Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas - AMS

Davi Ryan Konuma Lima
Kaique Alves de Mesquita
Lucas Feitosa Almeida Rocha
Luiz Filipe de Camargo
Matheus Henrique Schopp Peixoto

Documentação Técnica de Desenvolvimento de Software

Binance

Sorocaba
Dezembro – 2025

Davi Ryan Konuma Lima
Kaique Alves de Mesquita
Lucas Feitosa Almeida Rocha
Luiz Filipe de Camargo
Matheus Henrique Schopp Peixoto

Documentação Técnica de Desenvolvimento de Software

Binance

Projeto final apresentado à Faculdade de Tecnologia de Sorocaba, como parte dos pré-requisitos para finalização de disciplina Programação Multiplataforma do curso de Análise e Desenvolvimento de Sistemas do programa de Articulação do Ensino Médio e Superior do Centro Paula Souza.

Orientador: Prof. André Cassulino

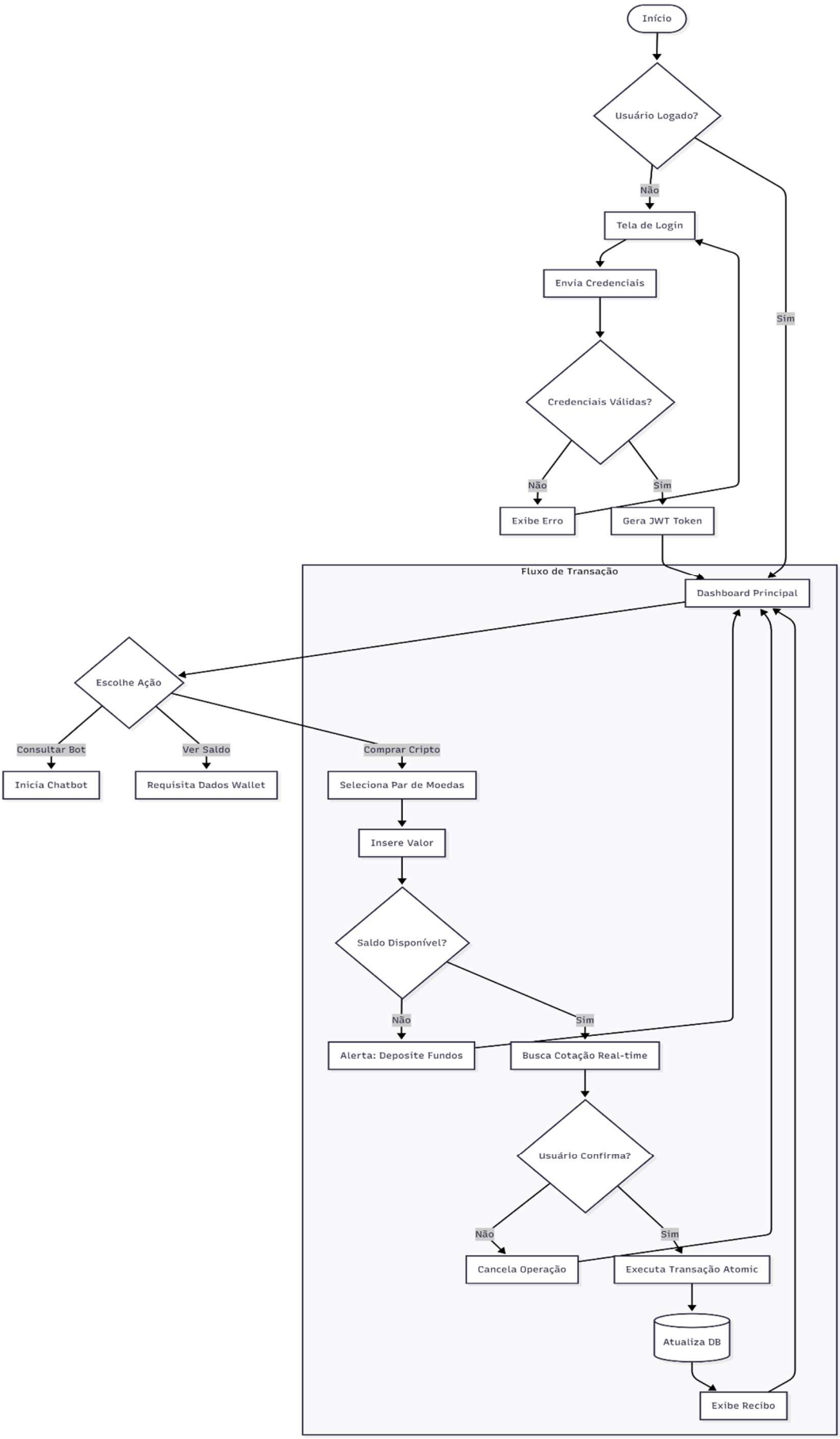
Sorocaba
Dezembro – 2025

Sumário

1. Diagrama de Atividades

Este diagrama modela o fluxo comportamental da aplicação, focando na jornada do usuário desde o acesso até a conclusão de uma transação.

1. **Autenticação:** O sistema verifica se o usuário está logado. Caso contrário, exige credenciais e, se válidas, gera um token JWT para sessão.
2. **Dashboard e Escolhas:** O usuário pode optar por consultar o Chatbot, visualizar Saldo ou realizar Compras.
3. **Fluxo de Transação (Trade):** Ao selecionar "Comprar Cripto", o sistema verifica a disponibilidade de saldo. Se houver fundos, busca a cotação em tempo real e solicita confirmação.
4. **Conclusão:** A operação é executada de forma atômica (tudo ou nada) no banco de dados, garantindo integridade financeira antes de exibir o recibo.

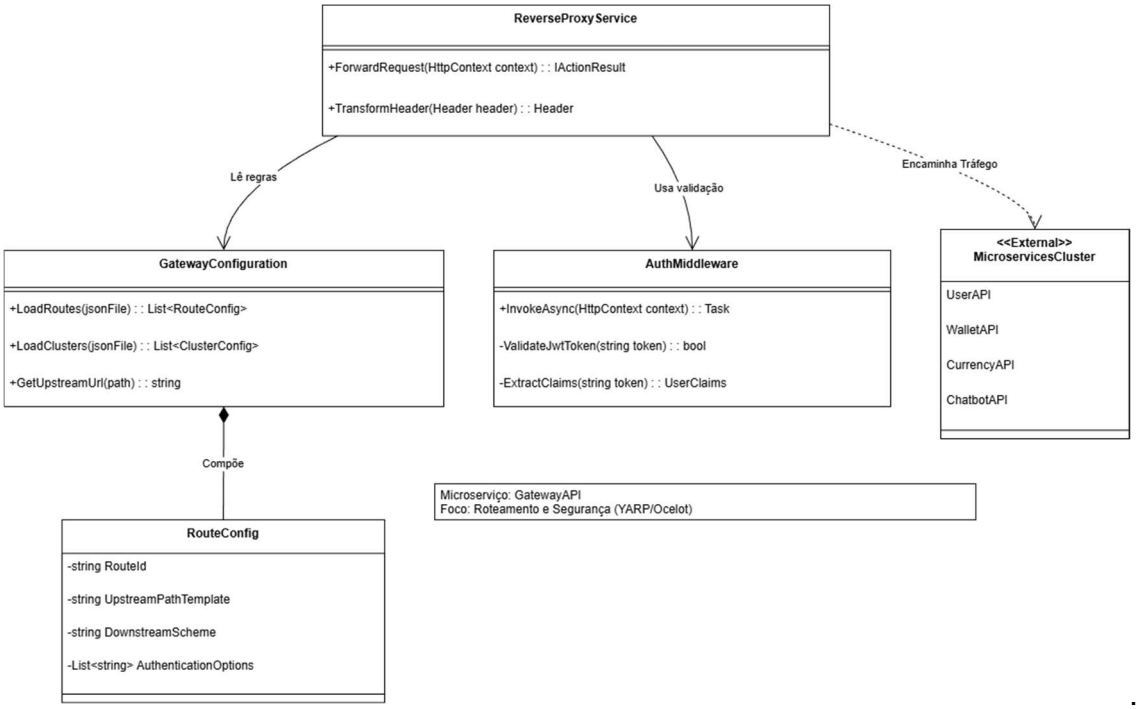

Início
Usuário Logado?
Não
Sim
Tela de Login
Envia Credenciais
Credenciais Válidas?
Não
Sim
Exibe Erro
Gera JWT Token
Fluxo de Transação
Dashboard Principal
Escolhe Ação
Consultar Bot
Ver Saldo
Comprar Cripto
Inicia Chatbot
Requisita Dados Wallet
Seleciona Par de Moedas
Insere Valor
Saldo Disponível?
Não
Sim
Alerta: Deposite Fundos
Busca Cotação Real-time
Usuário Confirma?
Não
Sim
Cancela Operação
Executa Transação Atomic
Atualiza DB
Exibe Recibo

2. Diagrama de Classes

O diagrama de classes detalha a arquitetura orientada a objetos do backend, organizada em camadas de responsabilidade:

2.1. GatewayAPI (Orquestrador de Tráfego)

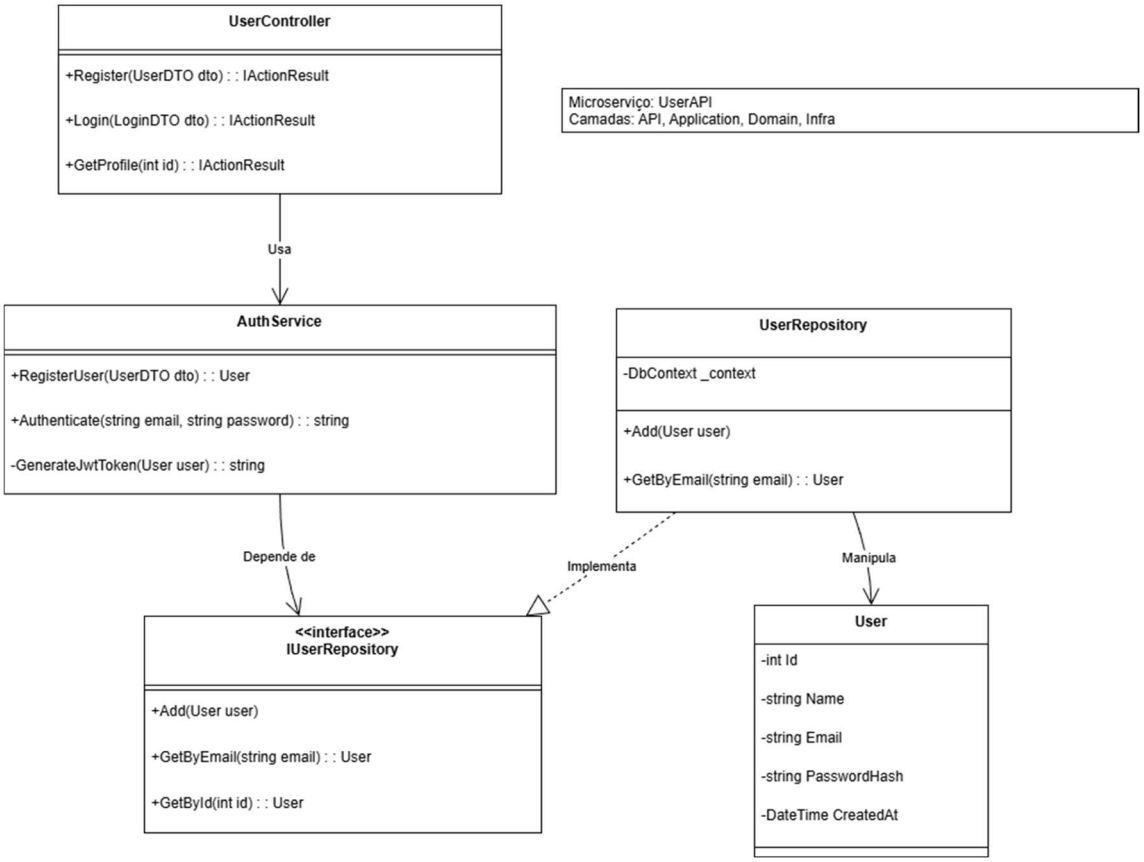
Atua como a barreira de entrada da infraestrutura. Sua função é interceptar todas as chamadas externas, verificar as credenciais de acesso (segurança) e distribuir o tráfego para os serviços de backend apropriados, ocultando a complexidade interna da rede



.

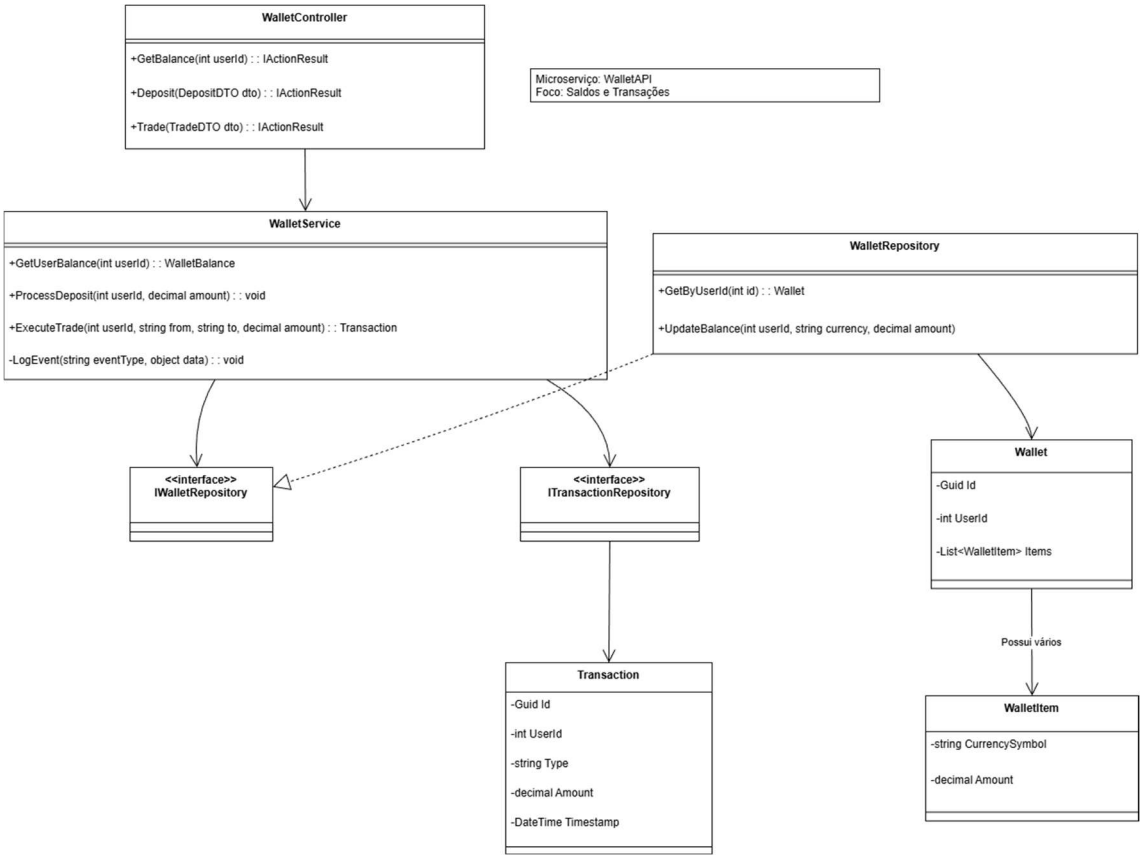
2.2 UserAPI (Gestão de Identidade)

Responsável pela administração das contas. Este componente cuida do registro de novos membros, valida as credenciais durante o acesso (Login) e emite as chaves digitais (Tokens) necessárias para que o usuário navegue de forma segura na plataforma.



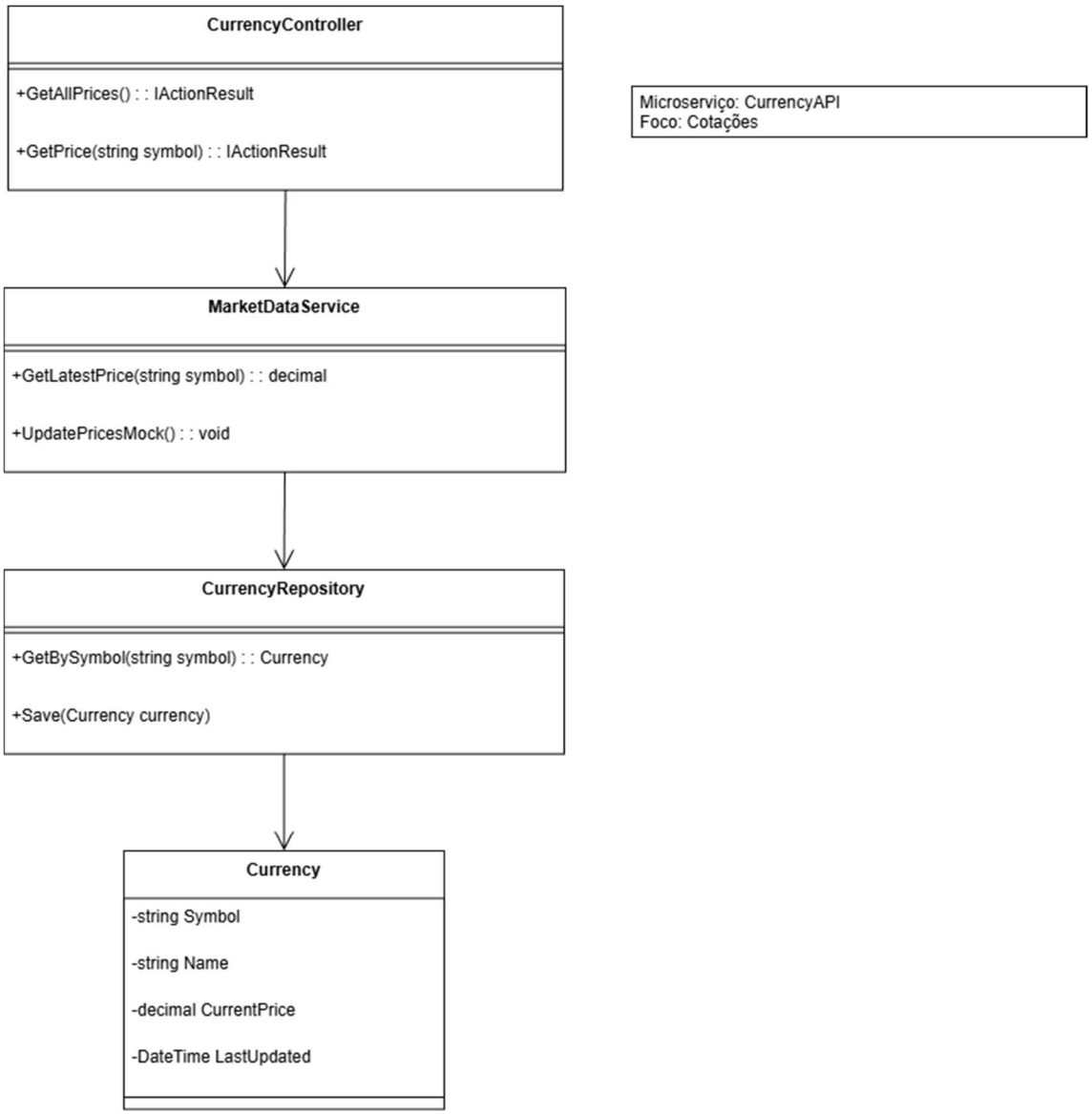
2.3 WalletAPI (Núcleo Financeiro)

O motor contábil do sistema. É encarregado de manter o livro-razão (ledger) atualizado, processar a entrada de recursos (depósitos) e executar a lógica de conversão entre ativos (trades), assegurando que nenhum saldo seja gasto duas vezes.



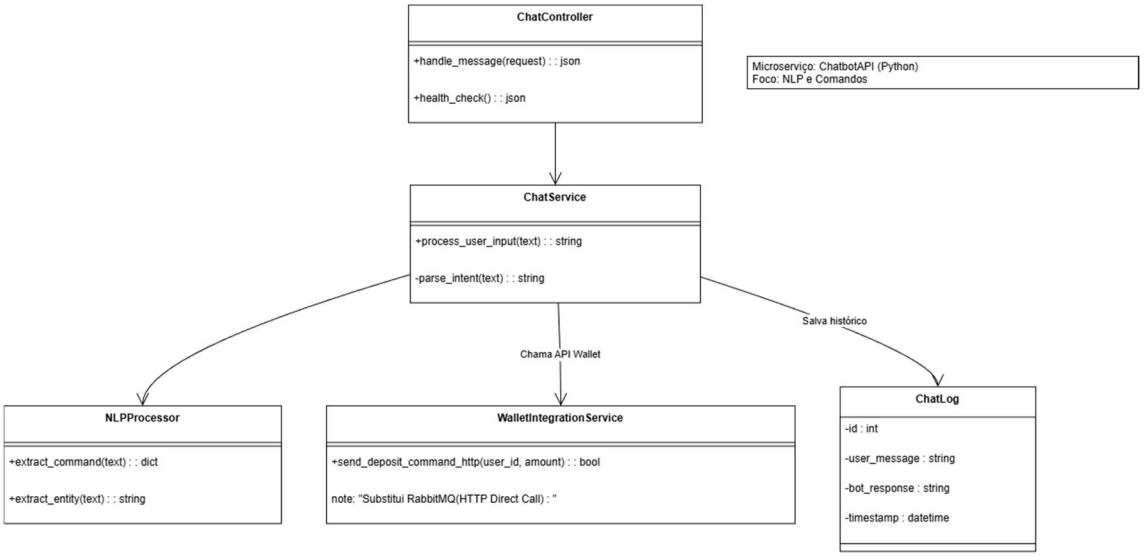
2.4 CurrencyAPI (Catálogo de Ativos)

Serviço dedicado à inteligência de mercado. Ele monitora e disponibiliza o valor em tempo real das criptomoedas, além de fornecer o histórico de flutuação de preços para a construção de gráficos analíticos na interface do usuário.



2.5 ChatbotAPI (Automação Inteligente)

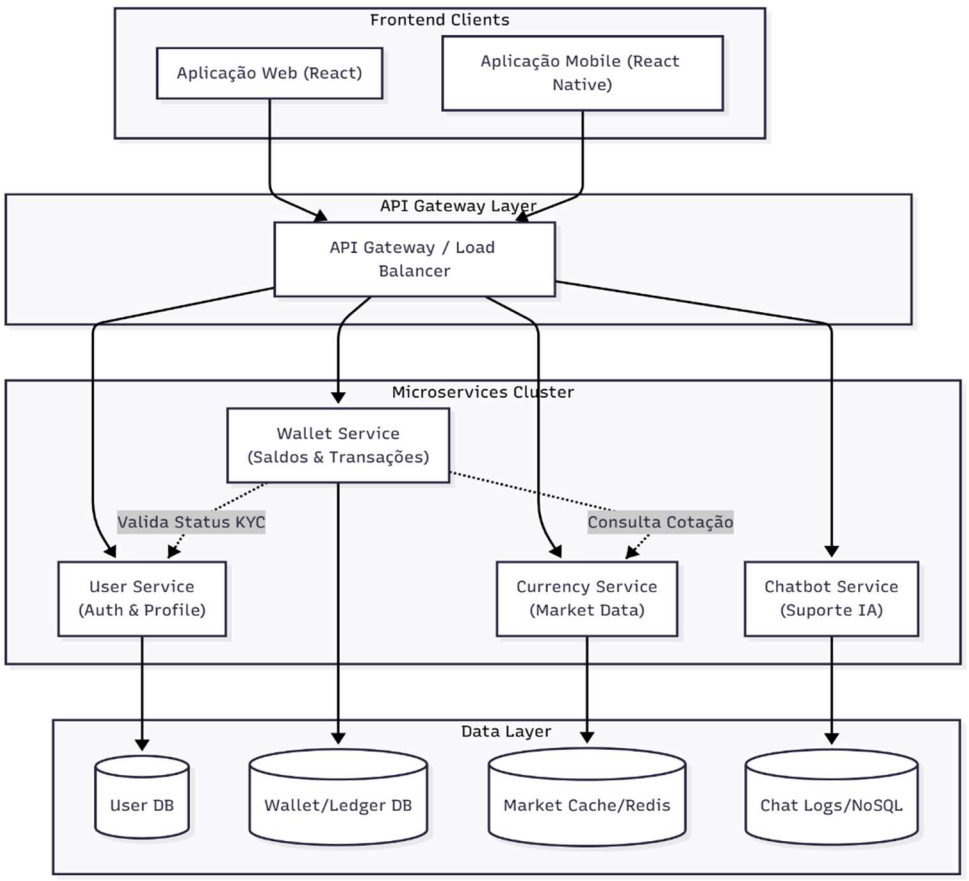
Interface de atendimento baseada em processamento de linguagem natural. Este serviço analisa as mensagens de texto enviadas pelo usuário para identificar intenções — como consultar valores ou ordenar operações — e executa essas tarefas automaticamente.



3. Diagrama de Componentes

Este diagrama oferece uma visão de alto nível das dependências e da comunicação entre os módulos do sistema.

- **Frontend Clients:** Aplicações Web (React) e Mobile (React Native) consomem a mesma interface.
- **API Gateway Layer:** Centraliza as requisições, realizando o roteamento para os microsserviços apropriados.
- **Microservices Cluster:** Mostra as interações diretas. Note que o Wallet Service (Saldos) depende do User Service para validar status KYC (Know Your Customer) e do Currency Service para obter cotações atuais.
- **Data Layer:** Evidencia a separação de dados, onde cada serviço possui acesso ao seu contexto de armazenamento (User DB, Wallet/Ledger DB, Market Cache), garantindo o desacoplamento.



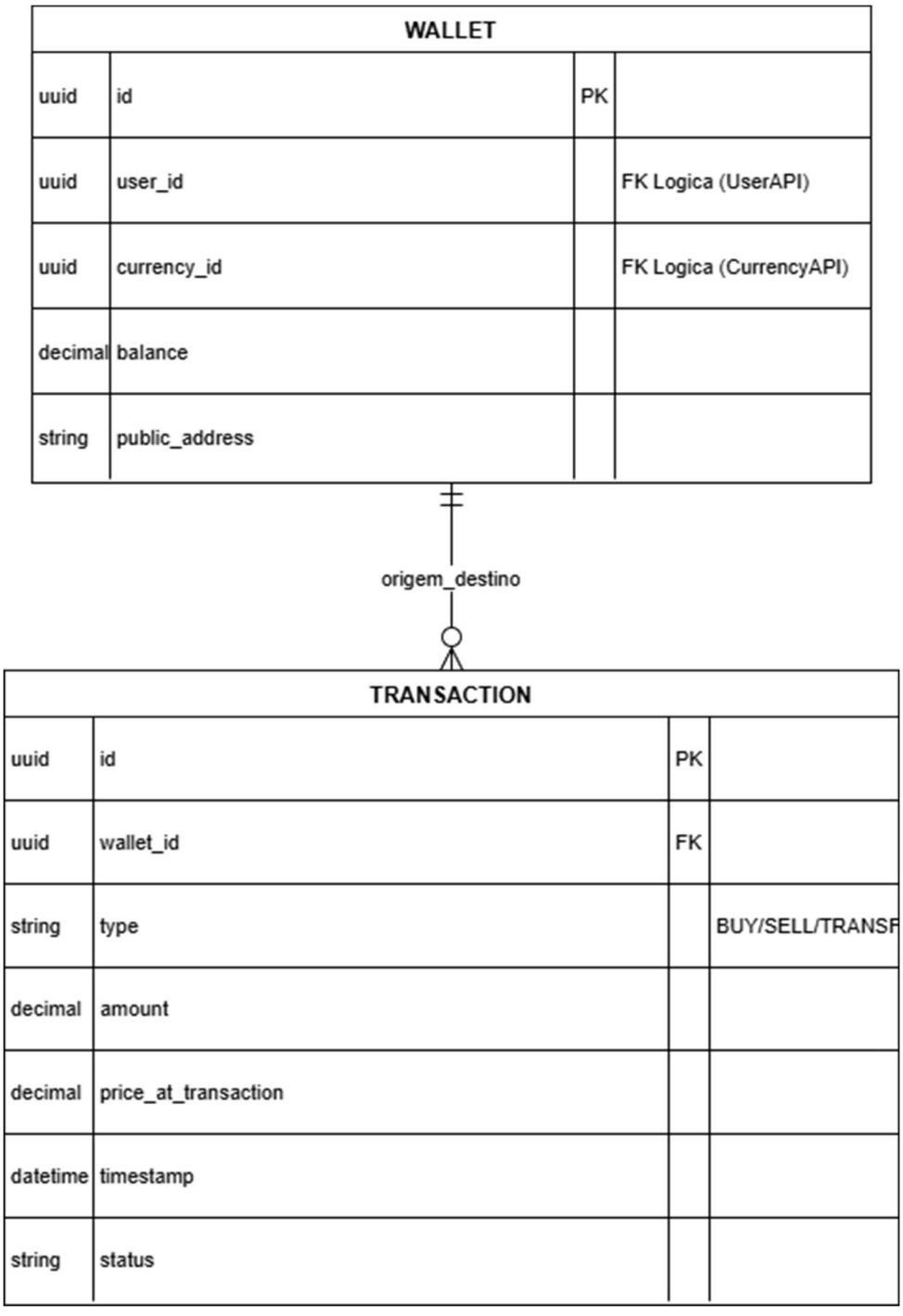
4. Diagrama Entidade-Relacionamento

O DER representa a estrutura do banco de dados relacional, definindo como as informações são armazenadas e conectadas:

- **USER:** Tabela central de usuários, contendo dados de acesso e status de verificação (KYC).

USER		
uuid	id	PK
string	username	
string	email	
string	password_hash	
string	kyc_status	
datetime	created_at	

- **WALLET:** Relaciona um Usuário a uma Moeda (CURRENCY), armazenando o saldo disponível. Um usuário pode ter múltiplas carteiras (ex: uma de BTC, uma de USD).



- **TRANSACTION:** Registra o histórico financeiro. Possui chaves estrangeiras para a Carteira de origem e define o tipo de operação (BUY/SELL/TRANSFER), garantindo auditabilidade.

CURRENCY			
uuid	id	PK	
string	symbol		ex: BTC
string	name		
decimal	current_price		
datetime	last_updated		

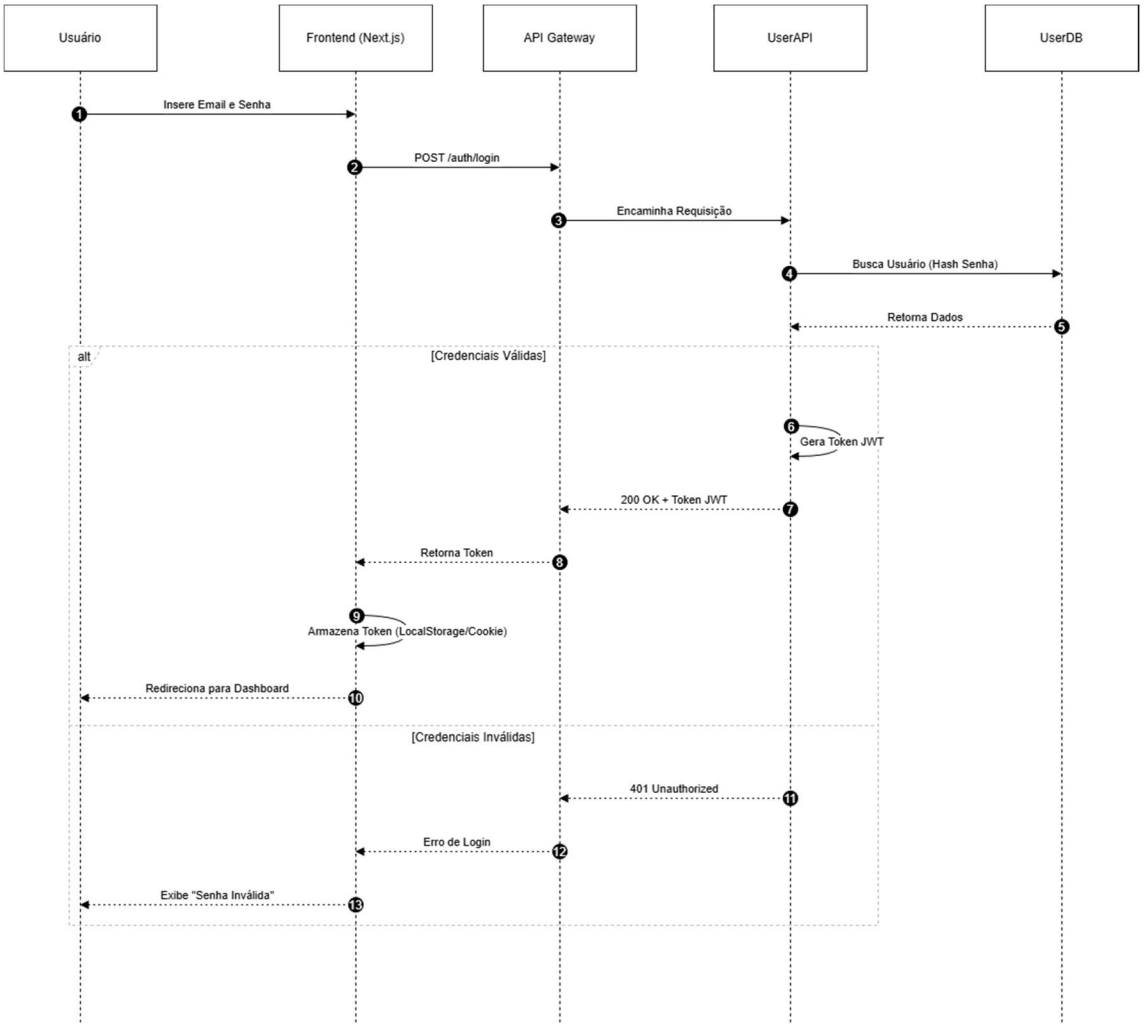
- **CHAT_SESSION:** Armazena o histórico de interações do usuário com o bot para fins de auditoria e melhoria do modelo.

CHAT_SESSION			
uuid	id	PK	
uuid	user_id		FK Logica (UserAPI)
string	message_input		
string	bot_response		
datetime	timestamp		

5. Diagrama de Sequência

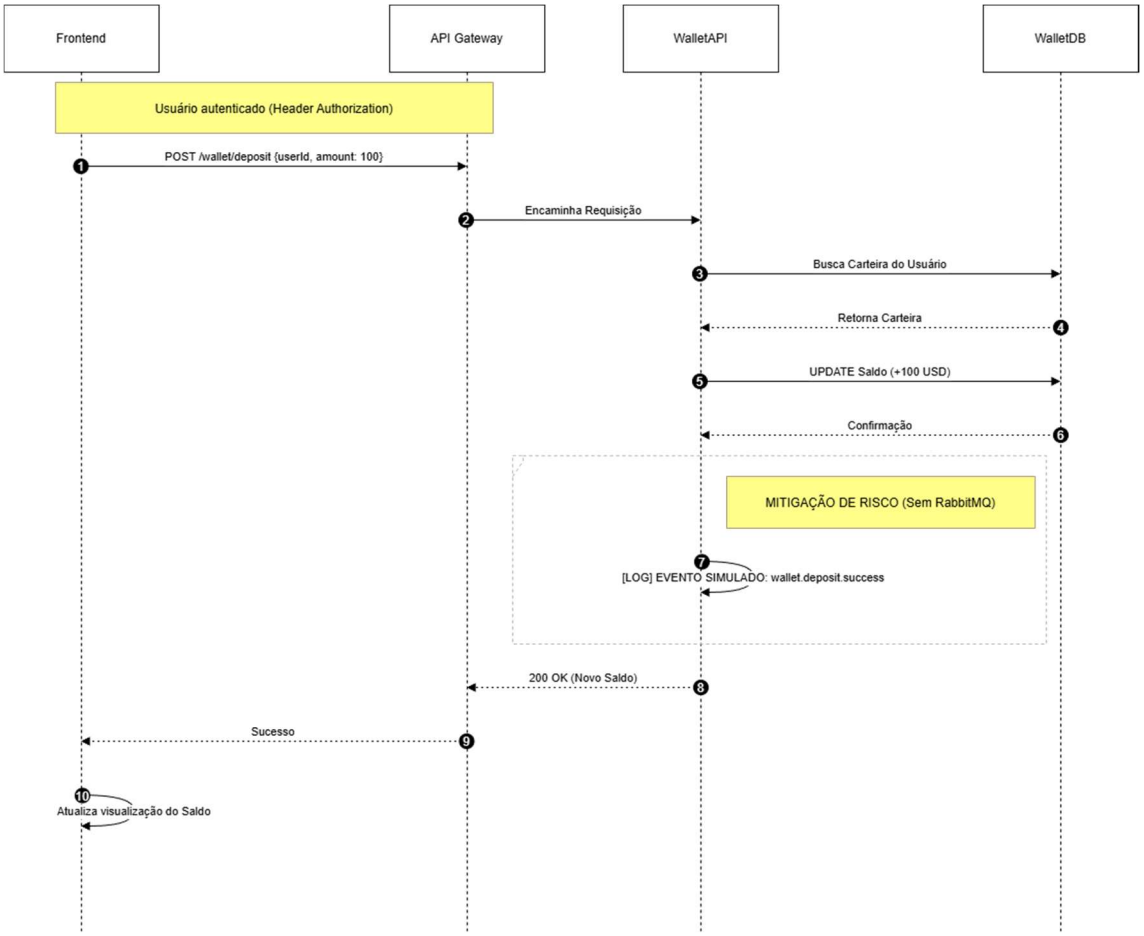
5.1 Fluxo de Login (Autenticação)

Este é o fluxo padrão de segurança. Mostra o Frontend pedindo o token e o Gateway validando.



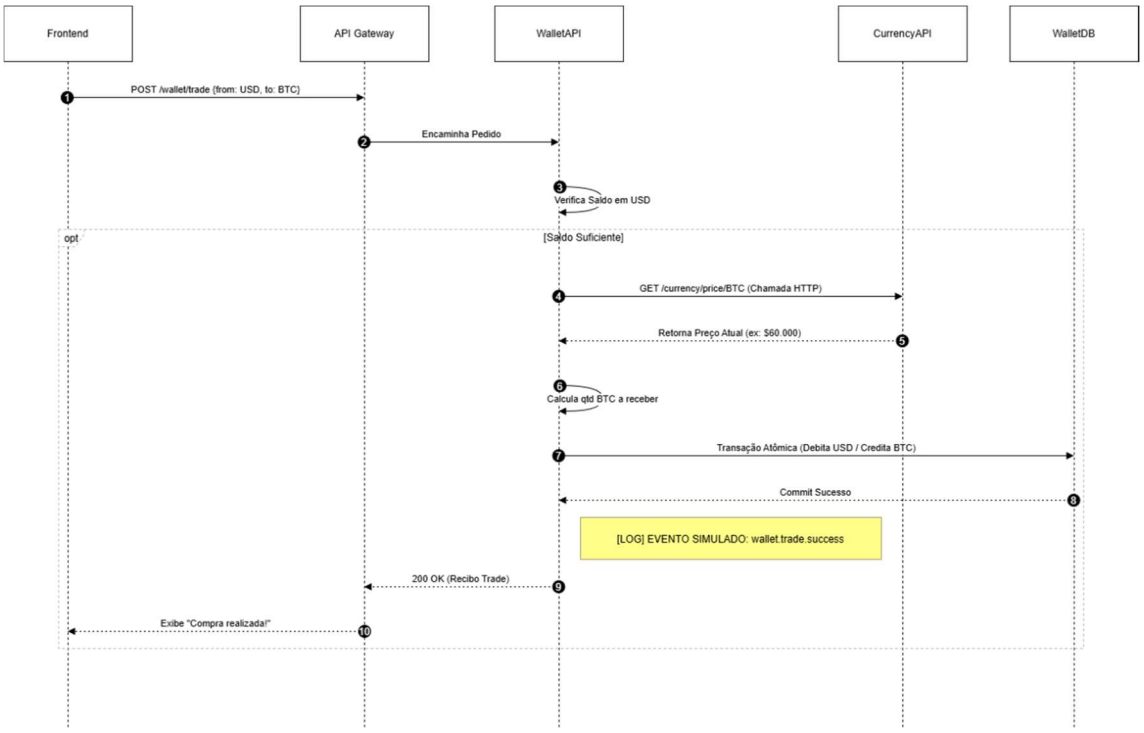
Demonstra a autenticação padrão via **JWT**. O Frontend envia as credenciais, a UserAPI valida o hash da senha no banco e retorna um token assinado, que será usado nas requisições subsequentes.

5.2 Fluxo de Depósito



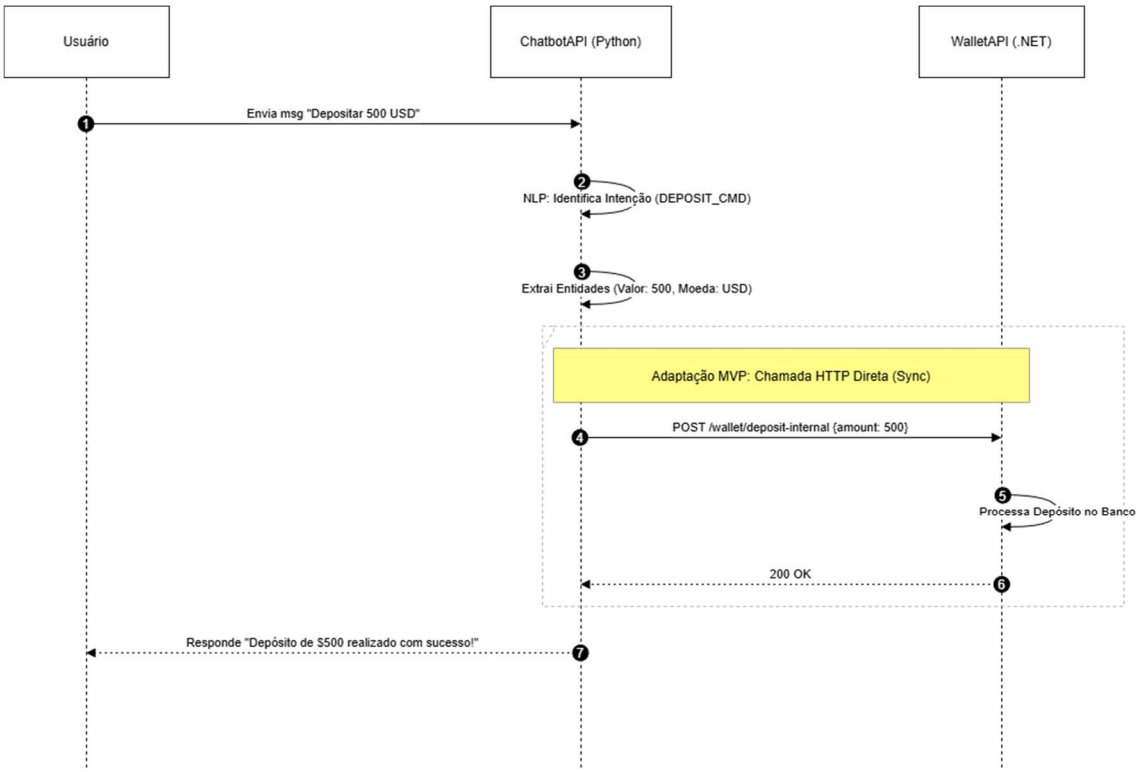
Ilustra a entrada de fundos na conta. A operação é processada atomicamente na WalletAPI. Devido à estratégia de mitigação de riscos, a notificação assíncrona foi substituída por um registro de **Log de Eventos** estruturado, garantindo a rastreabilidade da operação sem a complexidade de infraestrutura de mensageria no MVP.

5.3 Fluxo de Trade



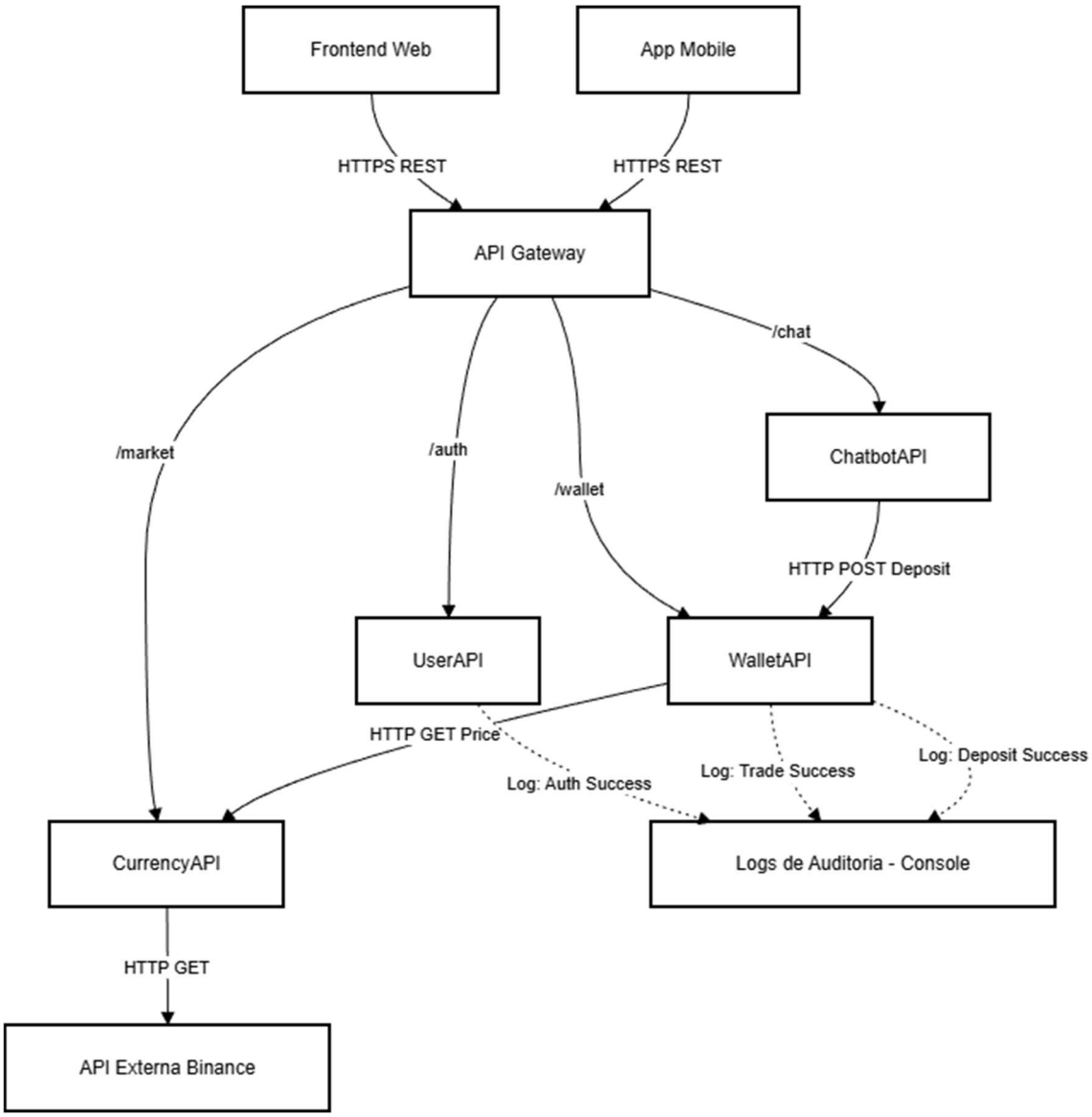
Exibe a orquestração entre microserviços. A WalletAPI atua como coordenadora, consultando a cotação atual na **CurrencyAPI** via HTTP (Síncrono) antes de efetivar a troca de saldo no banco de dados.

5.4 Fluxo do ChatBot



Representa o processamento de Linguagem Natural (NLP). O serviço Python interpreta a intenção do usuário e, para garantir a atualização imediata do saldo durante a demonstração, realiza uma chamada **HTTP direta** para a WalletAPI, confirmando a transação em tempo real.

6. Fluxo de Comunicação entre Serviços



6.1 Descrição do Fluxo:

O diagrama acima consolida as interações entre os componentes do sistema, destacando as adaptações arquiteturais realizadas para o MVP:

1. **Centralização via Gateway:** Todo o tráfego externo é obrigatoriamente roteado pelo API Gateway, que distribui as chamadas para os microserviços de destino (User, Wallet, Currency, Chatbot) utilizando o protocolo HTTP/REST.
2. **Comunicação Inter-Serviços (Síncrona):**

- A **WalletAPI** consome diretamente a **CurrencyAPI** para obter cotações em tempo real durante a execução de trades.
- A **ChatbotAPI** realiza chamadas diretas para a **WalletAPI** para efetivar comandos de depósito. Esta decisão substitui o fluxo assíncrono original para garantir a consistência imediata dos dados durante a demonstração.

3. **Simulação de Eventos (Logs):**
   - Para manter a rastreabilidade prevista na arquitetura original sem a complexidade de infraestrutura do RabbitMQ, os eventos de negócio (como wallet.trade.success e user.auth.success) são direcionados para **Logs Estruturados** no console da aplicação. Isso permite a auditoria dos fluxos de negócio conforme exigido nos requisitos não funcionais.

7. Descrição Textual dos Componentes e Tecnologias

7.1. Visão Geral da Solução

O sistema foi projetado como uma plataforma distribuída para simulação de operações de criptomoedas. A arquitetura adota o estilo de Microserviços, onde cada domínio de negócio (identidade, finanças, dados de mercado e atendimento) opera de forma isolada, garantindo baixo acoplamento.

Para viabilizar a entrega do MVP dentro do cronograma, o projeto sofreu adaptações estratégicas em relação à arquitetura de referência: a comunicação assíncrona foi substituída por interações diretas (REST) e registros de auditoria em log, mantendo a integridade funcional sem a complexidade de infraestrutura de um broker de mensagens.

7.2. Detalhamento dos Componentes de Backend

O ecossistema de servidor é composto por cinco serviços distintos, orquestrados para funcionar em conjunto:

- **GatewayAPI (Orquestrador de Borda):** Funciona como um *Reverse Proxy* inteligente. Sua responsabilidade é interceptar todo o tráfego externo, ocultando a topologia da rede interna. Ele gerencia o roteamento das requisições para os serviços corretos e aplica a primeira camada de segurança, rejeitando requisições sem tokens válidos.
    - *Stack:* .NET 8 com YARP (Yet Another Reverse Proxy) ou Ocelot.
- **UserAPI (Gestão de Identidade):** Serviço autoritativo para dados de usuários. Ele processa o registro de novas contas e valida credenciais. Ao confirmar um login bem-sucedido, este serviço emite um token assinado (JWT) que servirá como "passaporte" para o usuário acessar os demais recursos do sistema.
    - *Stack:* .NET 8, SQLite, BCrypt.Net.
- **WalletAPI (Motor Financeiro):** O núcleo transacional da plataforma. Este microserviço detém a "fonte da verdade" sobre os saldos dos usuários. Ele executa operações críticas, como a atualização de valores após depósitos e a lógica de conversão de ativos (trade), garantindo que as operações sejam atômicas.
    - *Stack:* .NET 8 (Clean Architecture), SQLite.
- **CurrencyAPI (Provedor de Dados):** Atua como um catálogo de ativos. Sua função é monitorar e disponibilizar o preço atual das criptomoedas (ex.: Bitcoin, Ethereum) para que outros serviços, como a WalletAPI, possam realizar cálculos de conversão precisos.
    - *Stack:* .NET 8, SQLite (Cache de Preços).
- **ChatbotAPI (Assistente Virtual):** Módulo de inteligência desenvolvido em Python para processar linguagem natural. Ele analisa frases enviadas pelos usuários (ex.: "Quero depositar 100 reais") e traduz essas intenções em comandos técnicos que são enviados diretamente para a WalletAPI.

- *Stack:* Python 3 (Flask), Requests.

7.3. Camadas de Interface (Frontend)

A interação com o usuário final é provida por duas aplicações clientes que consomem as APIs através do Gateway:

- **Interface Web:** Desenvolvida em **Next.js**, oferece uma experiência completa de dashboard, permitindo visualização de gráficos e gestão detalhada da conta.
- **Aplicação Mobile:** Construída com **React Native (Expo)**, foca na agilidade, permitindo consultas rápidas de saldo e interações via chat em dispositivos móveis.

7.4. Decisões de Arquitetura e Design

7.4.1. Clean Architecture (Camadas)

Nos serviços .NET, adotou-se o padrão de **Arquitetura Limpa**. Isso organiza o código em círculos concêntricos de dependência, onde o Domínio (regras de negócio) fica no centro, isolado de detalhes externos como Banco de Dados ou Frameworks Web. Isso facilita testes e futuras manutenções.

7.4.2. Adaptação da Comunicação (Mitigação de Riscos)

Originalmente planejado para usar RabbitMQ, o sistema adotou uma estratégia de mitigação para o MVP:

- **Sincronismo:** As integrações críticas (como Chatbot acionando Wallet) ocorrem via HTTP. Isso simplifica o fluxo e garante feedback imediato ao usuário na demonstração.
- **Observabilidade via Logs:** A natureza orientada a eventos foi preservada logicamente através de logs estruturados. Eventos de negócio (ex.: TRADE_EXECUTED) são impressos no console do servidor, permitindo auditoria visual do fluxo.

7.4.3. Segurança (Tokenização)

A segurança é baseada em **JWT (JSON Web Tokens)**. O sistema é *stateless* (sem sessão no servidor); a cada requisição, o Frontend envia o token no cabeçalho HTTP, e o Gateway valida sua assinatura criptográfica antes de permitir o acesso.