Sumário

1. Visão Geral do Backend

1.1. Objetivo

O objetivo principal do backend do EveryFin é fornecer uma base segura, escalável e de alto desempenho para o sistema financeiro, garantindo qualidade de código, resiliência e manutenibilidade. Ele será o coração da aplicação, responsável por toda a lógica de negócio, persistência de dados e segurança das operações financeiras.

1.2. Escopo

O escopo inicial do backend abrange as seguintes áreas-chave:

- Desenvolvimento de APIs RESTful robustas e bem definidas para autenticação de usuários, gerenciamento de transações financeiras e geração de relatórios.
- Criação de uma estrutura modular e preparada para futura expansão, visando a capacidade de evoluir para uma arquitetura de microsserviços, se necessário.
- Adoção de padrões corporativos rigorosos de segurança, garantindo a proteção dos dados financeiros e das informações dos usuários.
- Implementação de um sistema abrangente de testes automatizados para assegurar a confiabilidade e a integridade do código.

2. Tecnologias e Dependências

A seleção das tecnologias e dependências foi feita visando um equilíbrio entre performance, segurança, produtividade e o alinhamento com as melhores práticas do mercado para aplicações corporativas e financeiras.

2.1. Backend Core (Node.js & TypeScript)

- **Node.js & TypeScript**: A combinação oferece tipagem estática e acesso à modernidade do ESNext, o que resulta em um código mais seguro, legível e manutenível. Node.js é ideal para APIs de alta concorrência devido ao seu modelo I/O não bloqueante.
- **Express**: Um framework web leve e flexível para Node.js, amplamente utilizado para a construção de APIs RESTful, com foco na simplicidade e na capacidade de integração de middlewares.
- **Compression**: Utilização do middleware compression para habilitar a compressão Gzip das respostas HTTP, reduzindo o tamanho dos payloads transferidos e melhorando a performance da comunicação cliente-servidor.
- **Dotenv**: Essencial para o gerenciamento de variáveis de ambiente, permitindo que as configurações sensíveis (como credenciais de banco de dados, chaves de API) sejam carregadas de um arquivo .env no ambiente de desenvolvimento, mantendo-as fora do controle de versão.

2.2. Acesso a Dados (Prisma & PostgreSQL)

- **Prisma ORM**: Um mapeador objeto-relacional (ORM) de última geração que simplifica a interação com o banco de dados. Ele oferece um gerador de código para o cliente do Prisma, tipagem forte para consultas e um sistema robusto de migrations automatizadas, garantindo que o esquema do banco de dados e o código estejam sempre sincronizados.
- **PostgreSQL**: Um sistema de gerenciamento de banco de dados relacional robusto, de código aberto, conhecido por sua confiabilidade, integridade de dados e capacidade de lidar com cargas de trabalho complexas. É uma escolha padrão para aplicações financeiras. A versão 14+ será utilizada para aproveitar os recursos e otimizações mais recentes.

2.3. Segurança (JWT, Bcrypt, Helmet, CORS, Rate Limit)

- **JWT (JSON Web Tokens)**: Padrão amplamente adotado para autenticação e autorização, permitindo a criação de tokens de acesso de curta duração (15 minutos) e refresh tokens para revalidação, aumentando a segurança e a usabilidade. Utiliza a biblioteca jsonwebtoken.
- **Bcrypt**: Biblioteca de hash de senhas de alto desempenho, crucial para armazenar senhas de forma segura. Será configurado com 12 salt rounds, um fator de custo que oferece forte resistência a ataques de força bruta.
- **Helmet**: Um conjunto de middlewares Express que ajuda a proteger o aplicativo configurando cabeçalhos HTTP relacionados à segurança, como Content Security Policy, X-XSS-Protection, entre outros.

- **CORS (Cross-Origin Resource Sharing)**: Configurado para controlar quais origens (domínios de frontend) podem acessar a API, aplicando políticas corporativas de segurança e prevenindo ataques de origem cruzada.
- **Express-rate-limit**: Middleware para limitar o número de requisições que um IP pode fazer em um determinado período (ex: 100 requisições por 15 minutos). Isso é fundamental para mitigar ataques de força bruta, DoS e abuso de API.

2.4. Utilidades (Zod, Winston, Compression, Dotenv)

- **Zod**: Uma biblioteca de declaração e validação de esquemas que permite criar validadores de forma tipada e composable. Será usado para validar payloads de requisições, parâmetros de rota e queries, garantindo a integridade dos dados de entrada.
- **Winston**: Uma biblioteca de logging madura e versátil para Node.js, permitindo logs estruturados, configuráveis e rotativos, essenciais para ambientes de produção e depuração eficaz.
- **Compression**: Middleware Express para compressão de resposta HTTP (gzip), otimizando a velocidade de transferência de dados.
- **Dotenv**: Utilizado para carregar variáveis de ambiente de um arquivo .env para process.env, facilitando a configuração local do projeto.

2.5. DevDependencies

- typescript: Para a transpilação do código de TypeScript para JavaScript.
- ts-node-dev: Permite o hot-reloading em tempo de desenvolvimento, agilizando o ciclo de feedback.
- eslint e prettier: Ferramentas para linting (análise estática de código para identificar problemas) e formatação automática, garantindo consistência e qualidade do código.
- jest e supertest: Framework de testes e biblioteca para testar APIs HTTP, respectivamente, essenciais para a construção de testes automatizados robustos.

3. Modelos de Dados (Prisma)

A modelagem de dados no Prisma é a espinha dorsal da persistência de informações, definindo a estrutura das entidades e seus relacionamentos no banco de dados PostgreSQL.

3.1. User Model

Representa os usuários do sistema.

```
model User {
  id        Int      @id @default(autoincrement())
  name      String
  email     String   @unique
  cpf       String   @unique
  password  String   // Armazenar o hash da senha
  isAdmin   Boolean  @default(false)
  createdAt DateTime @default(now())
```

```
  updatedAt DateTime @updatedAt
  entries   Entry[]  // Relação com as receitas do usuário
  exits     Exit[]   // Relação com as despesas do usuário
}
```

- **id**: Identificador único do usuário.
- **name**: Nome completo do usuário.
- **email**: Endereço de e-mail único do usuário, também usado para login.
- **cpf**: Cadastro de Pessoa Física único do usuário, importante para validações financeiras.
- **password**: Campo para armazenar o hash da senha do usuário.
- **isAdmin**: Booleano para indicar se o usuário possui privilégios administrativos.
- **createdAt**: Data e hora de criação do registro.
- **updatedAt**: Data e hora da última atualização do registro.
- **entries**: Relação um-para-muitos com o modelo Entry (receitas), permitindo acessar todas as receitas de um usuário.
- **exits**: Relação um-para-muitos com o modelo Exit (despesas), permitindo acessar todas as despesas de um usuário.

3.2. Entry Model

Representa as receitas ou entradas financeiras.

```
model Entry {
  id          Int      @id @default(autoincrement())
  description String
  value       Float
  date        DateTime
  category    String
  client      String   // Quem pagou/gerou a receita
  user        User     @relation(fields: [userId], references: [id])
  userId      Int
}
```

- **id**: Identificador único da receita.
- **description**: Descrição detalhada da receita.
- **value**: Valor monetário da receita.
- **date**: Data em que a receita ocorreu.
- **category**: Categoria da receita (ex: "Salário", "Freelance", "Rendimentos").
- **client**: Nome do cliente ou fonte pagadora da receita.
- **user**: Relação com o User que registrou a receita.
- **userId**: Chave estrangeira para o User.

3.3. Exit Model

Representa as despesas ou saídas financeiras.

```
model Exit {
  id          Int      @id @default(autoincrement())
  description String
  value       Float
```

```
  date        DateTime
  category    String
  provider    String   // Quem recebeu o pagamento/fornecedor
  user        User     @relation(fields: [userId], references: [id])
  userId      Int
}
```

- **id**: Identificador único da despesa.
- **description**: Descrição detalhada da despesa.
- **value**: Valor monetário da despesa.
- **date**: Data em que a despesa ocorreu.
- **category**: Categoria da despesa (ex: "Alimentação", "Transporte", "Moradia").
- **provider**: Nome do fornecedor ou para quem o pagamento foi efetuado.
- **user**: Relação com o User que registrou a despesa.
- **userId**: Chave estrangeira para o User.

3.4. Considerações Adicionais para Modelos Financeiros

- **Tipos Numéricos**: Em sistemas financeiros, é crucial usar tipos de dados que evitem problemas de precisão com números de ponto flutuante. Embora Float seja usado no exemplo, para produção, Decimal (ou money no PostgreSQL) seria mais apropriado no esquema Prisma e no banco de dados para garantir precisão monetária.
- **Transações Atômicas**: A lógica de negócio que envolve múltiplas operações de Entry e Exit (ex: transferências entre contas, cálculo de saldos) deve ser encapsulada em transações de banco de dados para garantir atomicidade e consistência. O Prisma oferece APIs para isso.
- **Contas Financeiras (Account):** Embora não esteja nos modelos iniciais, um modelo Account pode ser adicionado para representar contas bancárias ou carteiras de um usuário, permitindo o controle de múltiplos saldos e transferências entre contas.

```
// Exemplo de um possível modelo Account para futura expansão
// model Account {
//   id        Int      @id @default(autoincrement())
//   name      String
//   balance   Decimal  @default(0.00) @db.Decimal(10, 2) // Usar
Decimal para precisão
//   type      String   // Ex: 'Checking', 'Savings', 'Credit
Card'
//   user      User     @relation(fields: [userId], references:
[id])
//   userId    Int
//   transactions Transaction[] // Relação se Entry/Exit virarem
um único modelo Transaction
// }
```

4. Organização de Pastas

A estrutura de pastas promove a modularidade, separação de responsabilidades e facilita a localização de arquivos, essenciais para a manutenibilidade e escalabilidade do projeto.

```
/backend
├── src/
|   ├── controllers/      # Lógica de manipulação de requisições HTTP e invocação de serviços. Cada arquivo representa um recurso (ex: `authController.ts`, `userController.ts`).
|   ├── services/         # Contém a lógica de negócio principal da aplicação. Funções que orquestram operações, interagem com o banco de dados via Prisma, e aplicam regras de negócio.
|   ├── middlewares/      # Funções que processam requisições antes ou depois dos controllers, como autenticação (`authMiddleware.ts`), tratamento de erros (`errorMiddleware.ts`) e rate limiting.
|   ├── models/           # Não diretamente usado para definição dos modelos Prisma (que ficam em `prisma.schema`), mas pode conter interfaces TypeScript ou DTOs (Data Transfer Objects) que representam a estrutura dos dados.
|   ├── utils/            # Funções utilitárias e helpers genéricos que podem ser reutilizados em diferentes partes da aplicação (ex: formatadores, validadores customizados, geradores de hash).
|   ├── config/           # Contém arquivos de configuração global da aplicação, como configurações de ambiente, chaves de API, e setup de bibliotecas (ex: Winston, Prisma).
|   └── index.ts          # Ponto de entrada da aplicação, onde o servidor Express é inicializado, rotas e middlewares são configurados.
├── tests/                # Contém todos os testes automatizados para o backend, organizados por tipo (unitários, integração) e/ou por funcionalidade.
├── .env.example          # Modelo de arquivo para variáveis de ambiente, usado para configuração local.
├── prisma/               # Diretório para o Prisma ORM, contendo o esquema do banco de dados e as migrações geradas.
|   └── schema.prisma     # Define o esquema do banco de dados e os modelos.
|   └── migrations/       # Contém os scripts de migração gerados pelo Prisma.
|   └── seed.ts           # Script para popular o banco de dados com dados iniciais ou de teste.
├── package.json          # Arquivo de configuração do Node.js, listando dependências e scripts.
└── tsconfig.json         # Configurações do compilador TypeScript.
```

5. Segurança e Autenticação

A segurança é o pilar de qualquer sistema financeiro. O backend do EveryFin implementará

múltiplas camadas de proteção.

5.1. JSON Web Tokens (JWT)

- **Access Token**: Utilizado para autorizar requisições a recursos protegidos. Terá um tempo de expiração curto (15 minutos) para minimizar o risco de uso indevido em caso de comprometimento.
- **Refresh Token**: Um token de longa duração (7 dias de expiração) usado exclusivamente para obter um novo access token quando o atual expirar, evitando que o usuário precise fazer login repetidamente. O refresh token será armazenado de forma segura no cliente (ex: HTTP-only cookie).

5.2. Hash de Senhas

- As senhas dos usuários serão armazenadas com hash usando bcrypt com um fator de custo de 12 salt rounds. Isso garante que, mesmo que o banco de dados seja comprometido, as senhas originais não possam ser facilmente recuperadas.

5.3. Middleware de Autenticação

- Um middleware auth dedicado será implementado para verificar a validade do JWT em cada requisição a rotas protegidas. Ele decodificará o token, validará sua assinatura e data de expiração, e injetará as informações do usuário (ID, roles) na requisição (ex: req.user), permitindo que a lógica de negócio use essas informações para autorização. Se o access token estiver expirado, mas um refresh token válido for fornecido, um novo access token poderá ser gerado e retornado.

5.4. Rate Limiting

- express-rate-limit será configurado para limitar o número de requisições por IP. Por exemplo, 100 requisições por 15 minutos para rotas gerais e limites mais estritos (ex: 5 requisições por 5 minutos) para rotas críticas como login e registro. Isso protege contra ataques de força bruta, negação de serviço (DoS) e varredura de vulnerabilidades.

5.5. Headers de Segurança (Helmet & CORS)

- **Helmet**: Será usado para configurar uma série de cabeçalhos HTTP que aumentam a segurança da aplicação, como X-Content-Type-Options, X-Frame-Options, Strict-Transport-Security, e Content-Security-Policy (CSP) para prevenir ataques como XSS e Clickjacking.
- **CORS**: As políticas de Cross-Origin Resource Sharing serão estritamente configuradas para permitir requisições apenas de origens confiáveis (domínios do frontend e de APIs de terceiros autorizadas), prevenindo requisições maliciosas de outros domínios.

5.6. Gerenciamento de Sessões e Revogação

- **Refresh Token Storage**: O refresh token será armazenado no banco de dados (associado ao usuário) ou em um cache seguro (ex: Redis). Isso permite a revogação de

tokens em caso de logout do usuário ou detecção de atividade suspeita.
- **Logout Seguro**: Ao fazer logout, o access token e o refresh token devem ser invalidados no servidor, adicionando o refresh token a uma lista negra ou excluindo-o do armazenamento.

5.7. Prevenção de Ataques Comuns

- **Injeção SQL**: O uso do Prisma ORM ajuda a mitigar injeções SQL, pois ele utiliza prepared statements por padrão.
- **XSS (Cross-Site Scripting)**: Embora seja mais um problema de frontend, o backend deve garantir que qualquer dado textual recebido e armazenado seja sanitizado antes de ser enviado ao frontend para exibição, se houver risco de conter conteúdo executável (HTML/JS).
- **Injeção de Comando/Shell**: Validar e sanitizar todas as entradas de usuário que possam ser usadas em comandos de sistema.
- **Exposição de Dados Sensíveis**: Evitar o retorno de dados sensíveis (hashes de senha, tokens de refresh) na resposta da API, a menos que estritamente necessário e seguro.

6. Validação e Logs

6.1. Validação de Dados com Zod

- **Schemas Modulares**: Zod será empregado para criar schemas de validação modulares para cada payload de requisição (corpo, parâmetros de query, parâmetros de rota). Isso garante que apenas dados bem formados e esperados cheguem à lógica de negócio.
- **Tipagem Forte**: Os schemas Zod fornecem inferência de tipos TypeScript, garantindo que os dados validados correspondam aos tipos esperados, melhorando a segurança e a manutenibilidade do código.
- **Validação na Camada de Entrada**: A validação Zod ocorrerá nas camadas de controllers ou middlewares, antes que os dados cheguem aos services, para falhar rapidamente em caso de dados inválidos.

6.2. Tratamento Centralizado de Erros

- Um middleware de tratamento de erros será implementado para padronizar as respostas de erro da API. Ele capturará erros lançados pela aplicação (incluindo erros de validação Zod, erros de banco de dados, e erros de lógica de negócio).
- **Respostas Padronizadas**: Todas as respostas de erro seguirão um formato JSON consistente, incluindo um status HTTP apropriado (ex: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error) e uma mensagem clara, muitas vezes com um código de erro para depuração.
- **Ocultar Detalhes em Produção**: Em ambiente de produção, stack traces e detalhes de erros internos não serão expostos ao cliente final, apenas mensagens genéricas de erro, para evitar vazamento de informações sensíveis.

6.3. Logging Estruturado com Winston

- **Logs Rotativos**: O Winston será configurado para gerar logs de erro, aviso (warning) e informação (info) em arquivos rotativos. Isso previne que os arquivos de log cresçam indefinidamente e facilita o gerenciamento.
- **Formatos de Log**: Os logs serão gerados em formato JSON para facilitar a análise por ferramentas de agregação de logs. Cada entrada de log incluirá metadados importantes como timestamp, nível do log, mensagem, ID da requisição (se aplicável), e informações do usuário/contexto.
- **Níveis de Log**: Utilizar diferentes níveis de log (debug, info, warn, error, fatal) para granularidade. Em produção, logs de debug podem ser desabilitados para reduzir o volume.
- **Captura de Exceções**: O Winston será configurado para capturar e logar exceções não tratadas e rejeições de promises no Node.js, garantindo que todos os erros inesperados sejam registrados.

7. Testes Automatizados

A adoção de testes automatizados é fundamental para a qualidade do software, identificação precoce de bugs e segurança de refatorações.

7.1. Estrutura de Testes

- **Jest + Supertest**: Jest será o framework de testes principal para o backend. Supertest será utilizado em conjunto com Jest para simular requisições HTTP e testar endpoints da API de forma eficaz.
- **Testes Unitários**: Foco em testar unidades isoladas de código (funções utilitárias, métodos de serviço, middlewares). Isso garante que cada parte do código funcione conforme o esperado em isolamento.
- **Testes de Integração**: Essenciais para endpoints críticos, verificando a interação entre diferentes componentes (controllers, services, banco de dados). Estes testes simularão cenários reais de uso da API.
- **Mocks e Stubs**: Uso extensivo de mocks e stubs para isolar as unidades de teste de dependências externas (ex: banco de dados, APIs de terceiros), tornando os testes mais rápidos e confiáveis.

7.2. Cobertura de Testes

- Uma meta mínima de 80% de cobertura de código será estabelecida. Embora a cobertura não garanta a ausência de bugs, ela indica a proporção do código que está sendo exercitada pelos testes e ajuda a identificar áreas com baixo escopo de teste.

8. CI/CD e Deploy

O pipeline de Integração Contínua (CI) e Entrega Contínua (CD) automatizará a construção, teste e implantação do backend, garantindo entregas rápidas e confiáveis.

8.1. GitHub Actions Workflow (backend-ci.yml)

- Um workflow (backend-ci.yml) será configurado no GitHub Actions para automatizar as etapas de CI.
- **Passos do Workflow:**
    1. **Instalação de Dependências**: Instalará todas as dependências do projeto.
    2. **Linting**: Executará o ESLint para verificar a qualidade do código e aderência a padrões.
    3. **Testes**: Executará todos os testes unitários e de integração definidos.
    4. **Build**: Compilará o código TypeScript para JavaScript.
    5. **Análise de Segurança**: Integrar ferramentas de análise estática de código (SAST) e varredura de vulnerabilidades de dependências (ex: npm audit, Snyk).
    6. **Publicação de Artefatos**: Se a aplicação for conteinerizada (Docker), a imagem será construída e enviada para um registro de contêineres (ex: Docker Hub, AWS ECR).

8.2. Ambientes de Deployment

- **dev**: Ambiente de desenvolvimento local.
- **staging**: Ambiente de pré-produção, utilizado para homologação, testes de aceitação do usuário (UAT) e demonstrações. O deploy para staging será acionado por merges na branch dev.
- **prod**: Ambiente de produção, onde a aplicação é acessível aos usuários finais. O deploy para produção será acionado por merges na branch main ou por tags semânticas (ex: v1.0.0).

8.3. Estratégia de Deploy

- **Scripts de Release com Tags Semânticas**: O processo de release será baseado em tags semânticas no Git (ex: v1.0.0, v1.0.1, v1.1.0). Isso permite rastrear versões e facilita rollbacks.
- **Conteinerização (Docker)**: O backend será empacotado como uma imagem Docker. Isso garante consistência entre ambientes e facilita a implantação em orquestradores de contêineres (Kubernetes, ECS).
- **Zero-Downtime Deployment**: As plataformas de deployment (Kubernetes, AWS ECS) serão configuradas para realizar deployments com zero-downtime, garantindo que o serviço permaneça disponível durante as atualizações.
- **Rollback Automatizado**: Capacidade de reverter rapidamente para uma versão anterior estável em caso de problemas pós-deploy.

9. Monitoramento e Métricas

Monitorar o backend é essencial para entender seu comportamento, identificar problemas e garantir alta disponibilidade e performance.

9.1. Coleta de Métricas (Prometheus + Grafana)

- **Prometheus**: Será utilizado para coletar métricas de performance do backend. Isso inclui métricas como tempo de resposta de requisições, taxa de erros (códigos 5xx), uso de

CPU e memória, tráfego de rede e latência de consultas ao banco de dados.
- **Grafana**: Dashboards personalizados no Grafana serão criados para visualizar as métricas coletadas pelo Prometheus, fornecendo uma visão em tempo real da saúde e performance do sistema.
- **Instrumentação de Código**: O código do backend será instrumentado para expor métricas personalizadas (ex: número de transações processadas, tempo de processamento de operações críticas) para o Prometheus.

9.2. Alertas

- Configurar regras de alerta no Prometheus (ou diretamente no Grafana) para disparar notificações automáticas para a equipe de operações.
- **Exemplos de Alertas:**
    - Tempo de resposta médio da API excedendo 300ms.
    - Taxa de erros (5xx) acima de um limite (ex: 1% das requisições).
    - Uso de CPU ou memória em excesso.
    - Latência de banco de dados elevada.
    - Falhas de health check.

9.3. Health Checks

- Um endpoint /health será implementado no backend para fornecer informações sobre a saúde da aplicação.
- **Readiness Probe**: Verifica se a aplicação está pronta para aceitar tráfego (ex: conexão com banco de dados estabelecida, dependências carregadas).
- **Liveness Probe**: Verifica se a aplicação está funcionando corretamente e não está em um estado de loop infinito ou deadlock.
- Esses health checks são cruciais para orquestradores como Kubernetes, que os utilizam para gerenciar o ciclo de vida dos contêineres e redirecionar o tráfego.

9.4. Observabilidade Adicional

- **Distributed Tracing**: Para sistemas mais complexos ou microsserviços, a implementação de Distributed Tracing (ex: OpenTelemetry, Jaeger, Zipkin) seria uma próxima etapa para rastrear requisições através de múltiplos serviços e depurar latência.
- **Log Aggregation**: Em ambientes de produção, os logs do Winston serão enviados para um sistema de agregação centralizado (ex: ELK Stack, Splunk, Datadog, AWS CloudWatch Logs) para facilitar a busca, análise e correlação de eventos.

10. Proposta Negocial - Backend

Esta seção detalha a proposta de valor, cronograma e investimento para a entrega do backend do EveryFin.

10.1. Objetivo de Negócio

O objetivo principal da entrega do backend é fornecer uma infraestrutura robusta e confiável,

seguindo padrões corporativos de qualidade, que assegure a confiabilidade e escalabilidade das operações financeiras do EveryFin.

10.2. Diferenciais

- **Segurança de Nível Empresarial**: Implementação de práticas de segurança avançadas, incluindo hash de senhas, JWT, rate limiting, Helmet e CORS, para proteger os dados sensíveis.
- **Código Modular e Testável**: Arquitetura organizada com separação de responsabilidades (controllers, services, middlewares) e alta cobertura de testes automatizados, facilitando a manutenção e a adição de novas funcionalidades.
- **Alta Performance e Monitoramento Ativo**: Otimização de consultas, compressão de dados e um sistema completo de monitoramento com Prometheus e Grafana para garantir a performance e a identificação proativa de problemas.
- **Documentação Técnica para Auditoria**: Geração de documentação detalhada (arquitetura, API) para facilitar a auditoria interna e externa, bem como o onboarding de novas equipes.

10.3. Cronograma

O desenvolvimento do backend será dividido nas seguintes fases com seus respectivos prazos estimados:

- **Setup e Configuração**: 1 semana
    - Configuração do ambiente de desenvolvimento, repositório, dependências, scripts de CI/CD básicos, e setup inicial do Prisma com modelos preliminares.
- **Implementação de Autenticação e Modelos de Usuário**: 2 semanas
    - Desenvolvimento das rotas de cadastro e login, implementação de JWT e Bcrypt, e a lógica de persistência do modelo User.
- **Rotas de Transações (Entries & Exits)**: 2 semanas
    - Implementação das APIs CRUD (Criar, Ler, Atualizar, Deletar) para as entidades Entry e Exit, incluindo validações de dados e regras de negócio.
- **Testes e QA**: 1 semana
    - Escrita e execução de testes de integração e unitários abrangentes para todas as funcionalidades implementadas, correção de bugs e garantia de qualidade.
- **Deploy e Documentação**: 1 semana
    - Configuração final do pipeline de deployment em ambientes de staging e produção, geração da documentação técnica e de API, e entrega final.

10.4. Investimento

- **Valor Único**: O valor total do investimento para o desenvolvimento e entrega do backend completo será de R$ X.XXX,XX.
- **Condições de Pagamento**: O pagamento será dividido em duas parcelas: 50% no início do projeto e 50% na entrega final do backend, após a conclusão de todas as fases e validação.