Documentação Técnica da API MiuBank

Bem-vindo ao MiuBank: Sua Jornada Financeira Gamificada!

Imagine um mundo onde suas finanças não são apenas números em uma planilha, mas uma jornada emocionante, cheia de conquistas e com um companheiro peludo ao seu lado. No MiuBank, acreditamos que gerenciar seu dinheiro pode ser divertido, intuitivo e, acima de tudo, **empoderador**.

Nossa API foi cuidadosamente projetada para ser o coração de uma plataforma de investimentos digital que simula o dinamismo do mercado financeiro real. Para você, nosso usuário final – seja um investidor iniciante buscando segurança, ou alguém que deseja otimizar seus rendimentos com um toque de gamificação –, o MiuBank é a ferramenta que transforma a complexidade do mercado em uma experiência acessível e gratificante.

Aqui, você não apenas consulta seu saldo ou realiza transferências; você **constrói seu futuro financeiro**. Cada depósito, cada investimento, cada meta alcançada reflete diretamente no bem-estar do seu Pet virtual, um companheiro que celebra suas vitórias e te motiva a ir além. Sinta a satisfação de ver seu patrimônio crescer e seu Pet mais feliz a cada decisão financeira inteligente.

Com o MiuBank, você tem o control total: desde as movimentações diárias em sua Conta Corrente, passando pela estratégia de seus investimentos em Renda Fixa e Variável na Conta Investimento, até a visualização clara de seus resultados e impostos. Tudo isso, com a transparência e a segurança que você merece.

Prepare-se para embarcar nesta aventura financeira. O MiuBank não é apenas um banco; é seu parceiro na construção de um futuro financeiro mais próspero e divertido.

1. Visão Geral Técnica

A API do MiuBank é um serviço backend RESTful desenvolvido em Node.js com JavaScript, projetado para simular as operações de um mini banco de investimentos. A arquitetura visa a modularidade, escalabilidade e manutenibilidade, seguindo princípios de boas práticas de desenvolvimento de software.

1.1. Arquitetura e Padrões de Design

O projeto adota uma arquitetura que se assemelha ao padrão **MVC (Model-View-Controller)**, embora adaptada para uma API REST, onde a "View" seria

o frontend que consome a API.

- **Controllers (src/controllers):** Responsáveis por lidar com as requisições HTTP de entrada, validar os dados (com Zod), chamar a lógica de negócio nos serviços apropriados e formatar a resposta para o cliente. Eles atuam como a camada de interface da API.
- **Services (src/services):** Contêm a lógica de negócio principal da aplicação. São responsáveis por orquestrar as operações, interagir com o banco de dados (via Prisma) e aplicar as regras de negócio. A separação dos serviços dos controladores garante que a lógica de negócio seja reutilizável e testável de forma independente.
- **Routes (src/routes):** Definem os endpoints da API e mapeiam as requisições para os controladores correspondentes. Incluem também a documentação Swagger JSDoc para cada endpoint.
- **Middlewares (src/middlewares):** Funções que interceptam as requisições antes que elas cheguem aos controladores. Utilizados para autenticação (authMiddleware) e tratamento global de erros (errorMiddleware).
- **Config (src/config):** Contém configurações da aplicação, como variáveis de ambiente, configuração do Prisma Client e do banco de dados.

Além disso, foram aplicados conceitos de **SOLID Principles** e **Clean Code**:

- **Single Responsibility Principle (SRP):** Cada módulo (controller, service, middleware) tem uma única responsabilidade bem definida. Por exemplo, um accountService lida apenas com a lógica de contas.
- **Open/Closed Principle (OCP):** O código é projetado para ser aberto para extensão, mas fechado para modificação. Novas funcionalidades podem ser adicionadas sem alterar o código existente em módulos centrais.
- **Dependency Inversion Principle (DIP):** Módulos de alto nível (controllers) não dependem de módulos de baixo nível (Prisma Client diretamente), mas sim de abstrações (services). O prismaClient é injetado nos serviços, e os serviços são chamados pelos controllers.

1.2. Tecnologias, Linguagens e Bibliotecas

- **Linguagem:** JavaScript (ES6+)
- **Runtime:** Node.js
- **Framework Web:** Express.js
- **ORM (Object-Relational Mapper):** Prisma ORM
  - Facilita a interação com o banco de dados, gerando um cliente TypeScript/JavaScript seguro para as operações CRUD.
  - Utiliza Prisma.Decimal para garantir precisão em cálculos financeiros,

evitando erros de ponto flutuante.
- **Banco de Dados:** PostgreSQL (configurado via DATABASE_URL no .env)
- **Autenticação:**
  - **bcryptjs:** Para hash e comparação segura de senhas.
  - **jsonwebtoken (JWT):** Para geração e verificação de tokens de autenticação.
- **Validação de Dados:**
  - **Zod:** Biblioteca para construção de esquemas de validação robustos e seguros para dados de entrada.
- **Agendamento de Tarefas:**
  - **node-cron:** Utilizado para agendar a atualização periódica dos preços dos ativos no mercado simulado.
- **Utilidades:**
  - **dotenv:** Para carregar variáveis de ambiente de um arquivo .env.
  - **cors:** Middleware para habilitar Cross-Origin Resource Sharing.
- **Documentação da API:**
  - **swagger-jsdoc:** Gera a especificação OpenAPI a partir de comentários JSDoc no código.
  - **swagger-ui-express:** Serve a interface gráfica do Swagger UI para explorar a API.

1.3. Estrutura do Projeto

```
├── .gitignore
├── README.md
├── assets-mock.json
├── assets
│   ├── assets-mock.json
│   └── users-mock.json
├── package-lock.json
├── package.json
├── prisma
│   ├── migrations
│   │   ├── ... (histórico de migrações)
│   │   └── migration_lock.toml
│   ├── schema.prisma
│   └── seed.js
├── regradenegocio.md
├── src
│   ├── config            # Configurações globais (DB, env, Prisma Client)
│   │   ├── database.js
│   │   ├── env.js
│   │   └── prismaClient.js
│   ├── controllers       # Lógica de requisição/resposta, validação de entrada
```

```
│   │   ├── accountController.js
│   │   ├── authController.js
│   │   ├── financialGoalController.js
│   │   ├── marketController.js
│   │   ├── reportController.js
│   │   └── userController.js
│   ├── index.js            # Ponto de entrada da aplicação, configuração do Express
│   ├── jobs                # Tarefas agendadas (ex: atualização de mercado)
│   │   └── marketUpdateJob.js
│   ├── middlewares         # Middlewares globais (autenticação, tratamento de erros)
│   │   ├── authMiddleware.js
│   │   └── errorMiddleware.js
│   ├── routes              # Definição de rotas da API
│   │   ├── account.routes.js
│   │   ├── authRoutes.js
│   │   ├── financialGoal.routes.js
│   │   ├── market.routes.js
│   │   ├── report.routes.js
│   │   └── user.routes.js
│   ├── services            # Lógica de negócio principal, interação com o DB
│   │   ├── accountService.js
│   │   ├── financialGoalService.js
│   │   ├── marketService.js
│   │   ├── reportService.js
│   │   └── userService.js
├── swagger
│   └── swagger.json        # (Gerado pelo swagger-jsdoc)
└── users-mock.json
```

1.4. Modelagem de Dados (Prisma Schema)

O prisma/schema.prisma define os modelos de dados que representam as entidades do domínio do MiuBank:

- **User**: Informações do usuário (nome, email, senha, CPF, data de nascimento), **incluindo agora um campo points para gamificação**.
- **Account**: Contas do usuário, separadas por CORRENTE e INVESTIMENTO, com seus saldos.
- **Asset**: Ativos disponíveis no mercado (ações, CDB, Tesouro Direto), com preço atual, tipo, e detalhes específicos.
- **Investment**: Registra os investimentos que um usuário fez em um Asset, com quantidade, preço de compra, data, e status de venda.
- **Movement**: Histórico de todas as movimentações financeiras (depósitos, saques, transferências, compra/venda de ativos).
- **Pet**: Entidade para a funcionalidade de gamificação, representando o pet do

usuário e seu humor.
- **FinancialTip**: Pílulas de riqueza ou dicas financeiras.
- **FinancialGoal**: Metas financeiras que os usuários podem criar e acompanhar.

1.5. Fluxo de Autenticação

A API utiliza autenticação baseada em JWT.

1. **Registro (/auth/register):** Um novo usuário é criado, a senha é hashada com bcryptjs, e um token JWT é gerado e retornado.
2. **Login (/auth/login):** O usuário fornece email e senha. A senha é comparada com o hash armazenado. Se válidas, um token JWT é gerado e retornado.
3. **Acesso Protegido:** Para acessar rotas protegidas, o cliente deve incluir o token JWT no cabeçalho Authorization como Bearer <token>. O authMiddleware verifica a validade do token e anexa as informações do usuário à requisição (req.user).

1.6. Operações Financeiras Chave

- **Depósito (/accounts/deposit):** Adiciona valor à Conta Corrente do usuário.
- **Saque (/accounts/withdraw):** Retira valor da Conta Corrente, com validação de saldo.
- **Transferência Interna (/accounts/transfer/internal):** Move fundos entre a Conta Corrente e a Conta Investimento do mesmo usuário, com validações de saldo e pendências de investimento.
- **Transferência Externa (/accounts/transfer/external):** Envia fundos da Conta Corrente para a Conta Corrente de outro usuário, aplicando uma taxa de 0.5% sobre o valor transferido.
- **Compra de Ativos (/market/buy):** Permite ao usuário comprar ações, CDBs ou Tesouro Direto usando o saldo da Conta Investimento. Aplica uma taxa de corretagem de 1% para ações.
- **Venda de Ativos (/market/sell):** Permite ao usuário vender seus investimentos. Calcula o lucro/prejuízo e retém automaticamente o Imposto de Renda (15% para ações, 22% para renda fixa) sobre o lucro, creditando o valor líquido na Conta Investimento.

Todas essas operações são realizadas dentro de **transações de banco de dados** para garantir a integridade e consistência dos dados, mesmo em caso de falhas.

1.7. Simulador de Mercado

Um job agendado (src/jobs/marketUpdateJob.js) é executado a cada 5 minutos (*/5 * * * * na cron expression) para simular a flutuação dos preços das ações e a rentabilidade dos ativos de renda fixa. Para ações, ele aplica variações percentuais

aleatórias (0.1% a 5%) em diferentes distribuições de probabilidade, garantindo que os preços nunca se tornem negativos. Para renda fixa (CDB e Tesouro Direto), a rentabilidade é simulada aplicando a taxa diária proporcional, com a possibilidade de simular um fator de inflação para os pós-fixados. Isso proporciona um ambiente de mercado dinâmico para as operações de investimento.

1.8. Gamificação

A API integra elementos de gamificação para enriquecer a experiência do usuário:

- **Pet:** O humor do Pet do usuário é atualizado com base em suas ações de "economia" (depósitos e investimentos), incentivando bons hábitos financeiros.
- **Metas Financeiras:** Os usuários podem criar e acompanhar metas financeiras personalizadas. Ao criar uma meta ou progredir nela, o sistema atribui pontos ao usuário, incentivando o planejamento e a disciplina.

1.9. Documentação da API (Swagger)

A API é auto-documentada utilizando Swagger JSDoc. Uma interface interativa está disponível em:

http://localhost:<PORTA_DA_API>/api-docs

Esta interface permite explorar todos os endpoints disponíveis, seus parâmetros de requisição, modelos de resposta e esquemas de autenticação, facilitando a integração com o frontend.

2. Versão de Entrega e Melhorias Futuras

Esta entrega representa uma base sólida e funcional para o desafio do Orange Hackathon, demonstrando a implementação das funcionalidades obrigatórias e de alguns diferenciais importantes. No entanto, como em qualquer projeto de software, há sempre espaço para aprimoramento e expansão.

2.1. Coerência com as Regras de Negócio Implementadas

A implementação atual demonstra uma aderência notável às regras de negócio fornecidas, especialmente no que tange às operações financeiras e à estrutura de contas.

- **Tipos de Contas e Separação de Saldos:** A distinção entre Conta Corrente (para movimentações diárias) e Conta Investimento (exclusiva para ativos) está bem implementada. Depósitos e saques são restritos à Conta Corrente, e compra/venda de ativos à Conta Investimento. A independência dos saldos é

garantida.

- **Validação e Custos de Transferência:** As validações de saldo suficiente e valor positivo para transferências internas e externas estão presentes. A regra de taxa de 0.5% para transferências externas entre usuários é aplicada corretamente, sendo debitada da conta de origem.
- **Registro de Movimentações:** Todas as operações financeiras (depósito, saque, transferências, compra e venda de ativos) são devidamente registradas no histórico de movimentações, incluindo data/hora, valor, contas de origem/destino e tipo.
- **Operações de Compra e Venda de Ativos:** A lógica para compra de ativos verifica o saldo na Conta Investimento e aplica a taxa de corretagem de 1% para ações. Na venda, o sistema calcula o lucro/prejuízo e retém automaticamente o Imposto de Renda (15% para Renda Variável/Ações e 22% para Renda Fixa/CDB e Tesouro Direto) sobre o rendimento, creditando o valor líquido.
- **Simulador de Mercado (Ações e Renda Fixa):** A simulação da flutuação de preços para ações fictícias e a rentabilidade para CDBs e Tesouro Direto é bem implementada através de um job agendado, seguindo as distribuições de probabilidade especificadas para variação de preço e garantindo que os valores não se tornem negativos.
- **Relatórios (Extrato, Resumo de Investimentos e Imposto de Renda Consolidado):** O sistema permite a geração de extratos detalhados para Conta Corrente e Conta Investimento, filtráveis por período. Um resumo dos investimentos ativos, incluindo lucro/prejuízo e valor atual, também é fornecido. Adicionalmente, foi implementado um relatório consolidado de Imposto de Renda, que agrega lucros e impostos pagos sobre investimentos vendidos, com opções de filtragem por ano ou período específico.
- **Gamificação (Pet e Metas Financeiras):** A integração do humor do Pet com ações de economia e a funcionalidade de criação e acompanhamento de metas financeiras demonstram um bom início na gamificação, **com o campo points agora devidamente implementado no modelo User para rastreamento de pontos.**

2.2. Pontos Fortes da Entrega

- **Arquitetura Limpa e Modular:** A separação de responsabilidades e a organização do código em camadas (controllers, services, routes) facilitam a compreensão e a manutenção.
- **Segurança Robusta:** Uso de JWT para autenticação e bcryptjs para senhas, além de validação de entrada com Zod, garantem um bom nível de segurança.
- **Transações Atômicas:** A utilização de transações de banco de dados para

operações financeiras críticas é um diferencial que garante a integridade dos dados.
- **Simulador de Mercado Dinâmico:** A implementação do job de atualização de preços de ações e a rentabilidade de renda fixa adiciona um elemento de realismo e dinamismo ao ambiente financeiro simulado.
- **Cálculos Financeiros Precisos:** O uso de Prisma.Decimal para todas as operações monetárias é fundamental para evitar erros de arredondamento.
- **Documentação Interativa (Swagger):** A API é bem documentada, o que é crucial para a colaboração e a integração com o frontend.
- **Gamificação Inicial:** A inclusão do Pet e das Metas Financeiras demonstra criatividade e um olhar para o engajamento do usuário.

2.3. Regras de Negócio Não Contempladas / Melhorias a Serem Implementadas

Para a próxima fase de desenvolvimento e para tornar a API ainda mais completa e robusta, as seguintes melhorias são recomendadas:

1. **Implementação de Testes Unitários com Jest:**
    - **Descrição:** Criar um conjunto abrangente de testes unitários utilizando a biblioteca Jest para os services e controllers. Isso garantirá a validação automática da lógica de negócio, a detecção precoce de regressões e a segurança para futuras refatorações.
    - **Prioridade: Alta (Qualidade e Manutenibilidade do Código).**
2. **Sistema de Notificações:**
    - **Descrição:** Implementar um módulo de notificações para alertar os usuários sobre eventos importantes, como vencimento de investimentos de renda fixa ou alterações significativas nos preços das ações em sua carteira, conforme as funcionalidades extras sugeridas.
    - **Prioridade:** Baixa/Média (Funcionalidade Opcional).
3. **Simulação de Cenários e Projeções de Investimento:**
    - **Descrição:** Desenvolver funcionalidades no backend que permitam aos usuários simular investimentos futuros, escolhendo ativos, valores e prazos, e visualizar projeções de rentabilidade com base em diferentes cenários de mercado (ex: taxas de juros, inflação ou variação de preços).
    - **Prioridade:** Baixa (Funcionalidade Opcional).
4. **Expansão da Gamificação (Badges, Ranking):**
    - **Descrição:** Adicionar mais elementos de gamificação, como a concessão de distintivos (badges) por conquistas específicas e a criação de rankings ou desafios entre usuários.
    - **Prioridade:** Baixa (Funcionalidade Opcional).
5. **Refinamento da Regra de "Pendências" na Transferência de Conta**

Investimento:

- **Descrição:** A regra de "não haver operações pendentes de compra ou venda de ativos" é atualmente simplificada pela verificação de investimentos não vendidos. Para um sistema mais robusto, pode-se considerar a introdução de um modelo de Order para rastrear ordens de compra/venda abertas e verificar pendências de forma mais granular.
- **Prioridade:** Baixa (Refinamento de Regra).