para Versões mais recentes

Atualiza dependências para as versões mais recentes sem mudanças significativas no código.

Revert - Adicionar Funcionalidade de Pagamento

Reverte o commit que adicionava a funcionalidade de pagamento devido a problemas de integração.

7 – Regras Gerais

Use o Tempo Presente

Utilize sempre verbos no presente (ex.: "adicionar", "corrigir", "refatorar", "melhorar").

Seja Conciso

A mensagem de commit deve ser curta, mas suficientemente clara para descrever a mudança.

Especifique o Escopo

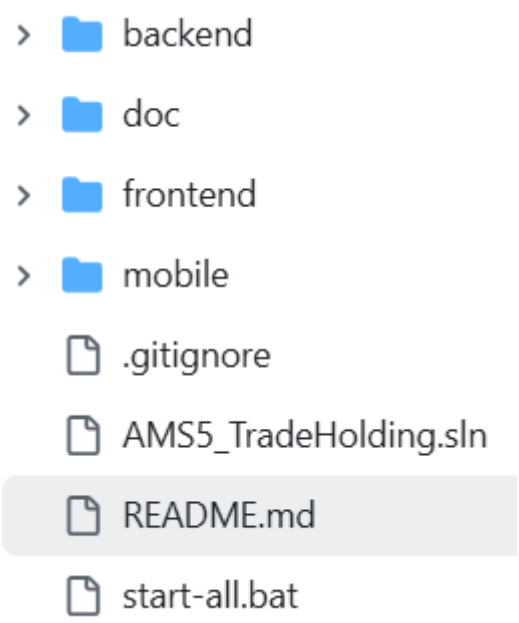
Se a mudança afeta uma área específica do código, inclua o nome dessa área (ex.: feat(auth), fix(ui)).

Quebre Commits Grandes

Evite commits com mudanças grandes e confusas. Divida-os em partes menores e específicas.

```
feat(auth): adicionar login com Google
Adiciona uma funcionalidade de login via Google na aplicação utilizando OAuth2.
```

Obs: Esse README abaixo tá no geral do repositório.



8 – Projeto Corretora de Criptomoedas

9 – Visão Geral do Projeto

O projeto interdisciplinar tem como objetivo o desenvolvimento de uma aplicação multiplataforma inspirada em corretoras de criptomoedas, como a Binance.

O sistema deve permitir autenticação de usuários, exibição de carteiras e ativos, simulação de transações (depósitos, saques e trocas) e interação via chatbot, integrando diferentes tecnologias e conceitos de sistemas distribuídos.

A arquitetura baseia-se em microserviços independentes, com comunicação síncrona via API Gateway (REST) e comunicação assíncrona via RabbitMQ (mensageria).

Principais tecnologias utilizadas:

- Backend: .NET (C#) com arquitetura limpa (Clean Architecture)
- Frontend Web: Next.js + TypeScript + Tailwind CSS
- Mobile: React Native com Expo
- Mensageria: RabbitMQ
- Chatbot: Flask (Python)
- Banco de dados: SQLite (nas APIs)
- Controle de versão: GitHub com fluxo GitFlow

10 – Arquitetura da Aplicação

10.1 – Estrutura Geral

O sistema é composto por cinco APIs principais e um API Gateway.

API	Função principal
UserAPI	Cadastro, autenticação e controle de acesso dos usuários.
WalletAPI	Controle de carteiras, saldos e transações.
CurrencyAPI	Consulta de cotações e histórico de preços de ativos.
ChatbotAPI	Atendimento automatizado e comandos de interação.
GatewayAPI	Camada intermediária entre o frontend/mobile e os microserviços.

10.2 – Componentes de Frontend e Mobile

- Frontend (Next.js + Tailwind CSS): interface principal de acesso às funcionalidades, exibindo saldo, ativos, histórico e chat.
- Mobile (React Native + Expo): versão simplificada e otimizada para acesso rápido a saldo e transações.

10.3 – Comunicação entre Componentes

- Síncrona (REST/HTTP): comunicação direta entre frontend e backend via API Gateway.
- Assíncrona (RabbitMQ): troca de mensagens entre microserviços (ex: eventos de depósito, trade, atualização de preços).

11 – Estrutura de Pastas e Clean Architecture

A arquitetura segue o padrão Clean Architecture, garantindo separação de responsabilidades, testabilidade e facilidade de manutenção.

11.1 – Comunicação entre Componentes

Estrutura Base de um Microserviço

```
/userAPI
  /API
    /Controllers
    /DTOs
    /Configurations
    Program.cs
  /Domain
    /Entities
    /Interfaces
  /Infrastructure
    /Data
    /Repositories
    /Migrations
  /Application
    /Services
    /Interfaces
    /UseCases
```

11.2 – Camadas da Arquitetura

1. API (Apresentação) Exposição de endpoints REST. Exemplo: UserController.cs responde às requisições de cadastro e login.

2. Domain (Domínio) Contém as entidades e interfaces base do negócio. Exemplo: User.cs, IUserRepository.cs.

3. Infrastructure (Infraestrutura e Persistência) gerencia o banco de dados e implementa os repositórios. Exemplo: UserRepository.cs, UserDbContext.cs.

4. Application (Casos de Uso e Regras de Negócio) implementa a lógica de aplicação, serviços e casos de uso. Exemplo: RegisterUserUseCase.cs, UserService.cs.

12 – Arquitetura da Aplicação

A autenticação é feita com JWT (JSON Web Token), garantindo que apenas usuários autenticados acessem rotas protegidas.

12.1 – Dependências utilizadas

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package BCrypt.Net-Next
```

12.2 – Fluxo de Autenticação

1. O usuário realiza login enviando e-mail e senha.
2. A API valida as credenciais e gera um token JWT.
3. O frontend armazena o token e o envia no cabeçalho das próximas requisições.
4. O token é validado em cada chamada por middleware de autenticação.

12.3 – Exemplo de Login

Request

```
POST /user/login
{
  "email": "user@example.com",
  "password": "123456"
}
```

Response

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI...",
  "mfaRequired": true,
  "mfaType": "sms"
}
```

12.4 – Hash de Senhas

Na criação do usuário, a senha deve ser armazenada de forma segura:

```
var hashedPassword = BCrypt.Net.BCrypt.HashPassword(userDto.Password);
```

13 – Comunicação entre Serviços

Os microserviços se comunicam de duas formas:

13.1 – Comunicação Síncrona (REST via GatewayAPI)

Usada em fluxos que exigem resposta imediata.

Exemplos:

- Login (POST /user/login)
- Consulta de saldo (GET /wallet/balance)
- Execução de trade (POST /wallet/trade)
- Envio de mensagem ao chatbot (POST /chatbot/message)

Exemplo de resposta resumida:

```
{
  "tradeId": "t98765",
  "status": "SUCCESS",
  "newBalances": { "BTC": 0.5, "USDT": 13750.20 }
}
```

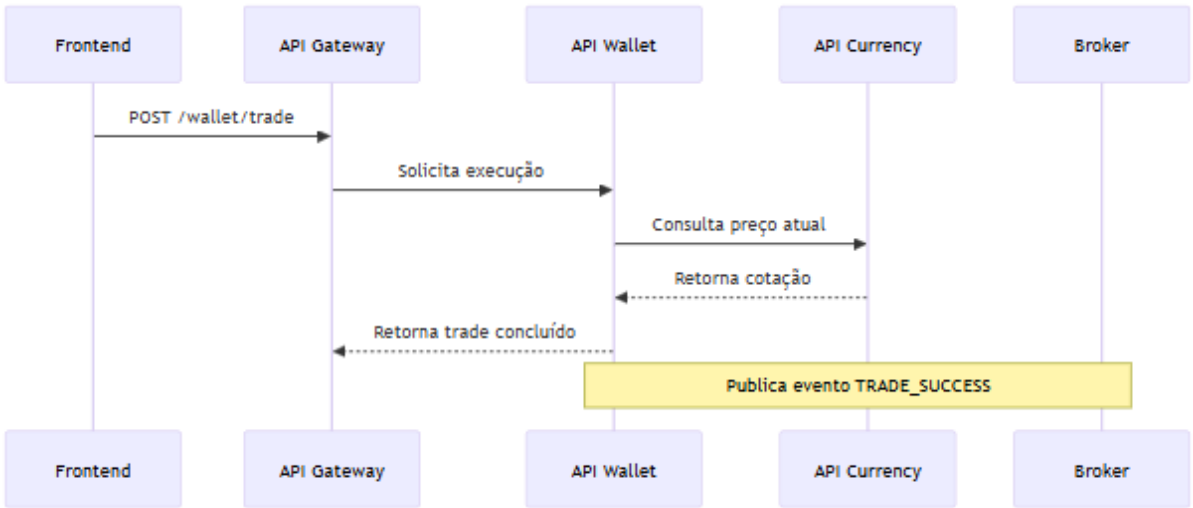
13.2 – Comunicação Assíncrona (RabbitMQ)

Usada para eventos internos entre serviços, sem depender de resposta imediata.

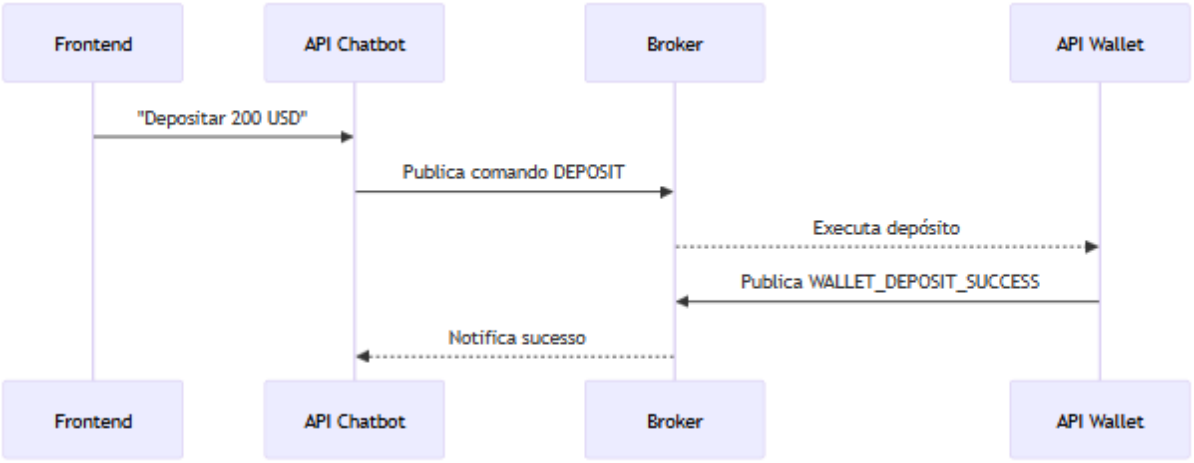
Principais Exchanges e eventos:

Origem	Evento	Destino	Ação
UserAPI	`user.auth.success`	Wallet, Chatbot	Registro de login concluído
WalletAPI	`wallet.deposit.success`	User, Chatbot	Notificação de depósito
CurrencyAPI	`currency.price.update`	Wallet, Chatbot	Atualização de preços
ChatbotAPI	`chatbot.wallet.deposit`	Wallet	Comando de depósito via chat

13.3 – Fluxo de Trade de Ativos



13.4 – Fluxo de Depósito via Chatbot



14 – Requisitos Funcionais e Não Funcionais (detalhado)

14.1 – Requisitos Funcionais (RF)

Cada RF está numerado para referência nas entregas e testes.

RF-01 — Cadastro de usuário

- Permitir que um usuário crie conta com nome, email e senha.
- Validar unicidade de email.
- Armazenar senha hasheada (BCrypt).

RF-02 — Autenticação

- Login via email + senha.
- Emissão de JWT com validade configurável.
- Suporte a MFA (campo mfaRequired no payload). Implementação mínima: simulação com sms flag (não é preciso integrar SMS real; pode ser flag simulada para entrega).

RF-03 — Consulta de saldo

- Exibir saldo total e saldos por carteira (spot, funding, overview).
- Endpoint: GET /wallet/balance?userId={id} (autenticado).

RF-04 — Simulação de depósito/saque

- Executar depósito fictício em carteira selecionada.
- Publicar evento de depósito (wallet.deposit.success) no broker.
- Endpoint: POST /wallet/deposit.

RF-05 — Trade (troca de ativos)

- Permitir troca entre dois ativos (ex: BTC → USDT) com verificação de saldo.
- Consultar preço atual na CurrencyAPI.
- Registrar transação e publicar evento wallet.trade.success.

RF-06 — Exibição de ativos populares e gráfico histórico

- GET /currency/list — listar ativos populares.
- GET /currency/history?symbol={} — retornar históricos (últimas 24h, 7d) para plotagem no frontend.

RF-07 — Chatbot (comandos e consultas)

- POST /chatbot/message — responde consultas (ex: "Qual meu saldo?") e aceita comandos simulados (ex: "Depositar 200 USD").
- Chatbot publica comandos no broker para ações que afetam outras APIs.

RF-08 — Perfil do usuário

- Endpoint para leitura/atualização de dados não sensíveis (nome, preferências).
- Endpoint protegido para upload de avatar (opcional, armazenar referência).

RF-09 — Histórico e extrato

- GET /wallet/history?userId={} — lista de transações (depósitos, saques, trades).

RF-10 — Logs e auditoria (mínimo)

- Registrar operações sensíveis (login, trade, depósito) em log acessível para avaliação (arquivo ou tabela simples).

14.2 – Requisitos Não Funcionais (RNF)

RNF-01 — Desempenho

- Tempo de resposta médio para endpoints principais (/wallet/balance, /currency/price) ≤ 1s em ambiente de testes local/avaliativo.

RNF-02 — Segurança

- Senhas armazenadas com bcrypt.
- JWT com assinatura HMAC (ou RSA se preferirem) e expiração.
- Rotas protegidas por middleware de autenticação.
- Validação de entrada (sanitização) em todos os endpoints.

RNF-03 — Escalabilidade

- Arquitetura baseada em microserviços; componentes desacoplados via RabbitMQ.
- Configurações para permitir execução em contêineres (Docker) — containers separados por serviço.

RNF-04 — Manutenibilidade

- Padrão de camadas (Clean Architecture).
- Documentação mínima por serviço: README com endpoints, dependências e rota de execução local.

RNF-05 — Observabilidade

- Logs estruturados (pelo menos console + arquivo).
- Métricas simples (contadores de requisições e latência) ou prints no ambiente de avaliação.

RNF-06 — Portabilidade / Deploy

- Aplicações devem rodar localmente com dotnet run (APIs) e npm run dev (frontend) e, preferivelmente, via Docker Compose para avaliação integrada.

RNF-07 — Tolerância a falhas

- Em caso de falha de um consumidor RabbitMQ, eventos devem ser reencaminhados / mantidos na fila (configuração básica de retry/ack).

14.3 – Modelos de Dados (resumo mínimo)

User

```
{
  "id": "u123",
  "name": "André Souza",
  "email": "andre@example.com",
  "passwordHash": "<bcrypt>",
  "createdAt": "2025-09-01T12:00:00Z"
}
```

WalletBalance

```
{
  "userId": "u123",
  "totalBalance": 15200.50,
  "currency": "USD",
  "wallets": [
    { "type": "spot", "balance": 8500.00 },
    { "type": "funding", "balance": 4700.50 },
    { "type": "overview", "balance": 2000.00 }
  ]
}
```

Trade

```
{
  "tradeId": "t98765",
  "userId": "u123",
  "fromAsset": "BTC",
  "toAsset": "USDT",
  "amount": 1.0,
  "executedPrice": 27500.40,
  "status": "SUCCESS",
  "timestamp": "2025-09-19T15:05:00Z"
}
```

15 – Definição do MVP (Produto Mínimo Viável) — critérios, escopo e checklist de aceitação

15.1 – Objetivo do MVP

Entregar um conjunto mínimo de funcionalidades completo e integrável que permita demonstrar o fluxo principal de uma corretora simulada: autenticar usuário, exibir saldo, executar depósito/ trade e receber confirmação via chat ou eventos.

15.2 – Escopo mínimo do MVP (must-have)

Os itens abaixo são obrigatórios para considerar o MVP funcional:

MVP-01 — Autenticação

- POST /user/register
- POST /user/login → retorno de JWT
- Middleware de proteção de rotas no GatewayAPI

MVP-02 — Consulta de saldo

- GET /wallet/balance autenticado

MVP-03 — Depósito simulado

- POST /wallet/deposit atualiza saldo e publica wallet.deposit.success no RabbitMQ

MVP-04 — Trade funcional

- POST /wallet/trade valida saldo, consulta preço em CurrencyAPI e atualiza saldos
- Publica evento wallet.trade.success

MVP-05 — CurrencyAPI básico

- GET /currency/price?symbol={} retorna preço atual (pode ser dados simulados atualizados a cada X segundos)

MVP-06 — Chatbot básico

- POST /chatbot/message com respostas a consulta de saldo e comando Depositar {valor} USD (que publica chatbot.wallet.deposit)

MVP-07 — Frontend mínimo

- Páginas: Login, Home (saldo), Trade (formulário simplificado), Chat (UI para testar chatbot)

MVP-08 — Documentação mínima

- README geral com instruções de execução do sistema integrado e script de demonstração

15.3 – Critérios de Aceitação do MVP (detalhados)

Para cada item do MVP existem critérios que definem aprovação:

Autenticação

- Ao registrar e logar, o usuário recebe token JWT válido.
- Rota protegida retorna 401 quando JWT ausente ou inválido.

Consulta de saldo

- Requisição autenticada retorna totalBalance consistente com depósitos/trades realizados na sessão de teste.

Depósito

- POST /wallet/deposit atualiza saldo e, ao consultar /wallet/balance, o valor refletido deve conter o depósito.
- Evento wallet.deposit.success é publicado (verificado por consumer simples ou log).

Trade

- Trade recusa se saldo insuficiente (retorno HTTP 400 e mensagem clara).
- Em caso de sucesso, saldos atualizados e evento wallet.trade.success publicado.

Chatbot

- Mensagem "Qual meu saldo?" retorna resposta textual com o saldo atual (usando dados da WalletAPI).
- Comando "Depositar 200 USD" gera evento chatbot.wallet.deposit e resulta em alteração de saldo após processamento.

Frontend

- Usuário consegue autenticar, ver saldo, abrir trade e enviar mensagens ao chatbot (apesar de UI simples).

15.4 – Endpoints mínimos recomendados (resumo)

- POST /user/register
- POST /user/login
- GET /wallet/balance
- POST /wallet/deposit
- POST /wallet/trade
- GET /currency/price
- GET /currency/history (opcional básico)
- POST /chatbot/message

15.5 – Métricas para considerar o MVP aceitável (para avaliação)

- Funcionalidade: 100% dos endpoints MVP respondendo conforme critérios de aceitação.
- Integração: Eventos básicos publicados e consumidos (depósito e trade).
- Documentação: README com passos de execução e script de demo.
- Usabilidade: Frontend navegável para demonstração (login → saldo → deposit → trade → chat).
- Robustez: Tratamento básico de erros (400/401/500 com mensagens claras).

15.6 – Checklist técnico para entrega do MVP (para subir no repositório)

- Repositório principal com README geral.
- Subpastas / repositórios por serviço com README e instruções de execução.
- Arquivo docker-compose.yml (opcional, mas recomendado) contendo: RabbitMQ, UserAPI, WalletAPI, CurrencyAPI, ChatbotAPI e Frontend (ou instruções para rodar localmente).
- Script ou documento DEMO.md com os passos para apresentação (ex.: criar usuário → login → depositar → executar trade → verificar eventos).
- Mapeamento de dependências e versões no README (ex.: dotnet SDK version, Node version).
- Rota de execução rápida: comandos exatos para executar cada serviço localmente.
- Branch final nomeada (ex.: deliver/mvp) ou tag com a versão entregue.

15.7 – Riscos conhecidos e mitigação (curto)

- Risco: Falta de tempo para integrar RabbitMQ. Mitigação: Implementar publicação em log e simular consumo; documentar onde o consumo real deve acontecer.
- Risco: Dados de preço real não disponíveis. Mitigação: Usar dataset estático ou gerador de preços simulados (script que atualiza preços a cada X segundos).
- Risco: Problemas de autenticação cross-service. Mitigação: Padronizar validação de JWT a partir de um secret/keystore compartilhado no .env de avaliação.

16 – Padrão de Documentação Exigido

Cada grupo deverá manter um padrão de documentação técnica unificado entre os microserviços, seguindo boas práticas de repositórios profissionais. A documentação faz parte da nota final e será avaliada em conjunto com o código e a apresentação.

Além dos arquivos de documentação no formato Markdown (.md), cada grupo deverá gerar uma documentação técnica adicional em PDF, nomeada:

```
Documentacao_Tecnica_Projeto_Corretora.pdf
```

Esse documento deverá conter os diagramas essenciais do projeto, acompanhados de uma breve explicação textual de cada um, com o objetivo de evidenciar o entendimento técnico do grupo sobre a arquitetura e o funcionamento do sistema.

Elementos obrigatórios da documentação técnica em PDF

1. Diagrama de Arquitetura Geral do Sistema Deve representar graficamente a visão macro da aplicação, destacando:

- O Frontend (Next.js) e o aplicativo mobile (React Native) como camadas de interface.
- O API Gateway, responsável por intermediar o tráfego entre os clientes e os microserviços.
- Os microserviços principais (UserAPI, WalletAPI, CurrencyAPI, ChatbotAPI), com suas responsabilidades.
- O mecanismo de mensageria (RabbitMQ), indicando os fluxos assíncronos de eventos.
- A persistência de dados (bancos SQLite ou outros) associada a cada serviço. O diagrama deve deixar evidente como ocorre a comunicação entre os componentes (REST e eventos) e como as camadas se integram.

2. Diagramas de Classes (UML) — por microserviço cada microserviço deverá possuir um diagrama de classes UML simplificado, contendo:

- Entidades (models): classes que representam as tabelas do domínio (ex.: User, Wallet, Currency).
- Serviços (services): classes que implementam regras de negócio (ex.: UserService, WalletService).
- Repositórios (repositories): classes responsáveis pela persistência e acesso a dados.
- Interfaces (contracts): definindo os métodos esperados de serviços e repositórios.

Cada diagrama deve conter:

- Atributos principais de cada classe (sem necessidade de todos os tipos).
- Relacionamentos entre classes (associações, dependências, heranças, etc.).
- Uma breve legenda explicando as camadas (Application, Domain, Infrastructure).

3. DER — Diagrama Entidade-Relacionamento (por API) cada microserviço com persistência própria deve apresentar seu modelo de dados relacional:

- Identificação das entidades (tabelas) e seus atributos essenciais.
- Indicação das chaves primárias e estrangeiras.
- Relacionamentos entre entidades (1:N, N:N, 1:1).
- Cardinalidades e dependências entre dados (ex.: um User pode ter várias Wallets; uma Wallet pertence a um único User).

O DER deve refletir o modelo efetivamente implementado nas classes de entidade do domínio.

4. Diagramas de Sequência — fluxos principais os diagramas de sequência devem representar as interações entre componentes durante os fluxos centrais do sistema. Fluxos obrigatórios:

- Login → validação do usuário e emissão de token JWT.
- Depósito → solicitação REST, processamento na WalletAPI e publicação de evento.
- Trade → interação entre WalletAPI e CurrencyAPI para conversão de ativos.
- Depósito via Chatbot → comando recebido no ChatbotAPI, publicado no RabbitMQ e processado pela WalletAPI.

Cada diagrama deve apresentar:

- As entidades participantes (Frontend, Gateway, APIs, RabbitMQ).
- As mensagens trocadas (requisições REST, eventos, respostas).
- O resultado final esperado (ex.: saldo atualizado, confirmação de trade).

5. Fluxo de Comunicação entre Serviços (REST e RabbitMQ) Representar de forma consolidada as rotas síncronas (HTTP) e eventos assíncronos (RabbitMQ) do sistema. O diagrama pode combinar setas diretas (REST) e setas tracejadas (eventos). Deve ilustrar:

- Principais endpoints utilizados entre serviços (ex.: /wallet/trade, /currency/price).
- Exchanges e routing keys usadas no RabbitMQ (ex.: wallet.events, chatbot.commands).
- Direção das mensagens e dependências entre os microserviços.

6. Descrição textual dos componentes e tecnologias após os diagramas, incluir uma seção descritiva com:

- O papel de cada componente (ex.: GatewayAPI, UserAPI, WalletAPI etc.).
- A tecnologia utilizada em cada camada (linguagem, frameworks, banco de dados, mensageria).
- As principais decisões de design adotadas (ex.: uso de Clean Architecture, comunicação assíncrona, token JWT).
- Versões e dependências mais relevantes.

Ela servirá como complemento visual e explicativo da documentação em Markdown, sendo obrigatória para avaliação final. Todos os diagramas devem ser originais, elaborados pelo grupo (pode-se utilizar ferramentas como Draw.io, Lucidchart, PlantUML ou Mermaid).

16.1 – Estrutura Geral de Repositório

Cada grupo poderá usar:

- 1 repositório monolítico (monorepo) com subpastas /userAPI, /walletAPI, /currencyAPI, etc. ou
- Vários repositórios (um por microserviço) vinculados a uma organização no GitHub.

Em ambos os casos, é obrigatório incluir:

```
ProjetoCorretora/
│
├── README.md                    # Documentação geral do sistema
├── Documentacao_Tecnica_Projeto_Corretora.pdf  # Documento técnico com diagramas
├── docker-compose.yml           # Opcional, mas recomendado
├── /userAPI/                    # Serviço de autenticação
│   ├── README.md
│   └── ...
├── /walletAPI/                  # Serviço de carteiras
│   ├── README.md
│   └── ...
├── /currencyAPI/                # Serviço de cotações
│   ├── README.md
│   └── ...
├── /chatbotAPI/                 # Serviço do chatbot
│   ├── README.md
│   └── ...
├── /frontend/                   # Aplicação web (Next.js)
│   ├── README.md
│   └── ...
└── DEMO.md                      # Passo a passo de execução e apresentação
```

16.2 – Estrutura mínima de cada README.md de serviço

Cada serviço deve conter:

1. Identificação

```
# UserAPI
Gerencia usuários e autenticação via JWT.
```

2. Stack e dependências

```
- Linguagem: C# (.NET 8)
- Banco: SQLite
- Mensageria: RabbitMQ
- Autenticação: JWT (Microsoft.AspNetCore.Authentication.JwtBearer)
```

3. Instruções de execução local

```
dotnet restore
dotnet run
```

ou, se via Docker:

```
docker build -t userapi .
docker run -p 8080:8080 userapi
```

4. Endpoints principais

Método	Endpoint	Descrição
POST	/user/register	Criação de novo usuário
POST	/user/login	Retorna JWT
GET	/user/profile	Perfil do usuário autenticado

5. Exemplos de Requisição/Resposta

```
POST /user/login
{
  "email": "teste@example.com",
  "password": "123456"
}
```

Response:

```
{
  "token": "eyJh...abc",
  "expiresIn": 3600
}
```

6. Integrações com outros serviços

- Publica evento user.auth.success no RabbitMQ.
- Consumido por WalletAPI e ChatbotAPI.

7. Observações / Known Issues

Anotar limitações conhecidas ou endpoints simulados.

16.3 – Documentação geral do sistema (README.md da raiz)

O documento principal do projeto deve conter:

1. Visão geral da aplicação e arquitetura.
2. Serviços existentes e suas funções.
3. Como executar o projeto completo (localmente ou via Docker Compose).
4. Fluxos principais de uso (Login → Depósito → Trade → Chatbot).
5. Integrantes do grupo e responsabilidades.
6. Versões das tecnologias (dotnet SDK, Node, etc.).
7. Descrição do ambiente de testes e instruções para o avaliador.

Exemplo de seção de execução integrada:

```
# Execução via Docker Compose
docker-compose up -d
```

16.4 – Documento de Demonstração (DEMO.md)

Deve conter o roteiro da apresentação final, com os comandos e passos de teste na ordem esperada. Exemplo:

```
## Passo 1 – Registro e login
POST /user/register
POST /user/login → copiar token JWT

## Passo 2 – Consultar saldo inicial
GET /wallet/balance

## Passo 3 – Realizar depósito
POST /wallet/deposit { "amount": 1000 }

## Passo 4 – Executar trade
POST /wallet/trade { "from": "BTC", "to": "USDT", "amount": 0.5 }

## Passo 5 – Chatbot
POST /chatbot/message "Qual meu saldo?"
```

16.5 – Padrão de Commits e Branches

Para manter o histórico organizado e rastreável, adotar convenção semelhante a:

```
feat(auth): adicionar autenticação JWT
fix(wallet): corrigir cálculo de saldo após depósito
docs(readme): atualizar instruções de execução
refactor(api): reorganizar estrutura de diretórios
```

Branch principal: main Branchs de desenvolvimento: feature/, fix/, hotfix/, release/ Branch final para entrega: deliver/mvp ou main com tag v1.0-final.

16.6 – Padrão de Avaliação Técnica da Documentação

Durante a correção, serão observados:

Critério	Peso	Descrição
Estrutura de pastas coerente	1.0	Pastas e serviços bem organizados
README por serviço	1.0	Instruções e endpoints claros
README geral e DEMO.md	1.0	Documentação do sistema completo
Histórico de commits claros	0.5	Histórico descritivo e padronizado
Fluxos e comandos testáveis	0.5	Comandos reproduzíveis no ambiente do avaliador

Pontuação total (documentação): 4,0 pontos dentro da nota global.

17 – Critérios de Avaliação Final

A avaliação final será composta pela entrega técnica (código e documentação) e pela apresentação prática do MVP, conforme rubrica a seguir.

17.1 – Estrutura de Avaliação

Dimensão	Peso	Descrição
1. Funcionalidade (execução)	3,0 pts	APIs e frontend funcionando conforme MVP, com fluxo principal executável.
2. Arquitetura e qualidade do código	2,0 pts	Aplicação modular, camadas respeitando Clean Architecture, boas práticas de C# e JS.
3. Integração e mensageria (RabbitMQ)	1,5 pts	Publicação e/ou consumo de eventos entre microserviços, mesmo que simulada.
4. Documentação técnica	2,0 pts	READMEs completos, comandos funcionais, descrição do sistema clara.
5. Apresentação e domínio do grupo	1,5 pts	Clareza na explicação do projeto, roteiro coerente, respostas técnicas seguras.

Total: 10,0 pontos

17.2 – Descritivo dos níveis de desempenho

Excelente (9–10):

- Sistema completo, todos endpoints MVP funcionando.
- Integração RabbitMQ funcional ou simulada e documentada.
- Documentação clara e reproduzível.
- Apresentação segura e domínio dos conceitos de arquitetura, comunicação e autenticação.

Bom (7–8):

- MVP funcional com pequenas falhas isoladas.
- RabbitMQ implementado parcialmente ou via logs.
- Documentação suficiente, mas com lacunas menores.
- Boa compreensão técnica do grupo.

Regular (5–6):

- MVP parcialmente implementado.
- Falhas em integração ou inconsistência de dados.
- Documentação incompleta.
- Apresentação superficial ou leitura excessiva de roteiro.

Insuficiente (≤4):

- Sistema não executa o fluxo principal completo (login → saldo → trade).
- Falta de documentação ou código fora do padrão.
- Dificuldade em explicar arquitetura ou decisões de projeto.

17.3 – Requisitos obrigatórios para avaliação

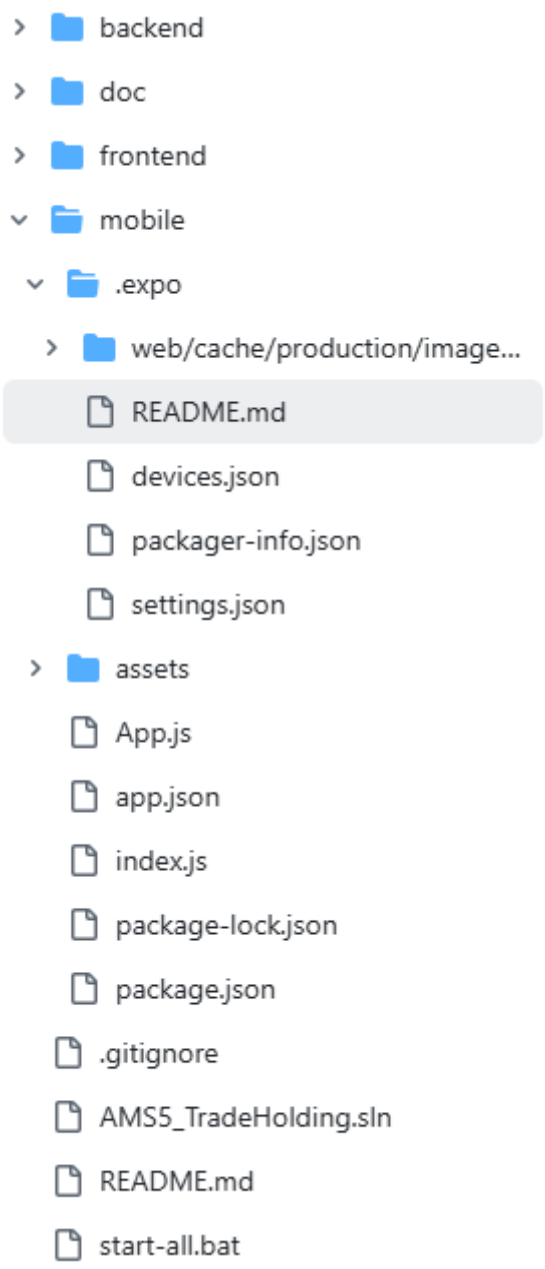
Para que o projeto seja avaliado, todos os itens abaixo devem estar presentes:

- Repositório acessível e público ou compartilhado com a coordenação.
- README geral + READMEs por serviço.
- MVP executável (localmente ou via Docker Compose).
- Passo a passo de demonstração (arquivo DEMO.md).
- Código-fonte entregue até a data limite.
- Apresentação oral com tempo máximo de 10 a 12 minutos.

17.4 – Recomendações finais aos alunos

- Priorize funcionalidade e integração antes da interface.
- Documente cada endpoint testado (exemplo de request e response).
- Em caso de falha em um serviço, explique tecnicamente na apresentação (não omita).
- Mantenha logs ativados e visíveis na execução (para que o avaliador veja eventos e chamadas REST).
- Prepare ambiente limpo para a demonstração (exemplo: containers zerados, banco ressetado).

Obs: Esse README abaixo é da Pasta mobile que tá dentro da Pasta .expo.



Por que tenho uma pasta chamada ".expo" no meu projeto?

A pasta ".expo" é criada quando um projeto Expo é iniciado usando o comando "expo start".

O que os arquivos contêm?

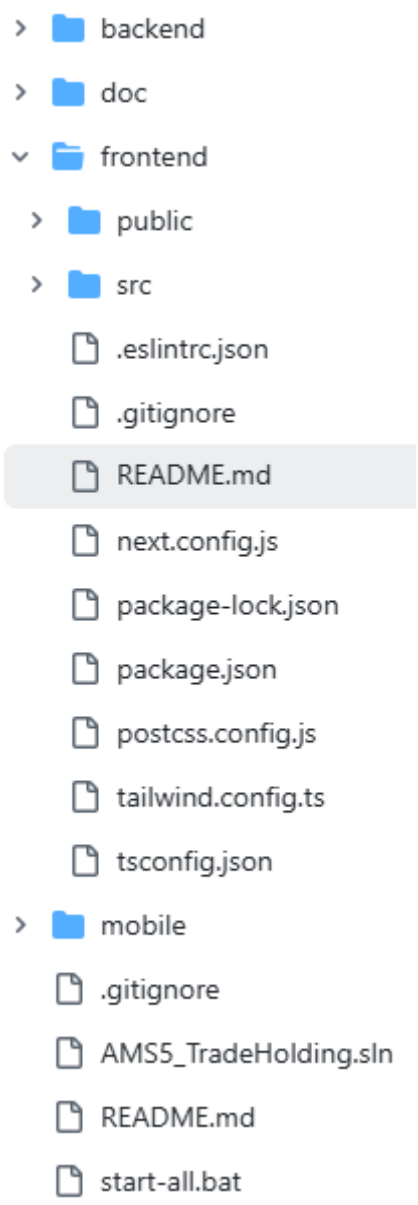
- "devices.json": contém informações sobre os dispositivos que abriram este projeto recentemente. Ele é usado para preencher a lista "Sessões de desenvolvimento" nas suas versões de desenvolvimento.
- "packager-info.json": contém os números de porta e os PIDs dos processos usados para servir o aplicativo ao dispositivo móvel/simulador.
- "settings.json": contém a configuração do servidor usada para servir o manifesto do aplicativo.

Devo adicionar a pasta ".expo" ao meu repositório?

Não, você não deve compartilhar a pasta ".expo". Ela não contém nenhuma informação relevante para outros desenvolvedores que trabalham no projeto, sendo específica para a sua máquina.

Após a criação do projeto, a pasta ".expo" já é adicionada ao seu arquivo ".gitignore".

Obs: Esse README abaixo é da Pasta frontend.



Este é um projeto Next.js inicializado com create-next-app.

Primeiros passos

Primeiro, execute o servidor de desenvolvimento:

```
npm run dev
# or
yarn dev
# or
pnpm dev
# or
bun dev
```

Abra http://localhost:3000 com seu navegador para ver o resultado.

Você pode começar a editar a página modificando o arquivo app/page.tsx. A página é atualizada automaticamente conforme você edita o arquivo.

Este projeto usa next/font para otimizar e carregar automaticamente a fonte Inter, uma fonte personalizada do Google.

Saiba mais

Para saber mais sobre o Next.js, consulte os seguintes recursos:

- Documentação do Next.js - saiba mais sobre os recursos e a API do Next.js.
- Aprenda Next.js - um tutorial interativo do Next.js.

Você pode conferir o repositório do Next.js no GitHub - seus comentários e contribuições são bem-vindos!

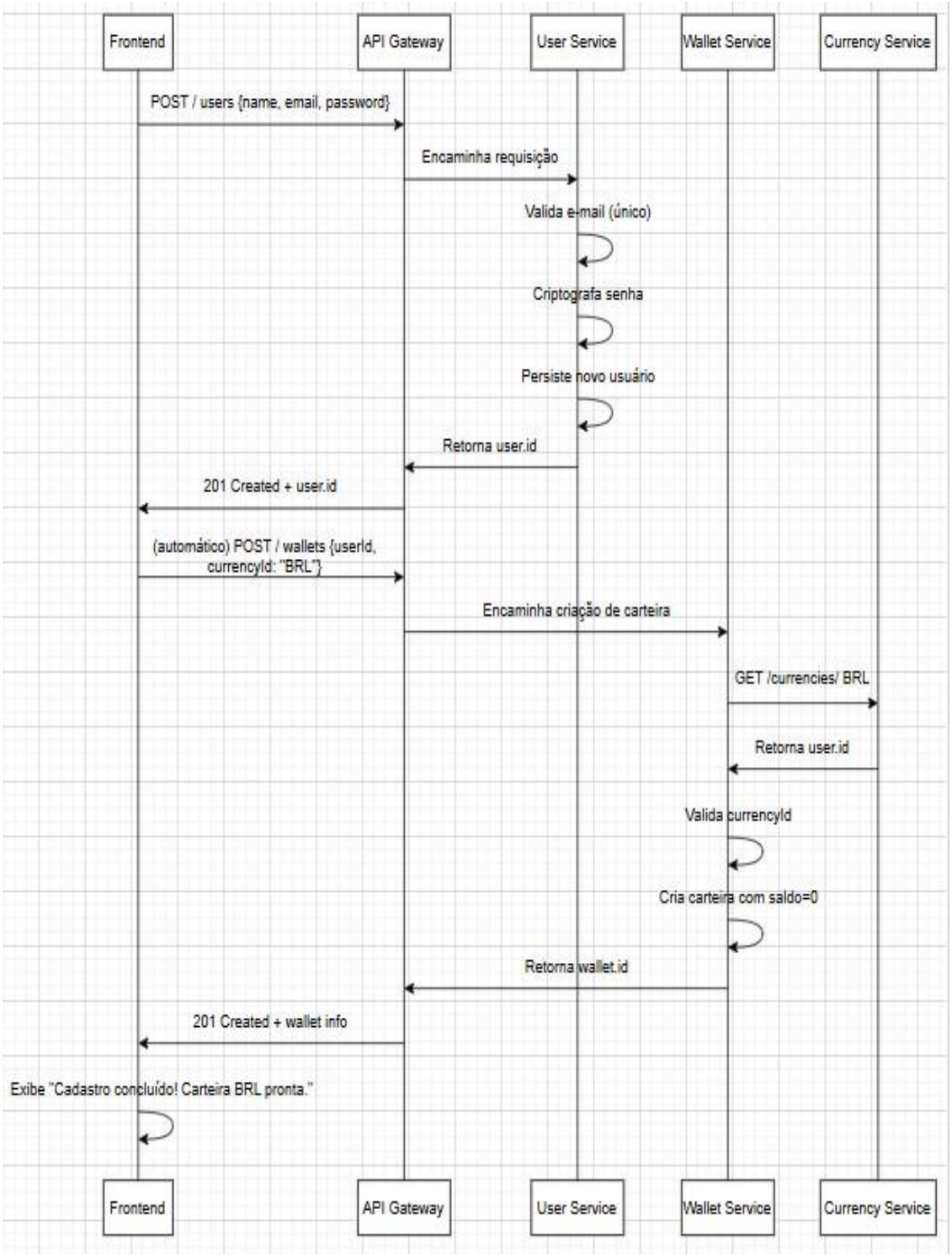
Implantação no Vercel

A maneira mais fácil de implantar seu aplicativo Next.js é usar a plataforma Vercel, dos criadores do Next.js.

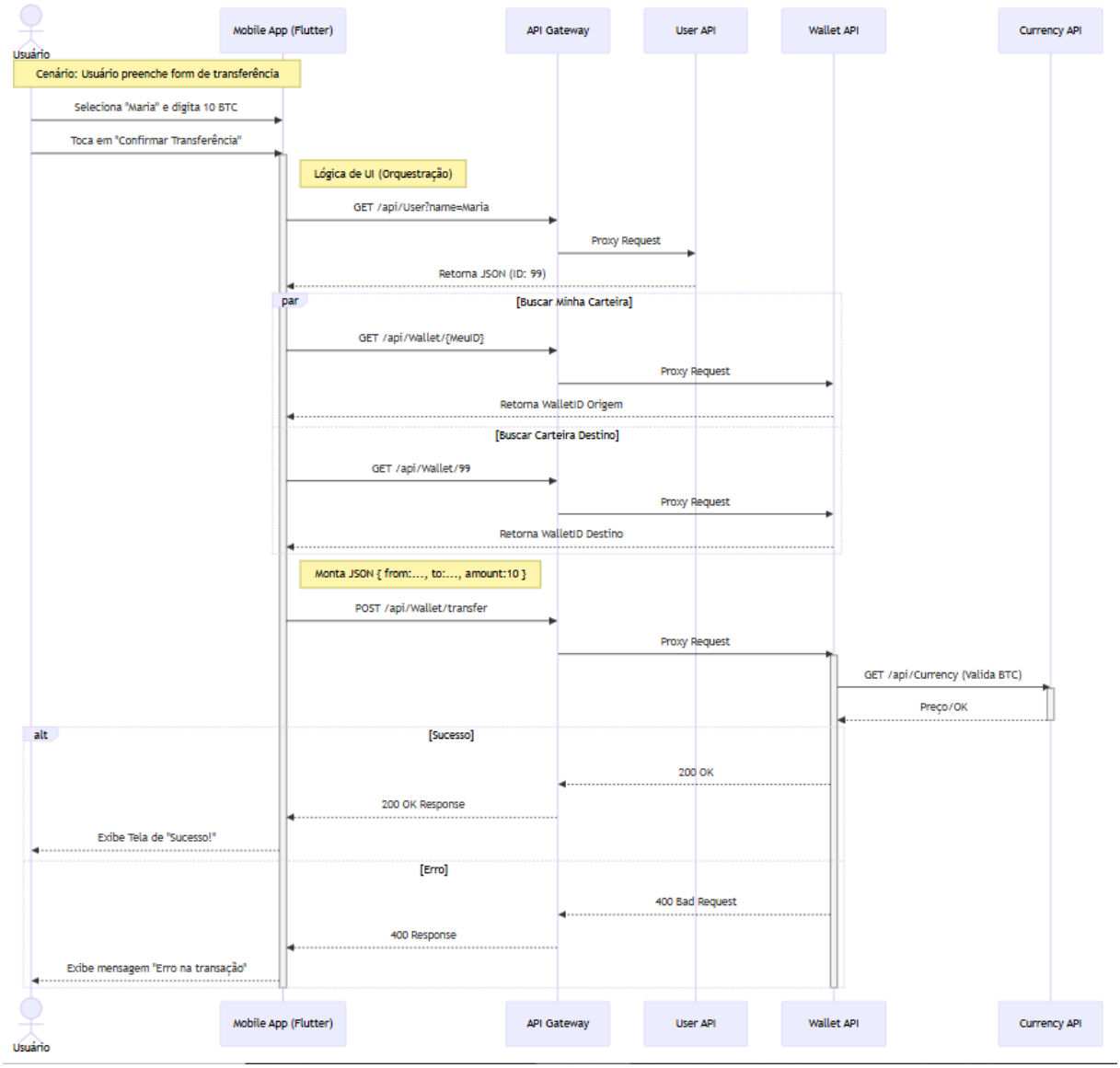
Confira nossa documentação de implantação do Next.js para obter mais detalhes.

18 – Diagrama de Sequência

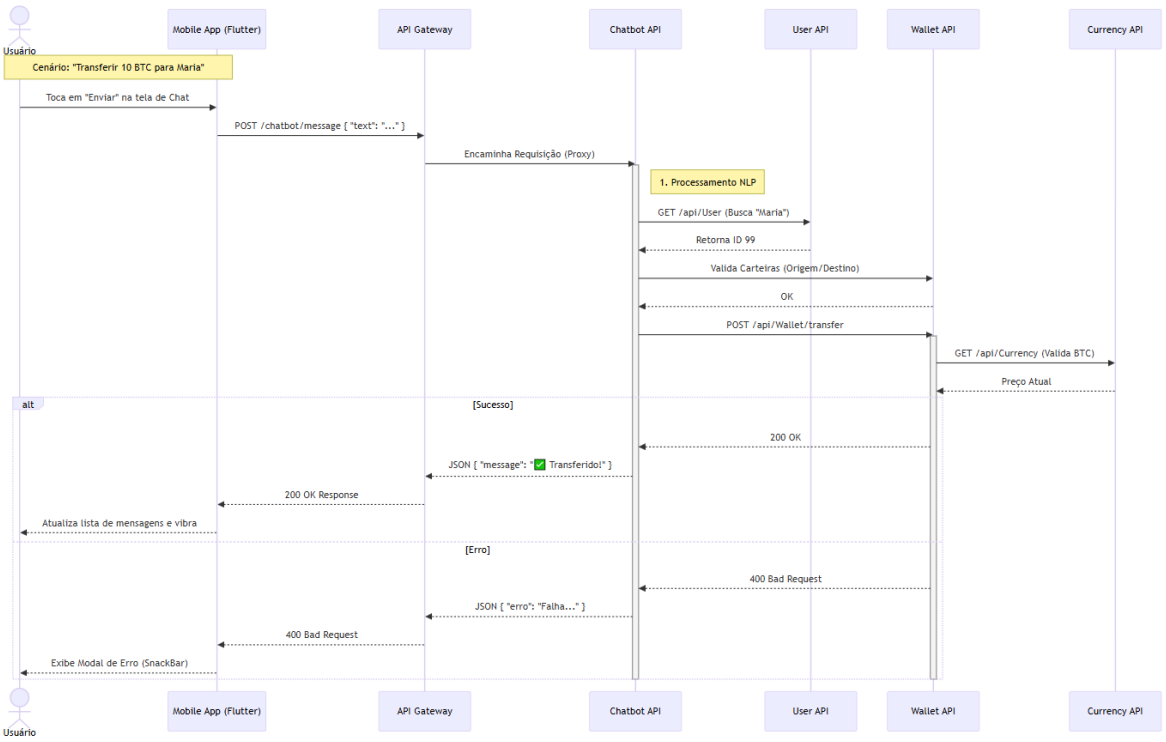
Cenário com Frontend sem ChatBot



Cenário com Mobile sem ChatBot



Cenário com Frontend com ChatBot



Cenário com Mobile com ChatBot