Agora, com base no conteúdo do PDF que você forneceu e incorporando as sugestões, vou gerar o conteúdo do documento "EveryFin - Setup do Projeto" aprimorado. Lembre-se que o resultado será em texto Markdown.

EveryFin - Setup do Projeto (Documento Técnico e Negocial) Versão Detalhada e Aprimorada

Sumário

\* Visão Geral do Projeto

1.1. Objetivo

1.2. Escopo Inicial

\* Arquitetura Técnica

2.1. Stack e Dependências

2.2. Organização de Pastas

2.3. Scripts de Automação (package.json)

2.4. Banco de Dados e Prisma

2.5. Autenticação & Segurança

2.6. Validação de Entradas

2.7. Comunicação API

2.8. Testes Automatizados

2.9. CI/CD (GitHub Actions)

2.10. Estratégia de Logs e Monitoramento

2.11. Tratamento de Exceções e Erros Globais

2.12. Gerenciamento de Segredos

2.13. Otimização de Performance e Cache

2.14. Segurança Avançada

2.15. Gerenciamento de Estado no Frontend

2.16. Internacionalização (i18n) e Localização (l10n)

\* Fluxo de Desenvolvimento Local

\* Estratégia de Deployment e Infraestrutura (DevOps)

4.1. Ambientes

4.2. Fluxo de Deployment

4.3. Infraestrutura Sugerida

\* Documentação Técnica

\* Proposta Negocial

6.1. Fases e Entregáveis

6.2. Investimento e Condições de Pagamento

1. Visão Geral do Projeto

1.1. Objetivo

Estabelecer a base sólida e escalável para o sistema financeiro EveryFin, garantindo padrões de qualidade, segurança, performance e manutenibilidade desde o início do ciclo de vida do desenvolvimento de software. Isso inclui a criação de um ambiente de desenvolvimento eficiente e a definição de uma arquitetura robusta para suportar futuras expansões e integrações.

1.2. Escopo Inicial

\* Configuração de ambiente de desenvolvimento local completo (backend e frontend).

\* Estruturação de banco de dados relacional (PostgreSQL) com ORM Prisma, incluindo modelos iniciais (User, Account, Transaction, Report), migrações versionadas e seed de dados para testes.

\* Implementação de autenticação e autorização robustas (JWT, refresh tokens, hash de senhas com bcrypt e middleware de verificação de token).

\* Validação rigorosa de dados usando Zod, com tratamento centralizado de erros tanto no backend quanto no frontend.

\* Arquitetura de frontend em React+TypeScript com Tailwind CSS, incluindo organização de pastas, roteamento e comunicação API via Axios.

\* CI/CD básico (testes, lint, build) via GitHub Actions, preparando o terreno para fluxos de deployment automatizados em múltiplos ambientes.

\* Documentação técnica e negocial abrangente para sustentação, onboarding de novos membros e negociação com stakeholders.

2. Arquitetura Técnica

2.1. Stack e Dependências

A escolha da stack tecnológica visa combinar performance, produtividade do desenvolvedor, segurança e um ecossistema maduro para garantir a longevidade e escalabilidade do EveryFin.

Backend: Node.js (v18+), TypeScript, Express

\* Dependências principais:

\* express: Framework web para construção de APIs RESTful.

\* cors: Middleware para habilitar Cross-Origin Resource Sharing.

\* dotenv: Carrega variáveis de ambiente de um arquivo .env.

\* bcrypt: Biblioteca para hash de senhas de forma segura (com 12 salt rounds recomendados).

\* jsonwebtoken: Implementação de JSON Web Tokens (JWT) para autenticação e autorização.

\* zod: Biblioteca de validação de esquemas para garantir a integridade dos dados de entrada.

\* prisma e @prisma/client: ORM moderno para interação com o banco de dados PostgreSQL.

\* winston: Biblioteca robusta para logging, permitindo logs estruturados e configuráveis.

\* DevDependencies:

\* typescript: Transpilador de JavaScript para adicionar tipagem estática.

\* ts-node-dev: Permite hot-reloading para um fluxo de desenvolvimento mais rápido no backend.

\* eslint: Ferramenta para identificar e reportar padrões problemáticos no código (linting).

\* prettier: Formatador de código para garantir consistência e legibilidade.

\* jest: Framework de testes unitários e de integração.

\* ts-jest: Adaptador para Jest rodar testes com TypeScript.

\* supertest: Biblioteca para testar APIs HTTP, facilitando testes de integração com Express.

Banco de Dados: PostgreSQL 14+ gerenciado via Prisma.

Frontend: React (v18), TypeScript, Tailwind CSS

\* Dependências principais:

\* react-router-dom: Para roteamento declarativo no React.

\* axios: Cliente HTTP baseado em Promises para comunicação com APIs.

\* zod: Para validação de dados no lado do cliente, espelhando a validação do backend.

\* react-hook-form: Biblioteca para gerenciamento de formulários com validação e hooks.

\* DevDependencies:

\* eslint: Para linting do código React e TypeScript.

\* prettier: Para formatação consistente do código frontend.

\* jest: Framework de testes para o frontend.

\* @testing-library/react: Utilitários para testar componentes React de forma eficaz, focando no comportamento do usuário.

Integrações:

\* JWT para autenticação e autorização segura.

\* Zod para validação de dados em todas as camadas da aplicação.

\* Axios com interceptors para comunicação API padronizada e tratamento de erros.

2.2. Organização de Pastas

Uma estrutura de pastas bem definida promove a modularidade, facilita a navegação e o entendimento do projeto por novos desenvolvedores, e apoia a manutenibilidade.

/

├── backend/

│ ├── src/

│ │ ├── config/ # Configurações gerais da aplicação (ambiente, banco de dados)

│ │ ├── controllers/ # Lógica de negócio que processa requisições HTTP e invoca services

│ │ ├── middlewares/ # Funções intermediárias (autenticação, CORS, tratamento de erros)

│ │ ├── routes/ # Definição das rotas da API e seus respectivos controllers

│ │ ├── services/ # Lógica de negócio principal, separada de controllers e acessando o ORM

│ │ ├── utils/ # Funções utilitárias e helpers genéricos (ex: formatadores de data)

│ │ └── index.ts # Ponto de entrada da aplicação backend

│ ├── prisma/

│ │ ├── migrations/ # Migrações do banco de dados geradas pelo Prisma

│ │ └── seed.ts # Script para popular o banco de dados com dados de teste

│ ├── tests/ # Testes de unidade e integração do backend

│ ├── .env.example # Exemplo de variáveis de ambiente

│ ├── package.json # Metadados e scripts do backend

│ ├── tsconfig.json # Configurações do TypeScript para o backend

│ └── prisma.schema # Esquema do banco de dados definido pelo Prisma

│

├── frontend/

│ ├── src/

│ │ ├── assets/ # Imagens, ícones, fontes

│ │ ├── components/ # Componentes React reutilizáveis (botões, cards, modais)

│ │ ├── contexts/ # Context API para gerenciamento de estado global (autenticação, tema)

│ │ ├── pages/ # Páginas da aplicação (Login, Dashboard, Transações)

│ │ ├── routes/ # Definição das rotas do frontend

│ │ ├── services/ # Serviços para comunicação com a API (authService, transactionService)

│ │ ├── utils/ # Funções utilitárias e helpers específicos do frontend

│ │ └── App.tsx # Componente raiz da aplicação

│ ├── tests/ # Testes de unidade e integração do frontend

│ ├── .env.local # Variáveis de ambiente locais do frontend

│ ├── tailwind.config.ts # Configurações do Tailwind CSS

│ ├── postcss.config.js # Configurações do PostCSS para Tailwind

│ ├── package.json # Metadados e scripts do frontend

│ └── tsconfig.json # Configurações do TypeScript para o frontend

│

└── docs/

├── arquitetura\_resumida.pdf # Documento de arquitetura de alto nível

├── arquitetura\_detalhada.pdf # Documento de arquitetura aprofundado

└── manual\_instalacao.pdf # Guia passo a passo para setup do ambiente

2.3. Scripts de Automação (package.json)

Os scripts automatizam tarefas comuns de desenvolvimento, build, testes e qualidade de código, garantindo consistência e eficiência.

Backend:

"scripts": {

"dev": "ts-node-dev --respawn src/index.ts",

"build": "tsc",

"start": "node dist/index.js",

"prisma:generate": "prisma generate",

"prisma:migrate": "prisma migrate dev",

"prisma:seed": "ts-node-dev prisma/seed.ts",

"lint": "eslint './src/\*\*/\*.{ts, tsx}' --fix",

"format": "prettier --write './\*\*/\*.{ts,tsx,json,md}'",

"test": "jest --coverage",

"test:watch": "jest --watchAll"

}

\* dev: Inicia o servidor de desenvolvimento com hot-reloading.

\* build: Compila o código TypeScript para JavaScript.

\* start: Inicia a aplicação compilada em ambiente de produção.

\* prisma:generate: Gera o Prisma Client com base no prisma.schema.

\* prisma:migrate: Executa migrações de banco de dados em ambiente de desenvolvimento, gerando novos arquivos de migração.

\* prisma:seed: Popula o banco de dados com dados iniciais ou de teste.

\* lint: Executa o ESLint para verificar e corrigir problemas de código.

\* format: Formata o código com Prettier.

\* test: Executa todos os testes com relatório de cobertura.

\* test:watch: Executa testes e os observa para mudanças nos arquivos.

Frontend:

"scripts": {

"dev": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"lint": "eslint './src/\*\*/\*.{ts, tsx}' --fix",

"format": "prettier --write './src/\*\*/\*.{ts, tsx,css,md}'",

"preview": "serve -s build" # Adicionado para pré-visualizar o build localmente

}

\* dev: Inicia o servidor de desenvolvimento React.

\* build: Cria a versão otimizada para produção da aplicação React.

\* test: Executa os testes do frontend.

\* lint: Executa o ESLint para verificar e corrigir problemas de código.

\* format: Formata o código com Prettier.

\* preview: Serve a aplicação compilada para testes locais de produção (requer serve instalado globalmente).

2.4. Banco de Dados e Prisma

O PostgreSQL como banco de dados relacional oferece robustez, confiabilidade e suporte a transações complexas, sendo ideal para sistemas financeiros. O Prisma ORM simplifica a interação com o banco de dados e oferece um fluxo de trabalho seguro e baseado em esquema.

\* Modelos Essenciais: O esquema do Prisma definirá os seguintes modelos principais:

\* User: Armazena informações de usuários (ID, nome, email, senha, etc.).

\* Account: Representa as contas financeiras dos usuários (ID, saldo, tipo de conta, relação com User).

\* Transaction: Detalhes de cada transação (ID, valor, tipo (receita/despesa), categoria, data, descrição, relação com Account).

\* Report: Modelos para relatórios financeiros agregados ou configurações de relatório.

\* Migrações Versionadas: O Prisma Migrate será utilizado para gerenciar as alterações no esquema do banco de dados de forma controlada e versionada. Isso garante que as mudanças de esquema sejam aplicadas de forma consistente em todos os ambientes.

\* Seed de Dados: Um script de seeding será desenvolvido para popular o banco de dados com dados de teste e desenvolvimento, facilitando o início rápido e a execução de testes.

\* Controle de Transações: O Prisma oferece suporte a transações programáticas, essenciais para operações financeiras que exigem atomicidade (ACID), garantindo que um conjunto de operações de banco de dados seja concluído com sucesso ou revertido em caso de falha.

2.5. Autenticação & Segurança

A segurança é primordial para um sistema financeiro. O EveryFin implementará um sistema de autenticação e autorização baseado em padrões da indústria.

\* JWT (JSON Web Tokens): Utilizado para controle de acesso. Um token de acesso de curta duração será fornecido após o login bem-sucedido.

\* Refresh Tokens: Um refresh token de longa duração será usado para obter novos tokens de acesso quando o token atual expirar, minimizando a frequência de logins e aumentando a segurança ao evitar que tokens de acesso comprometidos sejam válidos por muito tempo.

\* Hash de Senhas com bcrypt: As senhas dos usuários nunca serão armazenadas em texto simples. bcrypt será usado para hash seguro, com um fator de custo (salt rounds) de 12, que é um bom equilíbrio entre segurança e performance para 2025.

\* Middleware de Verificação de Token: Um middleware dedicado no backend verificará a validade do JWT para proteger rotas autenticadas, garantindo que apenas usuários autorizados possam acessar recursos restritos.

\* Estratégia de Revogação de Tokens: Implementar uma forma de invalidar refresh tokens em caso de logout ou comprometimento (por exemplo, armazenando refresh tokens em uma lista de revogação no banco de dados ou Redis).

2.6. Validação de Entradas

A validação de dados é crucial para a integridade do sistema e para prevenir ataques.

\* Zod no Backend e Frontend: Zod será a biblioteca padrão para definir esquemas de validação robustos para todos os dados de entrada (payloads de API, parâmetros de rota, queries).

\* Tratamento Centralizado de Erros: As validações do Zod serão integradas a um middleware de tratamento de erros global no Express, que capturará erros de validação e formatará respostas de erro padronizadas (ex: status HTTP 400 Bad Request com detalhes dos campos inválidos), proporcionando uma experiência consistente para o frontend e para o logging.

2.7. Comunicação API

A comunicação entre frontend e backend será padronizada e robusta.

\* Axios com Interceptors: Axios será o cliente HTTP principal no frontend. Interceptors serão configurados para:

\* Anexar o token de acesso JWT a todas as requisições autenticadas.

\* Lidar com erros de requisição globalmente (ex: redirecionar para login em caso de 401 Unauthorized, exibir mensagens de erro genéricas para o usuário).

\* Adicionar headers padrão (ex: Content-Type: application/json).

\* Serviços Modulados: A lógica de comunicação com a API será encapsulada em serviços dedicados (ex: authService.ts, transactionService.ts, reportService.ts), promovendo a reutilização de código e a modularidade.

2.8. Testes Automatizados

Uma cultura de testes automatizados será cultivada para garantir a qualidade, identificar regressões precocemente e facilitar a refatoração.

\* Backend: Jest + Supertest:

\* Testes Unitários: Serão escritos para funções isoladas (ex: utilitários, validadores, lógica de negócios em serviços).

\* Testes de Integração: Supertest será usado para testar as rotas da API, simulando requisições HTTP e verificando as respostas e o impacto no banco de dados.

\* Cobertura de Testes: Atingir uma cobertura de testes de no mínimo 80% é uma meta para as áreas críticas do sistema.

\* Frontend: React Testing Library:

\* Testes Unitários de Componentes: Foco em testar o comportamento dos componentes da perspectiva do usuário, garantindo que eles renderizem corretamente e respondam a interações.

\* Testes de Integração de Páginas/Fluxos: Testar a interação entre múltiplos componentes e a integração com os serviços de API mockados.

2.9. CI/CD (GitHub Actions)

O pipeline de Integração Contínua e Entrega Contínua (CI/CD) automatizará o processo de verificação, build e, futuramente, deployment do código.

\* Workflow ci.yml: Um workflow inicial será configurado no GitHub Actions para:

\* Linting: Executar ESLint para garantir a qualidade e o estilo do código.

\* Testes: Rodar os testes unitários e de integração do backend e frontend.

\* Build: Compilar a aplicação backend (TypeScript) e frontend (React).

\* Relatório de Testes: Publicar relatórios de cobertura de testes.

\* Verificação de Segurança: Adicionar passos básicos de varredura de vulnerabilidades em dependências (ex: npm audit).

\* Branch Protection Rules: Configurar regras de proteção para branches principais (ex: main, develop) para exigir que o pipeline de CI seja bem-sucedido antes que o código possa ser mesclado.

2.10. Estratégia de Logs e Monitoramento

Para um sistema financeiro, logs detalhados e monitoramento em tempo real são cruciais para depuração, auditoria e identificação proativa de problemas.

\* Logs Estruturados: Utilizar winston no backend para gerar logs em formato JSON ou similar, contendo informações como timestamp, nível do log (info, warn, error), mensagem, contexto (ID da requisição, ID do usuário) e metadados relevantes. Isso facilita a análise e a busca em ferramentas de agregação de logs.

\* Níveis de Log: Implementar diferentes níveis de log (debug, info, warn, error, fatal) para controlar a verbosidade em diferentes ambientes.

\* Agregação de Logs: Para ambientes de produção, os logs serão coletados e enviados para um serviço centralizado de agregação de logs (ex: ELK Stack - Elasticsearch, Logstash, Kibana; ou soluções de cloud como AWS CloudWatch Logs, Google Cloud Logging).

\* Monitoramento de Métricas:

\* Backend: Coletar métricas como latência de requisição, erros por segundo, uso de CPU/memória, número de requisições, e saúde do banco de dados. Ferramentas como Prometheus com Grafana (para dashboards) ou Datadog podem ser utilizadas.

\* Frontend: Monitorar métricas de performance da web (Core Web Vitals), erros de JavaScript e uso de recursos.

\* Alertas: Configurar alertas com base em limiares definidos para métricas críticas (ex: alta taxa de erros, latência excessiva, uso de recursos elevado) para notificar a equipe de operações.

\* APM (Application Performance Monitoring): Avaliar a inclusão de uma ferramenta de APM (ex: New Relic, Dynatrace, Sentry) para rastreamento de transações ponta a ponta, identificação de gargalos de performance e depuração distribuída.

2.11. Tratamento de Exceções e Erros Globais

Uma abordagem unificada para o tratamento de erros garante que a aplicação se comporte de forma previsível e forneça informações úteis para depuração, sem expor detalhes sensíveis.

\* Backend:

\* Middleware de Erro Global: Um middleware de erro no Express será o último a ser executado, capturando todas as exceções não tratadas e erros lançados na aplicação.

\* Padronização de Respostas de Erro: Todas as respostas de erro da API seguirão um formato JSON consistente (ex: { "status": "error", "message": "...", "code": "...", "details": [...] }).

\* Erros Personalizados: Criar classes de erro personalizadas (ex: NotFoundError, UnauthorizedError, ValidationError) para semântica de erro clara e fácil distinção.

\* Logging de Erros: Todos os erros, especialmente os de nível error e fatal, serão logados com contexto suficiente para depuração (stack trace, payload da requisição, ID do usuário).

\* Supressão de Detalhes em Produção: Em produção, os detalhes técnicos de erros (stack traces) não serão expostos ao cliente final, apenas mensagens amigáveis e um ID de correlação para rastreamento interno.

\* Frontend:

\* Boundary de Erros (React Error Boundaries): Utilizar Error Boundaries para capturar erros em componentes React e exibir uma UI de fallback em vez de quebrar a aplicação inteira.

\* Envio de Erros ao Servidor: Erros de JavaScript não tratados no frontend serão capturados (ex: usando uma ferramenta como Sentry ou enviando para um endpoint de log do backend) para análise e depuração.

\* Mensagens de Erro Amigáveis: Traduzir erros da API em mensagens compreensíveis para o usuário final.

2.12. Gerenciamento de Segredos

O gerenciamento seguro de chaves de API, senhas de banco de dados e outros segredos é fundamental para a segurança da aplicação.

\* Variáveis de Ambiente (.env): Para o desenvolvimento local, segredos serão armazenados em arquivos .env (que não serão versionados no Git).

\* Variáveis de Ambiente do Sistema Operacional/Ambiente de Deployment: Em ambientes de staging e produção, os segredos serão injetados como variáveis de ambiente pelo sistema operacional ou pela plataforma de deployment (ex: AWS Elastic Beanstalk, Heroku, Kubernetes Secrets).

\* Ferramentas de Gerenciamento de Segredos (Produção): Para maior segurança e conformidade em produção, considerar ferramentas dedicadas de gerenciamento de segredos:

\* HashiCorp Vault: Para gerenciamento centralizado de segredos, rotação automática e controle de acesso.

\* AWS Secrets Manager / Google Cloud Secret Manager / Azure Key Vault: Serviços de cloud para armazenamento e gerenciamento de segredos.

\* Não Hardcode Segredos: Em hipótese alguma segredos serão codificados diretamente no código-fonte.

2.13. Otimização de Performance e Cache

A performance é crítica para a experiência do usuário, especialmente em aplicações financeiras com grande volume de dados.

\* Otimização de Consultas SQL (Prisma):

\* Indexes: Garantir que índices adequados sejam criados nas colunas de banco de dados frequentemente consultadas (ex: foreign keys, colunas usadas em cláusulas WHERE, ORDER BY).

\* N+1 Problem: Evitar o problema N+1 de consultas ao utilizar o carregamento "eager" (ex: include no Prisma) para carregar relacionamentos de forma eficiente.

\* Paginação: Implementar paginação em listas longas (ex: transações, relatórios) para reduzir a carga no banco de dados e a transferência de dados.

\* Cache de Dados (Redis):

\* Cache de Leitura: Utilizar Redis como um cache em memória para dados frequentemente acessados e que não mudam com muita frequência (ex: configurações do sistema, dados de referência, resultados de relatórios complexos).

\* Cache de Sessão/JWT Revogação: Redis também pode ser usado para armazenar sessões de usuário ou para a lista de revogação de tokens JWT/refresh tokens, melhorando a performance e a segurança.

\* Minificação e Bundling (Frontend): O react-scripts build já realiza automaticamente a minificação e bundling do código JavaScript, CSS e HTML, otimizando o carregamento da aplicação.

\* Compressão Gzip/Brotli: Configurar o servidor (ou CDN) para servir os assets estáticos com compressão Gzip ou Brotli para reduzir o tamanho dos arquivos transferidos.

\* Otimização de Imagens: Comprimir e otimizar imagens, utilizando formatos modernos (WebP) e tamanhos adequados.

2.14. Segurança Avançada

Além da autenticação e validação, outras medidas de segurança serão implementadas para proteger contra um espectro mais amplo de ataques.

\* Proteção contra XSS (Cross-Site Scripting):

\* Sanitização de Input: No backend, validar e sanitizar todas as entradas de usuário antes de armazenar ou exibir (ex: usar bibliotecas como DOMPurify se o input puder conter HTML).

\* Codificação de Output: No frontend, certificar-se de que todo o conteúdo gerado pelo usuário seja devidamente escapado antes de ser renderizado no HTML (React já faz isso por padrão, mas é bom reforçar).

\* Proteção contra CSRF (Cross-Site Request Forgery):

\* Tokens CSRF: Implementar tokens CSRF para requisições que modificam o estado (POST, PUT, DELETE), garantindo que a requisição seja originada da aplicação legítima.

\* SameSite Cookies: Configurar cookies de sessão/JWT com o atributo SameSite=Lax ou Strict.

\* Rate Limiting: Implementar rate limiting nas rotas críticas (ex: login, registro de usuário, envio de transações) para prevenir ataques de força bruta e abuso de API.

\* Content Security Policy (CSP): Configurar um cabeçalho CSP no backend para mitigar ataques XSS e de injeção de dados, controlando de onde os recursos (scripts, estilos, imagens) podem ser carregados.

\* OWASP Top 10: Referenciar as diretrizes do OWASP Top 10 para identificar e mitigar as vulnerabilidades de segurança mais comuns em aplicações web.

\* Hardening de Banco de Dados:

\* Princípio do Menor Privilégio: Configurar as credenciais do banco de dados com o mínimo de privilégios necessários para as operações da aplicação.

\* Conexões Criptografadas: Garantir que a conexão entre a aplicação e o banco de dados seja criptografada (SSL/TLS).

\* Backup e Restauração: Definir uma estratégia de backup e restauração do banco de dados.

2.15. Gerenciamento de Estado no Frontend

O gerenciamento de estado no React será abordado para garantir uma aplicação responsiva e manutenível.

\* Context API: Para estados globais menos complexos e compartilhados por muitos componentes (ex: tema, informações do usuário autenticado).

\* Hooks de Estado Local (useState, useReducer): Para estados específicos de componentes.

\* Bibliotecas para Estado Global (Opcional, para escala futura): Para aplicações financeiras com grande volume de dados e lógica de estado complexa, considerar bibliotecas como Zustand, Jotai, Recoil (alternativas mais leves ao Redux) ou o próprio Redux para um gerenciamento de estado mais robusto e previsível. A escolha dependerá da complexidade e escala futuras percebidas.

\* react-query / SWR: Para gerenciamento de cache de dados e sincronização de estado com o servidor, reduzindo a necessidade de estado global complexo para dados de API.

2.16. Internacionalização (i18n) e Localização (l10n)

Para um sistema financeiro, a capacidade de suportar múltiplos idiomas e formatos monetários/de data é uma consideração importante para o futuro.

\* Bibliotecas de i18n: Integrar uma biblioteca como react-i18next para gerenciar as traduções de textos na interface do usuário.

\* Formatadores de Localização: Utilizar APIs nativas de JavaScript (Intl.NumberFormat, Intl.DateTimeFormat) ou bibliotecas como date-fns ou moment.js (com cautela, preferindo date-fns para modernidade) para formatar valores monetários, datas e horas de acordo com a localidade do usuário.

\* Moeda Padrão vs. Multi-moeda: Definir se o sistema operará com uma única moeda padrão ou se haverá suporte para transações em múltiplas moedas com conversão de taxa de câmbio.

3. Fluxo de Desenvolvimento Local

O fluxo de desenvolvimento local será otimizado para garantir que os desenvolvedores possam configurar o ambiente e começar a codificar rapidamente.

\* Clonar repositório: git clone [URL\_DO\_REPOSITORIO]

\* Configurar variáveis de ambiente: Copiar .env.example para .env (backend) e .env.local (frontend) e preencher com as informações necessárias (chaves de API, credenciais de banco de dados).

\* Configurar e iniciar backend:

\* Navegar para /backend.

\* Instalar dependências: npm install.

\* Gerar Prisma Client: npm run prisma:generate.

\* Executar migrações do banco de dados: npm run prisma:migrate.

\* Popular banco de dados com dados iniciais: npm run prisma:seed.

\* Iniciar o servidor de desenvolvimento: npm run dev.

\* Configurar e iniciar frontend:

\* Navegar para /frontend.

\* Instalar dependências: npm install.

\* Iniciar o servidor de desenvolvimento: npm run dev.

\* Executar testes: Durante o desenvolvimento, é fundamental executar testes unitários e de integração para garantir a qualidade do código.

\* npm test (no backend e frontend).

\* Lint e formatação: Utilizar os scripts de lint e format para manter a consistência do código.

\* npm run lint --fix

\* npm run format

4. Estratégia de Deployment e Infraestrutura (DevOps)

Uma estratégia clara de deployment e infraestrutura é essencial para a entrega contínua e a operação confiável do EveryFin.

4.1. Ambientes

Serão definidos três ambientes principais para o ciclo de vida do software:

\* dev (Desenvolvimento): Ambiente local de cada desenvolvedor.

\* staging (Homologação/Teste): Ambiente que espelha a produção o mais fielmente possível, usado para testes de integração, testes de aceitação do usuário (UAT) e demonstrações.

\* prod (Produção): Ambiente onde a aplicação está em operação e acessível aos usuários finais.

4.2. Fluxo de Deployment

O GitHub Actions será estendido para gerenciar os deployments para os ambientes de staging e produção.

\* Deployment para Staging:

\* Acionado por merges na branch develop ou por um comando manual no GitHub Actions.

\* O pipeline de CI/CD executará testes, lint e build.

\* Em seguida, a aplicação será empacotada (ex: Docker image) e implantada automaticamente no ambiente de staging.

\* Deployment para Produção:

\* Acionado por merges na branch main (após revisão e aprovação do staging).

\* Um processo manual de aprovação ou tag de release pode ser exigido.

\* O pipeline de CI/CD realizará novamente testes e build finais.

\* A aplicação será empacotada e implantada na infraestrutura de produção com zero-downtime, se possível.

4.3. Infraestrutura Sugerida

A infraestrutura será baseada em serviços de cloud para flexibilidade, escalabilidade e custo-benefício.

\* Provedor de Nuvem: Recomenda-se um provedor como AWS (Amazon Web Services), Google Cloud Platform (GCP) ou Microsoft Azure devido à sua robustez, vasta gama de serviços e escalabilidade.

\* Backend Hosting:

\* Opção 1 (Containerização): AWS ECS/EKS (com Docker) ou Google Kubernetes Engine (GKE) para orquestração de contêineres, proporcionando escalabilidade e resiliência.

\* Opção 2 (Serverless): AWS Lambda com API Gateway ou Google Cloud Functions para APIs serverless, ideal para custos otimizados e escalabilidade automática para cargas de trabalho irregulares.

\* Opção 3 (PaaS): AWS Elastic Beanstalk ou Heroku para um deployment mais gerenciado e simples.

\* Frontend Hosting:

\* AWS S3 + CloudFront: Para servir os assets estáticos do frontend de forma global e com alta performance via CDN.

\* Google Cloud Storage + Cloud CDN: Equivalente ao S3 + CloudFront.

\* Netlify / Vercel: Para deployments de frontend gerenciados e muito eficientes, especialmente para aplicações React.

\* Banco de Dados:

\* AWS RDS (PostgreSQL): Serviço de banco de dados relacional gerenciado, oferecendo backups automáticos, escalabilidade e alta disponibilidade.

\* Google Cloud SQL (PostgreSQL): Serviço equivalente no GCP.

\* Redis (para Cache/Sessão):

\* AWS ElastiCache (Redis): Serviço gerenciado de cache em memória.

\* Google Cloud Memorystore (Redis): Serviço equivalente no GCP.

\* Ferramentas de Monitoramento e Logs:

\* AWS CloudWatch / GCP Cloud Logging: Para coleta e análise de logs da infraestrutura.

\* Sentry: Para monitoramento de erros de aplicação.

\* Grafana + Prometheus: Para dashboards e alertas de métricas customizadas, se hospedado em VMs.

5. Documentação Técnica

Uma documentação completa é vital para a manutenção, escalabilidade e onboarding de novos membros da equipe.

\* Arquitetura Resumida (arquitetura\_resumida.pdf): Um documento de alto nível que apresenta a visão geral da arquitetura do sistema, os principais componentes e as interações. Ideal para stakeholders não técnicos e como um ponto de partida.

\* Arquitetura Detalhada (arquitetura\_detalhada.pdf): Um documento técnico aprofundado que descreve em detalhes cada componente da arquitetura, incluindo diagramas (componentes, implantação, sequência), decisões de design, justificativas técnicas e padrões aplicados.

\* Manual de Instalação (manual\_instalacao.pdf): Um guia passo a passo para configurar o ambiente de desenvolvimento local para backend e frontend, incluindo pré-requisitos, instalação de dependências e execução de scripts. Essencial para o onboarding rápido de novos desenvolvedores.

\* Documentação da API (Swagger/OpenAPI): Gerar automaticamente ou manualmente uma documentação interativa da API RESTful (utilizando ferramentas como Swagger/OpenAPI Spec) para facilitar o consumo pelo frontend e por futuras integrações.

6. Proposta Negocial

6.1. Fases e Entregáveis

O projeto será dividido em fases claras, com entregáveis bem definidos e um cronograma estimado.

\* Fase 1: Setup Inicial (1 semana)

\* Entregáveis: Repositórios Git configurados, estrutura de pastas inicial, scripts de automação (package.json), configuração de ambiente de desenvolvimento local para backend e frontend, esquema inicial do banco de dados (Prisma).

\* Fase 2: Autenticação & Usuários (2 semanas)

\* Entregáveis: Cadastro e login de usuários, autenticação JWT com refresh tokens, gerenciamento de perfis de usuário básicos (edição de senha/email), hashing de senhas com bcrypt, rotas protegidas por middleware de autenticação, testes unitários e de integração para autenticação.

\* Fase 3: Transações CRUD (3 semanas)

\* Entregáveis: Funcionalidades CRUD para transações financeiras (adicionar, visualizar, editar, excluir transações), validação de dados com Zod, listagem de transações com filtros e paginação, categorias de transações, testes abrangentes para as funcionalidades de transações.

\* Fase 4: Dashboard & Relatórios (3 semanas)

\* Entregáveis: Dashboard com resumo financeiro (saldo, receitas/despesas por período), relatórios básicos (ex: extrato por período, despesas por categoria), visualização de dados gráficos simples (ex: gráficos de barra/pizza), exportação de relatórios (ex: CSV).

\* Fase 5: Testes & QA (2 semanas)

\* Entregáveis: Execução de testes de integração ponta a ponta, testes de aceitação do usuário (UAT) com stakeholders, refinamento de testes unitários e de integração, correção de bugs identificados, testes de performance e segurança (básico).

\* Fase 6: Documentação & Deploy (1 semana)

\* Entregáveis: Finalização da documentação técnica (arquitetura, manual de instalação), documentação da API, configuração completa do pipeline de CI/CD para deploy em ambiente de staging e produção, treinamento inicial para o cliente sobre o uso e sustentação básica.

6.2. Investimento e Condições de Pagamento

A proposta de investimento será estruturada para garantir a transparência e o compromisso mútuo.

\* Valor Fixo por Fase: O projeto terá um valor fixo acordado para cada fase, garantindo previsibilidade de custos.

\* Percentuais de Pagamento:

\* Assinatura do Contrato: Um percentual do valor total como adiantamento para cobrir os custos iniciais e garantir o comprometimento.

\* Meio do Projeto (ex: Após Fase 3): Um percentual do valor total após a conclusão de marcos importantes, como a implementação das funcionalidades de transações.

\* Entrega Final do Projeto: O percentual restante após a conclusão de todas as fases, testes e deploy em produção.

\* Condições de Pagamento: Detalhes sobre os prazos de pagamento, métodos aceitos e eventuais multas por atraso.

\* Termos e Condições: Cláusulas sobre escopo, alterações, suporte pós-entrega e garantia.

Este documento revisado e expandido visa proporcionar máxima clareza, robustez e alinhamento com as melhores práticas de desenvolvimento de software, servindo como um guia abrangente para o sucesso do projeto EveryFin.