Guia de Desenvolvimento da Rede Social P2P Modular

Sumário

1. Introdução e Visão Geral do Projeto

Este guia destina-se à equipe de desenvolvimento da Rede Social P2P Modular. Ele fornece as informações necessárias para configurar o ambiente de desenvolvimento, entender a arquitetura do projeto, seguir os padrões de codificação e colaborar efetivamente.

1.1. Objetivos do Projeto

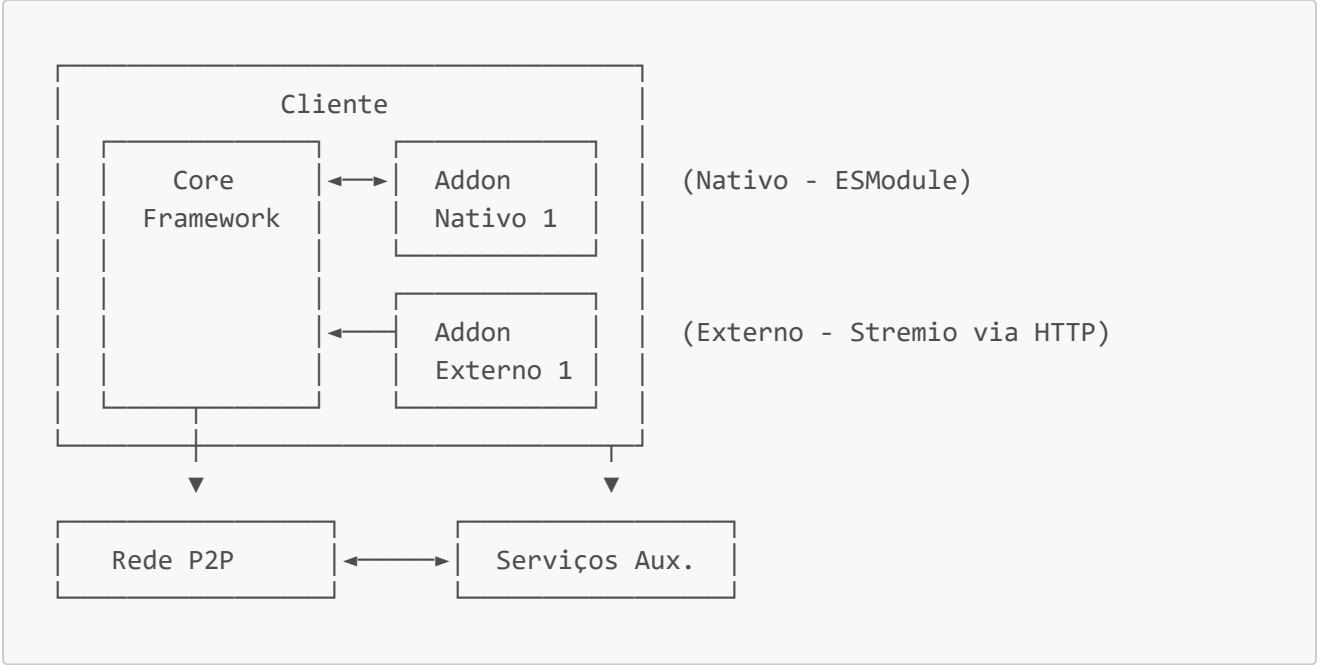
A Rede Social P2P Modular visa criar uma plataforma descentralizada e extensível onde os usuários podem compartilhar diversos tipos de conteúdo e interagir socialmente sem depender de servidores centrais. Os principais objetivos incluem:

- **Descentralização Real:** Construir um sistema onde os usuários mantêm controle sobre seus dados e identidade.
- **Modularidade Extrema:** Permitir que a funcionalidade seja expandida através de um sistema de addons, similar ao Stremio, fomentando um ecossistema de desenvolvimento comunitário.
- **Interoperabilidade:** Ser capaz de interagir com ecossistemas de addons existentes (como os do Stremio para mídia) para enriquecer o catálogo de funcionalidades desde o início.
- **Experiência de Usuário Fluida:** Oferecer uma interface intuitiva e performática, mesmo com a complexidade da P2P e a diversidade de addons.
- **Foco no P2P:** Utilizar tecnologias P2P para comunicação, armazenamento e distribuição de conteúdo, incluindo streaming de mídia (ex: WebTorrent).
- **Comunidade Forte:** Construir uma comunidade ativa de usuários e desenvolvedores.

(Para mais detalhes, consulte a Seção 1 do `projeto-rede-social-p2p.md`)

1.2. Arquitetura Macro

A plataforma é composta por um Core Framework que gerencia a identidade, a rede P2P, o armazenamento e o ciclo de vida dos addons. Os addons podem ser nativos (construídos especificamente para a plataforma) ou externos (como os do Stremio, consumidos via adaptadores).



Componentes Chave:

- **Core Framework:** Orquestra todas as operações, gerencia addons, e fornece a `CoreAPI` para addons nativos.
- **Addons Nativos:** Módulos ESModule que rodam em sandbox e estendem funcionalidades usando a `CoreAPI`.
- **Addons Externos (via Adaptadores):** Permitem consumir funcionalidades de plataformas como Stremio, principalmente para descoberta de mídia.
- **Rede P2P:** Camada de comunicação, armazenamento (IPFS, OrbitDB) e streaming (WebTorrent).
- **Serviços Auxiliares (Opcionais):** Discovery servers, relays, etc., para otimizar a experiência P2P.

(Consulte a Seção 2 do `projeto-rede-social-p2p.md` para a arquitetura completa.)

1.3. Stack Tecnológico Chave

A seguir, a stack tecnológica principal definida para o projeto:

- **Frontend (Web e Mobile):**
  - Framework: `React Native` + `Expo` (com suporte web via `@expo/webpack-config`)
  - Design System: `Tamagui` ou `NativeWind` (com base em `shadcn/ui`)
  - Player de Mídia: Player robusto com suporte a WebTorrent (ex: `Video.js`, `Plyr`)
  - Gerenciamento de Estado: `Zustand`
- **Comunicação e Rede P2P:**
  - Networking P2P: `js-libp2p` (e seus submódulos como `@chainsafe/libp2p-gossipsub`, `@libp2p/kad-dht`, etc.). Muitas funcionalidades são integradas e gerenciadas pela instância principal do Libp2p.

    - Streaming P2P: `WebTorrent` (se necessário para streaming de arquivos de mídia).
    - NAT Traversal: `ICE/STUN/TURN` (gerenciados pelo Libp2p, especialmente quando usando o transporte WebRTC).
- **Armazenamento P2P e Sincronização de Dados:**
    - Camada IPFS: `Helia` (uma implementação leve de IPFS que utiliza Libp2p internamente).
    - Banco de Dados P2P (para dados estruturados): `@orbitdb/core` (que opera sobre uma instância Helia/IPFS, utilizando CRDTs).
    - (Alternativa mais leve para dados em tempo real, se necessário no MVP: `Gun.js`)
- **Sistema de Addons:**
    - Addons Nativos: `ESModules` em `Web Workers`/`Iframes`, `MessageChannel API`
    - Addons Externos: Comunicação via `HTTP` com os servidores dos addons
- **Identidade:** Pares de chaves (criptografia de curva elíptica), `DIDs` (Decentralized Identifiers).
- **Ferramentas de Desenvolvimento:**
    - Gestor de Pacotes: `npm`
    - Monorepo: `Turborepo` ou `NX`

(Consulte a Seção 3 do `projeto-rede-social-p2p.md` para a discussão completa da stack.)

2. Configuração do Ambiente de Desenvolvimento

Esta seção descreve os passos para configurar seu ambiente de desenvolvimento local e começar a contribuir para o projeto.

2.1. Pré-requisitos de Software

Antes de começar, certifique-se de que você tem os seguintes softwares instalados em sua máquina:

- **Node.js:** Recomenda-se a versão LTS mais recente. Você pode baixá-lo de https://nodejs.org/ ou usar um gerenciador de versões como `nvm` ou `fnm`.
    - Para verificar sua versão: `node -v`
- **npm:** O Node Package Manager, vem com o Node.js. Usaremos `npm` como nosso gerenciador de pacotes principal.
    - Para verificar sua versão: `npm -v`
- **Git:** Essencial para controle de versão. Você pode baixá-lo de https://git-scm.com/.
    - Para verificar sua versão: `git --version`
- **Editor de Código:** Recomenda-se um editor moderno com bom suporte a TypeScript/JavaScript, como:
    - Visual Studio Code (https://code.visualstudio.com/)
    - Cursor (https://cursor.sh/)
    - WebStorm (ou outros IDEs da JetBrains)
- **Sistema Operacional:** Linux, macOS ou Windows (com WSL2 recomendado para melhor experiência de desenvolvimento).

2.2. Clonando o Repositório

Para obter o código-fonte do projeto, clone o repositório principal (o link será fornecido quando o repositório for criado):

```
# Substitua <URL_DO_REPOSITORIO> pelo link correto
git clone <URL_DO_REPOSITORIO>
cd nome-do-diretorio-do-projeto
```

2.3. Instalação de Dependências

Após clonar o repositório, instale todas as dependências executando o seguinte comando na raiz do projeto:

```
npm install
```

2.4. Configuração de Variáveis de Ambiente

Alguns pacotes ou funcionalidades podem requerer variáveis de ambiente para configuração (ex: chaves de API para serviços auxiliares de desenvolvimento, configurações de peers de bootstrap para P2P, etc.).

1. Na raiz do projeto (ou em pacotes específicos, conforme documentado neles), procure por um arquivo de exemplo de variáveis de ambiente, como `.env.example`.
2. Copie este arquivo para `.env` no mesmo diretório:

   ```
   # Exemplo, pode variar dependendo da estrutura final
   cp packages/core-p2p-lib/.env.example packages/core-p2p-lib/.env
   ```

3. Edite o arquivo `.env` recém-criado e preencha os valores das variáveis conforme necessário para o seu ambiente de desenvolvimento.

Nota: O arquivo `.env` é geralmente incluído no `.gitignore` para evitar que segredos sejam commitados no repositório.

Com esses passos, seu ambiente de desenvolvimento deve estar pronto. Para iniciar a aplicação ou pacotes específicos, consulte os scripts definidos no `package.json` na raiz do projeto ou nos `package.json` dos pacotes individuais (ex: `npm run dev`).

2.5. Observações Importantes sobre Dependências

Considerando a rápida evolução do ecossistema P2P e as experiências passadas no início deste projeto, é crucial ter um cuidado especial ao adicionar ou atualizar dependências:

- **Verifique a Atualidade e Compatibilidade:** Antes de instalar um novo módulo, pesquise se ele está ativamente mantido, se é compatível com as versões das nossas dependências principais (Node.js, Libp2p, Helia, OrbitDB) e se sua API é estável.
- **Real Necessidade:** Avalie se a funcionalidade desejada realmente requer um novo módulo externo ou se pode ser implementada com as bibliotecas já presentes no projeto. Muitas funcionalidades

do ecossistema IPFS/Libp2p estão se consolidando. Por exemplo, `Helia` já fornece uma camada IPFS sobre `Libp2p`, e o `@orbitdb/core` moderno interage bem com essa stack.
- **Experiências Anteriores:** No início do projeto, enfrentamos desafios com módulos desatualizados ou APIs que mudaram significativamente (ex: versões antigas de `libp2p-gossipsub`, diferentes formas de interagir com `OrbitDB` e suas stores). A configuração de dependências atual, visível no `package.json` do projeto, reflete uma stack que foi testada e provou ser funcional e estável para os nossos objetivos.
- **Evite Redundância e Conflitos:** Adicionar módulos desnecessários ou conflitantes pode levar a comportamentos inesperados e dificultar a depuração. Privilegie o uso das funcionalidades já oferecidas pelo Libp2p e suas dependências diretas sempre que possível.

Este cuidado nos ajudará a manter a estabilidade do projeto e a evitar a reintrodução de problemas já superados.

3. Estrutura do Projeto (Core Focado)

O projeto foca no desenvolvimento do `core-p2p-lib`, a biblioteca central da plataforma P2P. A estrutura será a de um projeto Node.js/TypeScript padrão, sem a complexidade de um monorepo nesta fase, pois estamos construindo apenas o core.

3.1. Visão Geral da Estrutura do Core

A estrutura de diretórios planejada para o `core-p2p-lib` é:

```
/
├── src/
│   ├── managers/          # Gerenciadores (NetworkManager, IdentityManager,
etc.)
│   ├── event-bus/         # Sistema de eventos interno
│   ├── interfaces/        # Definições de tipos e interfaces TypeScript
│   ├── utils/             # Funções utilitárias
│   └── index.ts           # Ponto de entrada principal da biblioteca
├── test/                  # Testes unitários e de integração
├── dist/                  # Saída da compilação TypeScript
├── package.json           # Metadados do projeto e dependências
├── tsconfig.json          # Configuração do TypeScript
├── .eslintrc.cjs          # Configuração do ESLint
├── .prettierrc.json       # Configuração do Prettier
└── GUIA_DESENVOLVIMENTO.md # Este guia
```

3.2. Responsabilidades dos Componentes do Core

- **`src/managers/`**: Contém os principais gerenciadores da lógica do core.
  - `IdentityManager`: Gerencia a identidade descentralizada.
  - `NetworkManager`: Lida com a rede P2P via `libp2p`.
  - `StorageManager`: Abstrai o armazenamento P2P.

    - `AddonManager`: Gerencia o ciclo de vida de addons (embora o foco inicial seja no core, a estrutura pode prever isso de forma simplificada).
- `src/event-bus/`: Sistema de eventos para comunicação interna desacoplada.
- `src/interfaces/`: Definições de tipos e interfaces TypeScript.
- `src/index.ts`: Ponto de entrada que inicializa e exporta as funcionalidades do core.

3.3. Como Navegar e Trabalhar no Projeto

- **Comandos `npm` na Raiz:**
    - `npm install`: Instala dependências.
    - `npm run <script>`: Executa um script definido no `package.json` (ex: `npm run build`, `npm run lint`, `npm run test`).
    - `npm start` ou `npm run dev` para iniciar a aplicação/biblioteca em modo de desenvolvimento.

Não utilizaremos ferramentas de monorepo como Turborepo ou NX, dado o foco no desenvolvimento do `core` como um pacote único inicialmente.

As dependências são gerenciadas diretamente pelo `npm` e listadas no `package.json`.

4. Padrões e Convenções de Código

Manter um código consistente e legível é fundamental para a colaboração eficaz e a sustentabilidade do projeto a longo prazo. Todos os desenvolvedores devem aderir aos seguintes padrões e convenções.

4.1. Linguagem Principal: TypeScript

- **TypeScript First:** O projeto será desenvolvido primariamente em **TypeScript**. Isso nos oferece tipagem estática, o que ajuda a pegar erros em tempo de desenvolvimento, melhora a legibilidade do código e facilita o refatoramento.
- **Modo Estrito:** Configure o `tsconfig.json` para usar as opções de checagem estrita (`strict: true` ou as flags individuais como `noImplicitAny`, `strictNullChecks`, etc.) para maximizar os benefícios da tipagem.
- **Tipos Explícitos:** Embora o TypeScript possa inferir muitos tipos, seja explícito com tipos para parâmetros de função, valores de retorno de funções públicas/exportadas e estruturas de dados complexas para melhorar a clareza.

4.2. Estilo de Código: ESLint e Prettier

- **ESLint:** Será usado para análise estática de código para identificar padrões problemáticos e impor regras de estilo de codificação.
    - Uma configuração base do ESLint (`.eslintrc.cjs` ou similar) será fornecida na raiz do projeto.
    - Plugins comuns incluem `@typescript-eslint/eslint-plugin`.
- **Prettier:** Será usado para formatação automática de código, garantindo um estilo visual consistente em todo o projeto.
    - Uma configuração do Prettier (`.prettierrc.json` ou similar e `.prettierignore`) será fornecida na raiz.

    - Recomenda-se integrar o Prettier com o editor de código para formatar ao salvar.
- **Integração:** ESLint e Prettier devem ser configurados para trabalhar juntos harmoniosamente (ex: usando `eslint-config-prettier` para desabilitar regras de estilo do ESLint que conflitam com o Prettier).
- **Hooks de Pré-commit:** O uso de ferramentas como `husky` e `lint-staged` pode ser considerado para rodar ESLint e Prettier automaticamente em arquivos modificados antes de cada commit.

4.3. Convenções de Nomenclatura

- **Variáveis e Funções:** Usar `camelCase`. Ex: `minhaVariavel`, `calcularResultado()`.
- **Constantes:** Usar `UPPER_SNAKE_CASE` para constantes globais ou valores que não mudam. Ex: `MAX_CONNECTIONS`, `DEFAULT_TIMEOUT`.
- **Classes e Tipos/Interfaces TypeScript:** Usar `PascalCase`. Ex: `class UsuarioService { ... }`, `interface OpcoesDePerfil { ... }`.
- **Arquivos e Diretórios:**
    - Usar `kebab-case` para nomes de arquivos e diretórios. Ex: `componente-de-usuario.tsx`, `servicos-de-rede/`.
    - Para componentes React, nomear o arquivo igual ao componente: `MeuComponente.tsx`.
    - Arquivos de teste devem ter o sufixo `.test.ts` ou `.spec.ts` (ex: `utils.test.ts`).
- **Enums TypeScript:** Usar `PascalCase` para o nome do enum e `PascalCase` (ou `UPPER_SNAKE_CASE` se representar constantes) para seus membros. Ex:

```
enum StatusPedido { PENDENTE, PROCESSANDO, CONCLUIDO }
enum PermissaoCore { LerArmazenamento, EscreverArmazenamento }
```

4.4. Comentários e Documentação de Código

- **Quando Comentar:**
    - Comente código que não é óbvio ou que possui lógica complexa.
    - Explique o *porquê* de uma determinada implementação, não apenas o *o quê* (o código já diz o quê).
    - Evite comentários óbvios que apenas repetem o que o código faz (ex: `// incrementa i`).
    - Use `// TODO:` ou `// FIXME:` para marcar áreas que precisam de atenção futura, com uma breve explicação.
- **Estilo de Comentário:**
    - Para comentários de uma linha: `// Seu comentário`
    - Para comentários de múltiplas linhas, prefira múltiplas linhas de `//` ou blocos `/* ... */` para documentação mais extensa.
- **Documentação de API (JSDoc/TSDoc):**
    - Todas as funções, classes, métodos e tipos exportados (APIs públicas dos módulos/pacotes) devem ser documentados usando a sintaxe JSDoc/TSDoc.
    - Descreva o propósito, os parâmetros (`@param`), o valor de retorno (`@returns`), e quaisquer exceções (`@throws`).
    - Exemplo:

```
/**
 * Envia uma mensagem para um peer específico na rede P2P.
 * @param peerId O ID do peer destinatário.
 * @param data Os dados a serem enviados (devem ser serializáveis).
 * @returns Uma Promise que resolve quando a mensagem é enviada ou
rejeita em caso de erro.
 * @throws {Error} Se o peerId for inválido ou a conexão falhar.
 */
async function enviarMensagem(peerId: string, data: any):
Promise<void> {
  // ... implementação ...
}
```

- Esta documentação será útil para a equipe e também pode ser usada para gerar documentação formal do projeto.

5. Fluxo de Trabalho de Desenvolvimento (Git e CI/CD)

Um fluxo de trabalho bem definido é essencial para a colaboração eficiente, a qualidade do código e a entrega contínua de valor. Este projeto adotará as seguintes práticas para o gerenciamento de código-fonte com Git e para a automação de processos com CI/CD.

5.1. Estratégia de Ramificação (Branching Strategy)

Utilizaremos um modelo de ramificação baseado no GitFlow, mas simplificado para agilidade:

- `main` **(ou** `master`**):**
  - Esta branch representa o código em produção, estável e testado.
  - Ninguém faz commit diretamente na `main`.
  - Merges para `main` ocorrem apenas a partir de branches `release` ou `hotfix` aprovadas.
  - Cada commit na `main` deve ser taggeado com um número de versão (ex: `v1.0.0`).
- `develop`**:**
  - Esta é a branch principal de desenvolvimento, contendo as funcionalidades mais recentes e integradas.
  - Serve como base para a criação de branches de `feature`.
  - Integração Contínua (CI) roda nesta branch para garantir que as novas funcionalidades não quebrem o build ou os testes.
  - Representa o estado da próxima release planejada.
- `feature/<nome-da-feature>` **(ou** `feat/<nome-da-feature>`**):**
  - Criadas a partir da `develop` para desenvolver novas funcionalidades.
  - Ex: `feature/autenticacao-did`, `feat/player-webtorrent`.
  - Ao concluir, são mescladas de volta na `develop` através de um Pull Request (PR).
  - Devem ser mantidas curtas e focadas em uma única funcionalidade para facilitar a revisão e o merge.

- `fix/<nome-da-correcao>` **(ou** `bugfix/<nome-da-correcao>`**):**
    - Criadas a partir da `develop` para corrigir bugs não críticos encontrados na branch `develop`.
    - Ex: `fix/erro-login-addon`, `bugfix/ui-quebrada-mobile`.
    - Ao concluir, são mescladas de volta na `develop` através de um PR.
- `release/<versao>` **(Opcional, dependendo da frequência de releases):**
    - Criadas a partir da `develop` quando esta atinge um estado estável e pronto para uma nova release.
    - Ex: `release/v1.1.0`.
    - Nesta branch, apenas correções de bugs críticos para a release (`hotfix` para a release) e preparação final (ex: atualização de versão, documentação) são permitidas.
    - Após testes finais e aprovação, a branch `release` é mesclada na `main` (e taggeada) e também de volta na `develop` (para incorporar quaisquer correções feitas na release).
- `hotfix/<nome-do-hotfix>` **(ou** `hotfix/<versao-do-hotfix>`**):**
    - Criadas a partir da `main` para corrigir bugs críticos encontrados em produção.
    - Ex: `hotfix/v1.0.1-falha-seguranca`.
    - Após a correção e testes, são mescladas de volta na `main` (e taggeada com nova versão patch) e também na `develop` (ou na `release` ativa, se houver) para garantir que a correção seja incorporada no desenvolvimento futuro.

5.2. Ciclo de Vida de uma Feature/Bugfix

1. **Criação da Issue:** Antes de iniciar o desenvolvimento, idealmente, deve haver uma issue no sistema de rastreamento (ex: GitHub Issues, Jira) descrevendo a feature ou bug.
2. **Atribuição:** A issue é atribuída a um desenvolvedor (ou o desenvolvedor se auto-atribui).
3. **Criação da Branch:** O desenvolvedor cria uma nova branch a partir da `develop` seguindo a convenção de nomenclatura (ex: `git checkout -b feature/minha-nova-feature develop`).
4. **Desenvolvimento e Commits:**
    - O desenvolvedor implementa a funcionalidade ou correção na sua branch.
    - Faça commits atômicos e frequentes com mensagens claras e descritivas (ver convenção de commits abaixo).
    - Mantenha sua branch atualizada com a `develop` frequentemente (`git pull origin develop` ou `git rebase develop`) para evitar conflitos grandes no final.
5. **Testes Locais:** Execute os testes relevantes (unitários, integração) localmente para garantir que as mudanças não quebraram nada.
6. **Push da Branch:** Faça push da sua branch para o repositório remoto (`git push origin feature/minha-nova-feature`).
7. **Criação do Pull Request (PR):** Abra um Pull Request da sua branch para a `develop`.
8. **Revisão de Código:** Outros membros da equipe revisam o PR.
9. **Discussão e Ajustes:** O autor do PR discute o feedback e faz os ajustes necessários (novos commits na mesma branch).
10. **Aprovação e Merge:** Após aprovação e passagem dos checks de CI, o PR é mesclado na `develop` (preferencialmente usando "Squash and merge" ou "Rebase and merge" para manter o histórico da `develop` limpo, a ser definido pela equipe).

11. **Exclusão da Branch (Opcional):** A branch de feature/fix pode ser excluída após o merge.

5.3. Pull Requests (PRs)

- **Obrigatórios:** Todos os merges para `develop` e `main` devem passar por Pull Requests.
- **Descrição Clara:** O PR deve ter um título claro e uma descrição que explique:
  - O que a mudança faz (propósito).
  - Por que a mudança é necessária (link para a issue, contexto).
  - Como a mudança foi implementada (visão geral das alterações técnicas, se relevante).
  - Como testar a mudança.
  - Quaisquer notas ou preocupações para os revisores.
  - (Um template de PR pode ser criado no repositório para padronizar isso).
- **PRs Pequenos e Focados:** Facilita a revisão e reduz o risco.
- **Revisão Construtiva:** As revisões devem ser respeitosas e focadas em melhorar a qualidade do código. O autor do PR deve estar aberto a feedback.
- **Mínimo de Revisores:** Definir um número mínimo de aprovações antes do merge (ex: 1 ou 2, dependendo do tamanho da equipe e criticidade).
- **Checks de CI:** O PR só deve ser mesclado se todos os checks de CI (linting, testes, build) passarem.

5.4. Testes Automatizados

Testes automatizados são cruciais para garantir a qualidade e permitir refatorações seguras.

- **Tipos de Testes:**
  - **Testes Unitários:** Testam pequenas unidades de código (funções, métodos) isoladamente. Devem ser rápidos e cobrir a lógica principal dos módulos.
    - Ferramentas: Jest, Vitest.
    - Localização: No diretório `test/` ou próximo ao código que testam (ex: `arquivo.test.ts`).
  - **Testes de Integração:** Testam a interação entre múltiplos componentes ou módulos (ex: interação do Core com um addon, comunicação entre diferentes serviços do Core).
    - Ferramentas: Jest, Vitest, possivelmente com bibliotecas auxiliares para mockar dependências.
  - **Testes End-to-End (E2E):** Testam o fluxo completo da aplicação do ponto de vista do usuário (ex: login, criação de post, interação com UI).
    - Ferramentas: Playwright, Cypress (para `app-web`). Ferramentas específicas para React Native (ex: Detox, Appium) para `app-mobile`.
- **Execução:**
  - Os desenvolvedores devem rodar testes localmente antes de abrir PRs.
  - CI deve rodar todos os testes automaticamente para cada PR e para merges em `develop` e `main`.
- **Cobertura de Teste:** Esforçar-se para uma boa cobertura de teste, especialmente para a lógica crítica do `core-p2p-lib` e funcionalidades chave.
- **Como Escrever Bons Testes:** Testes devem ser legíveis, confiáveis (não flaky), e testar comportamentos, não detalhes de implementação.

5.5. Integração Contínua e Deploy Contínuo (CI/CD)

Um pipeline de CI/CD será configurado (ex: usando GitHub Actions, GitLab CI) para automatizar o processo de build, teste e deploy.

- **Pipeline de CI (para PRs e `develop`):**
    1. Checkout do código.
    2. Instalação de dependências (cache de dependências para acelerar).
    3. Linting e checagem de formatação.
    4. Execução de testes unitários e de integração.
    5. Build dos pacotes (verificando se o projeto compila corretamente).
    6. (Opcional) Análise estática de segurança.
    7. Notificação do status (sucesso/falha).
- **Pipeline de CD (para `main` ou `release`):**
    1. Todos os passos do pipeline de CI.
    2. Build dos artefatos de produção.
    3. (Opcional) Publicação de pacotes (ex: `core-p2p-lib`, `addon-sdk` para npm, se desejado).
    4. Deploy para um ambiente de Staging/Homologação para testes finais.
    5. (Opcional, com aprovação manual) Deploy para Produção.

Convenção de Commits (Ex: Conventional Commits):
Para mensagens de commit mais estruturadas e que podem ser usadas para gerar changelogs automaticamente, considera-se o uso do padrão Conventional Commits.
Exemplos:

- `feat: adiciona funcionalidade de login com DID`
- `fix: corrige erro de renderização no perfil do usuário`
- `docs: atualiza documentação da API de armazenamento`
- `style: formata código usando Prettier`
- `refactor: melhora performance do módulo de rede`
- `test: adiciona testes unitários para o IdentityManager`
- `chore: atualiza dependências de desenvolvimento`

6. Desenvolvimento do Core (`core-p2p-lib`)

O pacote `core-p2p-lib` é o coração da plataforma, contendo a lógica P2P fundamental, o gerenciamento de addons e a API principal consumida pelos addons nativos. O desenvolvimento neste pacote requer atenção especial à estabilidade, performance e segurança.

6.1. Módulos Principais

Conforme descrito no `projeto-rede-social-p2p.md` e referenciado anteriormente, os módulos principais do `core-p2p-lib` incluem (mas não se limitam a):

- **`IdentityManager`:**
    - Responsável pela criação, restauração, armazenamento seguro e gerenciamento da identidade descentralizada do usuário (baseada em pares de chaves, DIDs).
    - Fornece funcionalidades para assinar dados e verificar assinaturas.
- **`NetworkManager`:**
    - Gerencia as conexões P2P usando `js-libp2p`.

    - Lida com descoberta de peers (DHT, bootstrap), estabelecimento de conexões (WebRTC, relays), e comunicação direta entre peers.
    - Gerencia a participação em tópicos pub/sub para disseminação de informações.
- `StorageManager`:
    - Abstrai o armazenamento e a sincronização de dados P2P.
    - Integra com `IPFS` para armazenamento de arquivos e `OrbitDB` para bancos de dados P2P com CRDTs (perfis, posts, etc.).
    - Fornece uma interface para o Core e addons (via `CoreAPI`) para interagir com o armazenamento.
    - Pode incluir lógica para WebTorrent para streaming P2P.
- `AddonManager`:
    - Responsável pelo ciclo de vida completo dos addons nativos (carregamento, inicialização em sandbox, comunicação, descarregamento).
    - Gerencia o registro de addons, suas permissões e a `CoreAPI` exposta a eles.
    - Contém a lógica para atuar como cliente e adaptador para addons externos (ex: Stremio), traduzindo seus manifestos e protocolos.
- `EventBus`:
    - Um sistema de eventos interno que permite a comunicação desacoplada entre diferentes módulos do Core e entre o Core e os addons nativos.
    - Addons podem emitir e escutar eventos (conforme suas permissões) para interagir com a plataforma e outros addons.
- `PlayerManager` **(Potencial):**
    - Se o streaming de mídia for uma funcionalidade central, um módulo para gerenciar o player de vídeo/áudio integrado (com suporte WebTorrent) pode ser necessário.

Cada um desses módulos deve ser bem encapsulado, com interfaces claras e ser exaustivamente testado.

6.2. Entendendo a `CoreAPI` (para Addons Nativos)

A `CoreAPI` é a interface que o `core-p2p-lib` expõe para os addons nativos. É através dela que os addons interagem com a plataforma para:

- Acessar dados do usuário (com permissão).
- Ler e escrever em coleções de armazenamento P2P.
- Enviar e receber mensagens na rede.
- Registrar componentes de UI e rotas.
- Emitir e escutar eventos do `EventBus`.
- Acessar informações de identidade do usuário.

Princípios ao Designar e Evoluir a `CoreAPI`:

- **Mínima Exposição Necessária:** Exponha apenas o que é estritamente necessário para a funcionalidade do addon, seguindo o princípio de menor privilégio.
- **Abstração:** A `CoreAPI` deve abstrair os detalhes complexos da implementação P2P, fornecendo uma interface mais simples e estável para os desenvolvedores de addons.
- **Segurança:** Cada chamada da `CoreAPI` deve ser verificada contra as permissões declaradas pelo addon em seu manifesto.

- **Versionamento:** A `CoreAPI` deve ser versionada (ex: `coreApiVersion` no manifesto do addon). Mudanças que quebram a compatibilidade devem resultar em um incremento da versão maior da API.
- **Clareza e Documentação:** Todas as funções e namespaces da `CoreAPI` devem ser exaustivamente documentados (via TSDoc) para os desenvolvedores de addons.

6.3. Boas Práticas ao Modificar o Core

Dado o papel central do `core-p2p-lib`, qualquer modificação deve ser feita com cuidado:

- **Testes Rigorosos:**
  - Adicione testes unitários para toda nova lógica ou modificação.
  - Adicione ou atualize testes de integração para garantir que os módulos interajam corretamente e que a `CoreAPI` funcione como esperado.
  - Busque alta cobertura de testes neste pacote.
- **Performance:** Monitore o impacto de performance das suas mudanças. Operações P2P e criptográficas podem ser custosas.
  - Evite gargalos, otimize algoritmos e considere o uso de Web Workers para tarefas pesadas se elas puderem impactar a responsividade do cliente que consome o core.
- **Segurança:** Revise o código sob a perspectiva de segurança. Valide todas as entradas, sanitize saídas, e esteja ciente das implicações de segurança de interações P2P e gerenciamento de chaves.
- **Compatibilidade Retroativa:** Ao modificar a `CoreAPI` ou comportamentos que afetam addons:
  - Tente manter a compatibilidade retroativa o máximo possível.
  - Se uma mudança quebra a compatibilidade for inevitável, planeje um ciclo de depreciação claro:
    1. Introduza a nova API/comportamento.
    2. Marque a API/comportamento antigo como obsoleto (`@deprecated`), indicando a versão em que será removido e qual a alternativa.
    3. Comunique claramente aos desenvolvedores de addons.
    4. Remova a API/comportamento obsoleto apenas em uma futura versão maior da `CoreAPI`.
- **Modularidade e Coesão:** Mantenha os módulos do core coesos (com responsabilidades bem definidas) e com baixo acoplamento entre si.
- **Documentação:** Atualize toda a documentação relevante (TSDoc, `GUIA_DESENVOLVIMENTO.md`, `projeto-rede-social-p2p.md`) para refletir suas mudanças.
- **Revisão de Código Criteriosa:** PRs que modificam o `core-p2p-lib` devem passar por uma revisão de código particularmente rigorosa por membros experientes da equipe.

7. Desenvolvimento de Addons Nativos

Addons nativos são módulos ESModule construídos especificamente para esta plataforma. Eles rodam em um ambiente de sandbox dentro do cliente e interagem com o `core-p2p-lib` através da `CoreAPI` para estender as funcionalidades da rede social.

7.1. Utilizando o `addon-sdk`

O pacote `addon-sdk` é fornecido para simplificar e padronizar o desenvolvimento de addons nativos. Ele oferece:

- **Tipos e Interfaces:** Definições TypeScript para a `CoreAPI`, manifestos de addon, e outras estruturas de dados relevantes.
- **Funções de Conveniência:** Utilitários para tarefas comuns, como registro de componentes de UI, manipulação de eventos, ou interações simplificadas com a `CoreAPI`.
- **Templates (Potencial):** Exemplos de estruturas de addons para iniciar rapidamente novos projetos de addons.
- **Abstração:** Pode abstrair algumas mudanças menores na `CoreAPI` ao longo do tempo, oferecendo uma camada mais estável para os desenvolvedores de addons.

É altamente recomendável que todos os addons nativos utilizem o `addon-sdk` para garantir compatibilidade e seguir as melhores práticas.

```
// Exemplo hipotético de uso do SDK (a ser definido em addon-sdk)
import { defineAddon, UIElements, StorageUtils } from 'addon-sdk';

export default defineAddon({
  manifest: {
    id: 'meu.addon.exemplo',
    name: 'Meu Addon de Exemplo',
    version: '0.1.0',
    // ... outras propriedades do manifesto ...
    permissions: ['core:storage:readSelf', 'core:ui:showNotification']
  },

  async initialize(coreAPI) {
    console.log('Meu Addon de Exemplo está inicializando!');
    coreAPI.ui.showNotification('Meu Addon carregado!');

    // Exemplo de uso de um utilitário do SDK
    const userData = await StorageUtils.getScopedData(coreAPI, 'preferencias');
  },

  getUIComponents() {
    return {
      MeuComponenteDeConfig: () => <div>Configurações do Meu Addon</div>
    };
  }
  // ... outros métodos do ciclo de vida ...
});
```

7.2. Anatomia de um Addon Nativo

Um addon nativo é composto principalmente por:

1. **Manifesto (`manifest.json`):**

- Um arquivo JSON que descreve o addon: `id`, `name`, `version`, `description`, `author`, `coreApiVersion` (a versão da `CoreAPI` que ele espera), `entryModule` (o caminho para o seu bundle JavaScript, geralmente hospedado no IPFS), `permissions` (lista de permissões da `CoreAPI` que ele requer), e seções para `ui` (rotas, componentes) e `events` (eventos que provê ou consome).
- Exemplo de manifesto foi detalhado na Seção 6.2 do `projeto-rede-social-p2p.md`.

2. **Módulo de Entrada (Entry Module):**
   - Um arquivo JavaScript (ESModule) que serve como o ponto de entrada principal para o addon. Este é o arquivo especificado no `entryModule` do manifesto.
   - Este módulo deve exportar as funções do ciclo de vida do addon.

3. **Funções do Ciclo de Vida:**
   O módulo de entrada do addon deve exportar as seguintes funções (ou um objeto contendo-as, dependendo da convenção final do `addon-sdk`), que são chamadas pelo `AddonManager` do Core:
   - `async initialize(coreAPI: CoreAPI, addonContext: AddonContext): Promise<AddonInstance | void>`:
     - Chamada quando o addon é carregado pela primeira vez.
     - Recebe uma instância da `CoreAPI` (para interagir com a plataforma) e um `addonContext` (com informações sobre o addon, como seu ID, diretório de dados privado, etc.).
     - Deve realizar qualquer configuração inicial, registrar rotas, componentes de UI, e handlers de eventos.
     - Pode retornar um objeto `AddonInstance` que expõe métodos que outros addons ou o Core podem chamar (via `EventBus` ou RPC, a ser definido).
   - `async activate(): Promise<void>` (Opcional):
     - Chamada quando o addon é ativado (após `initialize` ou se estava desativado e é reativado).
     - Útil para iniciar processos em background ou restaurar estado volátil.
   - `async deactivate(): Promise<void>` (Opcional):
     - Chamada quando o addon é desativado (ex: pelo usuário, ou antes de ser descarregado).
     - Deve limpar timers, listeners de eventos externos, e liberar recursos que não precisam ser mantidos enquanto desativado.
   - `async terminate(): Promise<void>` (Opcional):
     - Chamada quando o addon está prestes a ser completamente descarregado.
     - Deve realizar qualquer limpeza final e persistir estado se necessário.
     - Após `terminate`, o addon não deve mais ser usado.

4. **Componentes de UI e Lógica:**
   - O addon conterá os componentes React (ou da framework de UI escolhida) para as suas interfaces, além da lógica de negócios específica.

7.3. Interagindo com a `CoreAPI`

A `coreAPI` passada para a função `initialize` é o principal meio de interação do addon com a plataforma. Exemplos:

```
// Dentro do addon, após receber coreAPI na inicialização

// Registrar uma rota na UI principal
coreAPI.ui.registerRoute('/meu-addon/pagina', MeuComponentePagina);

// Mostrar uma notificação
coreAPI.ui.showNotification('Ação concluída pelo Meu Addon!', { type: 'success'
});

// Obter o perfil do usuário atual
const perfil = await coreAPI.identity.getCurrentUserProfile();

// Ler/escrever dados no armazenamento P2P (requer permissão)
const meusDados = await coreAPI.storage.getCollection('meu-addon-data');
await meusDados.get('config').put({ tema: 'escuro' });

// Emitir um evento para outros addons ou o Core
await coreAPI.events.emit('meuAddon:eventoCustomizado', { detalhe: 'valor' });

// Escutar um evento da plataforma
coreAPI.events.on('core:profile:updated', (novoPerfil) => {
  console.log('Perfil do usuário atualizado no Meu Addon:', novoPerfil);
});
```

Consulte a documentação da `CoreAPI` (a ser criada) para a lista completa de funcionalidades e seus usos.

7.4. Gerenciamento de Estado em Addons

Addons frequentemente precisam armazenar seu próprio estado (configurações, dados específicos do addon).

- **Armazenamento Persistente:**
  - Use a `CoreAPI` (ex: `coreAPI.storage.getScopedCollection('nome-da-sua-colecao')`) para armazenar dados de forma persistente e sincronizada via P2P. O Core pode fornecer um namespace isolado para cada addon.
  - Ideal para configurações do addon, dados gerados pelo addon que precisam sobreviver entre sessões.
- **Estado Volátil na UI:**
  - Para estado de UI dentro dos componentes do addon, use os mecanismos padrão da framework de UI (ex: `useState`, `useReducer` no React, ou stores do Zustand escopadas para o addon).
- **Limpeza:** Use as funções `deactivate` ou `terminate` do ciclo de vida para limpar estado em memória que não precisa persistir.

7.5. Sandboxing, Permissões e Segurança

- **Sandbox:** Lembre-se que seu addon nativo rodará em um ambiente de sandbox (Web Worker ou Iframe) com acesso limitado ao DOM global e outras APIs do navegador, exceto o que é exposto pela `CoreAPI`.
- **Permissões:**
  - Seu addon **deve** declarar todas as permissões da `CoreAPI` que necessita em seu `manifest.json`.
  - Tentativas de usar partes da `CoreAPI` para as quais o addon não tem permissão resultarão em erro.
  - Solicite apenas as permissões estritamente necessárias para a funcionalidade do seu addon.
- **Validação de Dados:** Valide e sanitize quaisquer dados recebidos de fontes externas ou do usuário antes de usá-los ou armazená-los.
- **Não Confie Cegamente:** Mesmo que a `CoreAPI` forneça dados, seja defensivo na sua programação.

7.6. Empacotamento e Distribuição

1. **Build do Addon:**
   - O código TypeScript/JavaScript do seu addon precisará ser transpilado e empacotado (bundle) em um ou mais arquivos ESModule.
   - Ferramentas como `Rollup`, `esbuild`, ou `Webpack` podem ser usadas. O `addon-sdk` pode fornecer scripts ou configurações padrão para isso.
2. **Hospedagem no IPFS:**
   - O bundle resultante do addon deve ser adicionado ao IPFS. O CID (Content Identifier) resultante será usado no `entryModule` do manifesto do addon.
   - Isso garante que o código do addon seja distribuído de forma descentralizada.
3. **Publicação do Manifesto:**
   - O `manifest.json` do addon (contendo o CID do `entryModule`) deve ser publicado. Isso pode ser feito adicionando o manifesto a um banco de dados OrbitDB que serve como um catálogo de addons, ou publicando o manifesto no IPFS e seu CID (ou um ponteiro IPNS para ele) em uma lista conhecida pela comunidade ou pelo Core.

O `addon-sdk` e as ferramentas do projeto visarão simplificar ao máximo esse processo de build e publicação.

8. Integração com Addons Externos (Ex: Stremio)

Uma das estratégias para enriquecer rapidamente o catálogo de funcionalidades da plataforma, especialmente para mídia como filmes e séries, é integrar-se com ecossistemas de addons existentes, como o do Stremio. Esta seção descreve como essa integração funciona.

Importante: A plataforma não executa o código de addons externos diretamente. Em vez disso, ela atua como um cliente para os serviços que esses addons expõem (geralmente via HTTP).

8.1. Como o Core Descobre e Consome Addons Externos

O processo de descoberta e consumo de addons externos (focando no exemplo do Stremio) envolve:

1. **Descoberta de Manifestos de Addons Externos:**

- **Listas Curadas/Padrão:** O `core-p2p-lib` (ou um addon nativo de "catálogo Stremio") pode vir com uma lista pré-definida de URLs que apontam para manifestos de addons Stremio populares e confiáveis ou para listas de manifestos (ex: `https://stremio-addons.netlify.app/manifest.json`, `https://stremio-community-addons.netlify.app/manifest.json`).
- **Adição pelo Usuário:** A UI da plataforma pode permitir que os usuários adicionem URLs de manifestos de addons Stremio individuais que desejam usar.
- **Formato do Manifesto Stremio:** O Core precisa ser capaz de buscar e parsear o `manifest.json` padrão do Stremio, que descreve:
  - `id`, `version`, `name`, `description`
  - `resources`: Um array que define os tipos de conteúdo e funcionalidades que o addon oferece (ex: `catalog`, `stream`, `meta`, `subtitles`).
  - `types`: Array dos tipos de mídia suportados (ex: `movie`, `series`, `channel`, `tv`).
  - `idPrefixes`: Para identificar os IDs de itens que este addon pode fornecer streams (ex: `tt` para IMDb ID, `kitsu` para Kitsu ID).
  - `behaviorHints`: Dicas sobre o comportamento do addon (ex: se requer configuração).

2. **Interação com Endpoints do Addon Externo:**

- Uma vez que um manifesto de addon Stremio é carregado e o usuário deseja interagir com ele (ex: navegar em um catálogo de filmes ou buscar streams para um item específico), o `core-p2p-lib` (ou o addon ponte) faz requisições HTTP `GET` para os endpoints definidos implicitamente pela combinação de `resources`, `types` e IDs.
- **Exemplo de Catálogo:** Para buscar filmes populares de um addon Stremio cujo servidor está em `https://meu-addon-stremio.exemplo.com` e que declara um recurso `catalog` para o tipo `movie`:

```
GET https://meu-addon-stremio.exemplo.com/catalog/movie/popular.json
GET https://meu-addon-
stremio.exemplo.com/catalog/movie/popular/skip=20.json (para
paginação)
```

- **Exemplo de Streams:** Para buscar streams para um filme com IMDb ID `tt0120338`:

```
GET https://meu-addon-
stremio.exemplo.com/stream/movie/tt0120338.json
```

- As respostas dessas requisições são JSON (ex: `{ metas: [...] }` para catálogos, `{ streams: [...] }` para streams).

8.2. O Papel dos "Addons Ponte" ou Adaptadores

Embora o `core-p2p-lib` possa conter a lógica básica para consumir addons Stremio, uma abordagem mais modular poderia ser ter um **addon nativo especializado** atuando como uma "Ponte Stremio" (Stremio Bridge Addon).

- **Responsabilidades de um Addon Ponte Stremio:**
  - Gerenciar a lista de URLs de manifestos de addons Stremio (permitindo que o usuário adicione/remova).
  - Buscar e parsear os manifestos Stremio.
  - Expor uma interface para o `core-p2p-lib` ou para a UI da plataforma para listar os addons Stremio disponíveis e seus catálogos.
  - Quando uma solicitação é feita (ex: buscar filmes), este addon ponte faz as chamadas HTTP para o addon Stremio correspondente.
  - Recebe a resposta JSON do addon Stremio e a **transforma/adapta** para um formato que a UI da plataforma possa entender e exibir consistentemente.
  - Encaminha os links de stream (ex: magnéticos) para o `core-p2p-lib` para que ele possa iniciar a reprodução via WebTorrent.
- **Vantagens desta Abordagem:**
  - **Modularidade:** A lógica específica do Stremio fica contida em um addon, mantendo o Core mais limpo.
  - **Extensibilidade:** No futuro, poderiam ser criados outros addons ponte para diferentes ecossistemas de addons externos, se existirem.

8.3. Tratamento de Dados e UI para Conteúdo Externo

- **Adaptação de Metadados:** Os metadados recebidos de addons Stremio (para filmes, séries, etc.) precisam ser mapeados para os modelos de dados internos da UI da plataforma para exibição consistente (ex: pôsteres, títulos, descrições, informações de elenco).
- **Apresentação na UI:**
  - A UI deve indicar claramente quando o conteúdo está sendo fornecido por um addon externo.
  - Os catálogos e resultados de busca de addons externos devem ser integrados de forma fluida na experiência do usuário.
- **Gerenciamento da Reprodução (pelo Core):**
  - Quando um addon Stremio fornece um link de stream (ex: um `magnet:?xt=urn:btih:...` para torrents, ou um link HTTP para um arquivo `.mp4`), o `core-p2p-lib` assume a responsabilidade pela reprodução:
    - **Para Torrents (WebTorrent):**
      1. O Core utiliza sua instância `WebTorrent`.
      2. Inicia o download e streaming P2P do conteúdo do torrent.
      3. Fornece o stream de vídeo/áudio para o player de mídia integrado na aplicação (web ou mobile).
      4. A UI exibe informações de progresso do download/streaming, número de peers, etc.
    - **Para Links HTTP Diretos:** O player de mídia integrado pode reproduzir diretamente esses links, embora o foco P2P seja preferível.
- **Segurança e Confiança:**
  - O usuário deve ser informado sobre a origem dos addons externos.
  - O Core não executa código arbitrário desses addons, apenas consome seus dados via HTTP. O principal risco é um addon fornecer links maliciosos, o que é uma preocupação geral da web. A curadoria de listas de addons ou feedback comunitário pode ajudar a mitigar isso.

Ao implementar essa integração, é crucial consultar a documentação oficial do Stremio sobre desenvolvimento de addons (ou recursos similares) para entender completamente o protocolo e os formatos de dados esperados.

9. Desenvolvimento Frontend (`app-web`, `app-mobile`)

Os pacotes `app-web` e `app-mobile` são responsáveis por fornecer a interface do usuário (UI) e a experiência do usuário (UX) da plataforma. Eles são construídos sobre o `core-p2p-lib` e interagem com ele para exibir dados, funcionalidades P2P e conteúdo de addons.

9.1. Arquitetura Frontend

- **Framework Principal:** `React` é a biblioteca base para a construção da UI.
- **Cross-Platform com Expo:** Utilizamos `Expo` para o desenvolvimento e build das aplicações web (`@expo/webpack-config`) e mobile (`React Native`). Isso permite um alto grau de compartilhamento de código.
    - Site Expo: https://expo.dev/
    - Documentação React Native: https://reactnative.dev/docs/getting-started
- **Navegação:** `React Navigation` (https://reactnavigation.org/) será usado para gerenciar a navegação entre telas e fluxos em ambas as plataformas (web e mobile), configurando navegadores de pilha (stack), abas (tabs), e gavetas (drawer) conforme necessário.
- **Estrutura de Diretórios (Feature-First/Atomic):**
    - Dentro de `app-web/src` e `app-mobile/src`, organizar o código preferencialmente por `features` (funcionalidades) ou `screens` (telas).
    - Componentes de UI reutilizáveis específicos da aplicação (que não pertencem ao `design-system` global) podem seguir uma abordagem de `Atomic Design` (atoms, molecules, organisms, templates, pages/screens) ou serem agrupados por funcionalidade.
    - Exemplo de estrutura:

```
app-web/src/
├── components/      # Componentes de UI específicos da app-web
├── features/        # Módulos de funcionalidades (ex: auth, feed,
profile)
│   ├── auth/
│   │   ├── components/
│   │   ├── hooks/
│   │   └── screens/
│   └── ...
├── navigation/      # Configuração do React Navigation
├── services/        # Lógica para interagir com o core (se não
hooks)
├── screens/         # Telas principais (se não dentro de features)
├── App.tsx          # Ponto de entrada principal
└── ...
```

9.2. Componentização e Design System

- `packages/design-system`: Este é o pacote central para todos os componentes de UI reutilizáveis e estilos visuais da plataforma.
    - **Tecnologia:** `Tamagui` (https://tamagui.dev/) ou `NativeWind` (https://www.nativewind.dev/) para criar componentes verdadeiramente cross-platform (web e mobile) a partir de uma única base de código, com inspiração em `shadcn/ui` para a API e estética.
    - **Princípios:** Seguir os princípios de Atomic Design. Construir desde `átomos` (Botão, Input, Texto) até `organismos` (CardDePost, PlayerDeVideo) reutilizáveis.
    - **Tematização:** O Design System deve suportar tematização (claro/escuro, cores customizáveis) para permitir personalização.
- **Uso nas Aplicações:** `app-web` e `app-mobile` devem importar e utilizar componentes diretamente do `design-system` sempre que possível para garantir consistência visual e reduzir duplicação.
- **Contribuições ao Design System:** Novas necessidades de componentes genéricos devem, preferencialmente, ser adicionadas ao `design-system`.

9.3. Gerenciamento de Estado

- **Estado Global:** Para o estado que precisa ser compartilhado entre diferentes partes da aplicação (ex: informações do usuário logado, status da conexão P2P, configurações globais), utilizaremos `Zustand` (https://github.com/pmndrs/zustand).
    - **Vantagens:** Simples, leve, flexível, e não requer provedores React no topo da árvore de componentes.
    - Criar stores separadas por domínio (ex: `authStore`, `profileStore`, `settingsStore`).
    - Usar persistência do Zustand (ex: com `persist` middleware) para dados que precisam sobreviver entre sessões (ex: token de sessão, preferências do usuário).
- **Estado Local de Componentes:** Para estado que é relevante apenas para um componente ou uma pequena árvore de componentes, usar os hooks padrão do React (`useState`, `useReducer`).
- **Estado de Servidor / Cache de Dados P2P:**
    - Para dados buscados do `core-p2p-lib` (que por sua vez vêm da rede P2P ou do armazenamento local), considerar o uso de bibliotecas como `TanStack Query (React Query)` (https://tanstack.com/query/latest) para gerenciar cache, sincronização em background, e estado de requisições.
    - Isso pode simplificar o tratamento de dados assíncronos, estados de carregamento/erro, e otimizações como re-fetching automático.

9.4. Interação com o Core e Addons

- **Consumindo o `core-p2p-lib`:**
    - As aplicações frontend (web e mobile) instanciarão e interagirão com o `core-p2p-lib`.
    - Idealmente, criar uma camada de abstração (ex: custom hooks ou services) no frontend para encapsular as chamadas ao `core-p2p-lib` e gerenciar o estado resultante.

```
// Exemplo de custom hook
// app-web/src/features/profile/hooks/useUserProfile.ts
import { core } from '../../../services/coreInstance'; // Instância
do core
import { useQuery } from '@tanstack/react-query';
```

```
export function useUserProfile(userId: string) {
  return useQuery(['userProfile', userId], () =>
core.identity.getUserProfile(userId));
}
```

- **Renderizando UI de Addons Nativos:**
  - Addons nativos podem registrar rotas e componentes de UI através da `CoreAPI`.
  - O frontend precisa ter um mecanismo para renderizar dinamicamente esses componentes/rotas registrados pelos addons.
  - Isso pode envolver um componente "placeholder" que o `AddonManager` preenche, ou um sistema de roteamento que consulta os addons ativos.
- **Exibindo Conteúdo de Addons Externos (Stremio):**
  - O frontend receberá os metadados (ex: lista de filmes) do `core-p2p-lib` (que os obteve do addon externo).
  - Renderizará esses metadados usando os componentes do `design-system`.
  - Quando o usuário selecionar um item para reprodução, o frontend informará o Core, que lidará com o início do streaming (ex: WebTorrent). O frontend então usará o player de mídia integrado para exibir o stream fornecido pelo Core.

9.5. Otimização de Performance e Boas Práticas de UI/UX

- **Carregamento Progressivo (Lazy Loading):**
  - Usar `React.lazy` e `Suspense` para carregar componentes e telas sob demanda, especialmente para rotas menos acessadas.
  - Para listas de dados (feeds, catálogos), carregar apenas um conjunto inicial e implementar paginação ou scroll infinito.
- **Virtualização de Listas:** Para listas longas e feeds, usar bibliotecas de virtualização (ex: `FlashList` da Shopify para React Native/Expo, `react-window` ou `react-virtualized` para web) para renderizar apenas os itens visíveis na tela, melhorando drasticamente a performance.
- **Otimização de Imagens:** Usar formatos de imagem otimizados (ex: WebP), compressão, e carregamento responsivo de imagens (diferentes tamanhos para diferentes telas).
- **Memoization:** Usar `React.memo` para componentes funcionais, e `useMemo`/`useCallback` para evitar recálculos e re-renderizações desnecessárias.
- **Debouncing e Throttling:** Para eventos frequentes (ex: input de busca, eventos de scroll/resize), usar debounce ou throttle para limitar a frequência das atualizações ou chamadas de API.
- **Tratamento de Erros e Estados de Carregamento:** Fornecer feedback claro ao usuário para operações assíncronas (indicadores de carregamento) e tratar erros graciosamente (mensagens de erro amigáveis, opções de retry).
- **Acessibilidade (a11y):**
  - Seguir as diretrizes do WCAG.
  - Usar HTML semântico, atributos ARIA apropriados, garantir contraste de cores, e testar a navegação por teclado e com leitores de tela.
  - Expo e React Native oferecem APIs para melhorar a acessibilidade.
- **Design Responsivo:** A UI deve se adaptar fluidamente a diferentes tamanhos de tela e orientações.
- **Testes de UI:** Usar React Testing Library para testes de componentes e Playwright/Cypress/Detox para testes E2E para validar o comportamento da UI.

10. Backend Descentralizado (Detalhes da Implementação P2P)

Esta seção foca nos detalhes de implementação das tecnologias P2P que formam o backend descentralizado da plataforma, principalmente dentro do `core-p2p-lib`.

10.1. js-libp2p

`js-libp2p` (https://libp2p.io/) é o framework fundamental para a camada de rede P2P. Ele oferece um conjunto modular de protocolos para descoberta, roteamento, transporte e comunicação segura entre peers.

- **Inicialização e Configuração:**
  - Uma instância do `Libp2p` será criada no `NetworkManager` dentro do `core-p2p-lib`.
  - A configuração incluirá:
    - **PeerId:** Gerado a partir das chaves de identidade do usuário (via `IdentityManager`) para garantir um ID de nó persistente e criptograficamente verificável.
    - **Transports:**
      - `WebSockets` (com `Noise` para criptografia e `Mplex` ou `Yamux` para multiplexação de streams) para conexões com nós de bootstrap e relays.
      - `WebRTC` (ou `WebRTCStar` para signaling via um servidor Star) para conexões diretas browser-to-browser, otimizando a latência e a transferência de dados.
    - **Stream Muxers:** `Mplex` ou `Yamux` para permitir múltiplas streams lógicas sobre uma única conexão de transporte.
    - **Connection Encryption:** `Noise` (https://noiseprotocol.org/) para garantir a privacidade e autenticidade das conexões entre peers.
    - **Peer Discovery:**
      - `Bootstrap`: Uma lista de endereços de nós de bootstrap confiáveis para conectar-se inicialmente à rede.
      - `DHT (Kademlia)`: Para descoberta descentralizada de outros peers e conteúdo. O `js-libp2p-kad-dht` será configurado para operar no modo cliente ou servidor, conforme apropriado.
      - `MulticastDNS (mDNS)`: Para descoberta de peers na rede local (útil para desenvolvimento e cenários offline locais).
      - (Opcional) `PubSub Peer Discovery`: Descobrir peers inscritos nos mesmos tópicos PubSub.
    - **PubSub:**
      - `GossipSub` (ou `FloodSub` para redes menores/iniciais) será usado para disseminação de mensagens em tempo real em tópicos específicos (ex: atualizações de feed, eventos de addons).
- **Uso no `NetworkManager`:**
  - Gerenciar o ciclo de vida do nó libp2p (`start`, `stop`).
  - Registrar handlers para eventos de conexão/desconexão de peers.
  - Expor funcionalidades para abrir streams diretas para outros peers, publicar e subscrever em tópicos PubSub.
  - Utilizar a DHT para encontrar peers ou anunciar a própria presença/serviços.

10.2. WebTorrent

`WebTorrent` (https://webtorrent.io/) será usado para o streaming P2P eficiente de arquivos de mídia (vídeos, áudios), especialmente para conteúdo descoberto via addons Stremio ou compartilhado diretamente pelos usuários.

- **Integração:**
  - Uma instância do cliente WebTorrent será gerenciada pelo `core-p2p-lib` (provavelmente dentro do `StorageManager` ou um `PlayerManager` dedicado).
  - Quando um link magnético (`magnet:`) ou um arquivo `.torrent` é fornecido (ex: por um addon Stremio ou um addon nativo de compartilhamento), o cliente WebTorrent inicia o processo.
- **Funcionalidades Chave:**
  - **Streaming:** WebTorrent permite que a reprodução comece quase imediatamente, enquanto o restante do arquivo é baixado em segundo plano e compartilhado com outros peers interessados.
  - **Descoberta de Peers (via Trackers e DHT):** Usa trackers WebTorrent (HTTP/UDP/WS) e a Mainline DHT para encontrar peers que possuem o conteúdo.
  - **Comunicação WebRTC:** Prioriza conexões WebRTC para transferência de dados diretamente entre navegadores/peers.
- **Interação com o Player de Mídia:**
  - O stream de dados do WebTorrent será conectado ao player de vídeo/áudio no frontend (`app-web`/`app-mobile`).
  - Bibliotecas como `Video.js` ou `Plyr` podem ser configuradas para aceitar um stream de WebTorrent.
  - A UI deve exibir informações sobre o progresso do download, velocidade, número de peers, etc.
- **Seeding:** Após o download (ou mesmo durante), o cliente WebTorrent local atuará como um seeder, ajudando a distribuir o conteúdo para outros peers.

10.3. IPFS (InterPlanetary File System)

`js-ipfs` (https://docs.ipfs.tech/reference/js/) será usado para armazenamento e endereçamento de conteúdo de forma descentralizada e resiliente.

- **Nó IPFS:**
  - O `core-p2p-lib` rodará uma instância de `js-ipfs` (ou `helia` - uma implementação mais leve e modular do IPFS).
  - Este nó se conectará à rede IPFS pública (ou a uma rede privada/comunitária, se definido).
- **Casos de Uso:**
  - **Armazenamento de Arquivos de Addons Nativos:** Os pacotes de código (bundles JS) dos addons nativos serão adicionados ao IPFS, e seus CIDs (Content Identifiers) serão referenciados nos manifestos dos addons.
  - **Mídia Compartilhada por Usuários:** Arquivos de imagem, áudio, vídeo ou documentos enviados por usuários através de addons nativos podem ser armazenados no IPFS.
  - **Metadados e Configurações (Opcional):** Pequenos arquivos de configuração ou metadados podem ser armazenados como objetos JSON no IPFS.
  - **Avatares de Perfil e Imagens de Posts:** Imagens podem ser armazenadas no IPFS.

- **Gerenciamento de CIDs:** O `StorageManager` será responsável por adicionar arquivos ao IPFS, obter seus CIDs, e recuperar arquivos a partir de CIDs.
- **Pinning:** Para garantir a persistência de dados importantes (ex: addons populares, conteúdo relevante), o nó IPFS local pode "fixar" (pin) CIDs. A integração com serviços de pinning externos também pode ser considerada para maior redundância.
- **Gateways IPFS:** Para facilitar o acesso a conteúdo IPFS a partir de navegadores comuns ou para embeds, gateways IPFS públicos (ex: `ipfs.io`) ou um gateway customizado podem ser usados para resolver CIDs via HTTP.

10.4. OrbitDB

`OrbitDB` (https://orbitdb.org/) é um banco de dados P2P serverless construído sobre IPFS. Ele permite criar bancos de dados dinâmicos que são automaticamente sincronizados entre peers e usam CRDTs (Conflict-free Replicated Data Types) para consistência eventual e resolução de conflitos sem necessidade de um servidor central.

- **Integração com IPFS:** OrbitDB usa uma instância IPFS para armazenar os dados do banco de dados e para comunicação P2P (via libp2p, que o IPFS usa internamente).
- **Tipos de Stores:** OrbitDB oferece diferentes tipos de stores para diferentes necessidades:
  - `feed` **(Log Append-Only):** Ideal para feeds de atividades, posts, comentários, onde novas entradas são adicionadas sequencialmente. Cada entrada é assinada pelo autor.
  - `docstore` **(Document Store):** Para armazenar documentos JSON, onde cada documento tem um ID único. Bom para perfis de usuário, configurações de addons, ou qualquer objeto que precise ser atualizado.
  - `keyvalue` **(Key-Value Store):** Um simples armazenamento de chave-valor.
  - `counter` **(CRDT Counter):** Para contadores distribuídos (ex: likes, visualizações), embora possa ser mais simples implementar likes como entradas em um `feed` ou um campo em um `docstore`.
- **Uso no `StorageManager`:**
  - O `StorageManager` criará e gerenciará instâncias de bancos de dados OrbitDB para diferentes tipos de dados da plataforma (ex: `user_profiles_db`, `social_feed_db`, `addon_catalog_db`).
  - As permissões de escrita para esses bancos de dados serão controladas (ex: apenas o proprietário do perfil pode escrever em seu documento de perfil, qualquer um pode adicionar a um feed público, mas as entradas são assinadas).
- **CRDTs e Resolução de Conflitos:** OrbitDB lida com a mesclagem de dados de diferentes peers usando CRDTs, garantindo que todos os peers eventualmente cheguem a um estado consistente. A lógica específica de resolução de conflitos pode depender do tipo de store (ex: "última escrita vence" com timestamps para `docstore`).
- **Sincronização e Replicação:**
  - OrbitDB sincroniza automaticamente os bancos de dados com outros peers que estão replicando o mesmo banco de dados.
  - A replicação pode ser seletiva (peers replicam apenas os bancos de dados de seu interesse).
  - O `core-p2p-lib` pode incentivar a replicação de dados de amigos/seguidores ou conteúdo popular.

Ao combinar essas tecnologias, o objetivo é criar um backend robusto, descentralizado e resiliente para a rede social.

11. Segurança para Desenvolvedores

- 11.1. Práticas de Codificação Segura
- 11.2. Gerenciamento de Segredos e Chaves
- 11.3. Lidando com Dados do Usuário

12. Debugging e Solução de Problemas

Saber como depurar eficientemente é crucial para um ciclo de desenvolvimento rápido. Esta seção cobre ferramentas e técnicas para diagnosticar e resolver problemas na plataforma.

12.1. Ferramentas de Debugging

- **Browser Developer Tools (para `app-web` e addons em Iframes):**
  - **Console:** Para visualizar logs (`console.log`, `console.error`, etc.), executar comandos JavaScript e inspecionar o estado.
  - **Debugger (Sources Panel):** Para definir breakpoints no código JavaScript/TypeScript, inspecionar variáveis, e percorrer a execução do código passo a passo.
  - **Network Panel:** Para inspecionar requisições HTTP (ex: feitas para addons Stremio), WebSockets e outras atividades de rede.
  - **Application Panel:** Para inspecionar armazenamento local (LocalStorage, IndexedDB), Service Workers, e cache.
- **React Developer Tools (Extensão de Navegador):**
  - Permite inspecionar a árvore de componentes React, seus props, state e hooks.
  - Ajuda a entender como os componentes estão sendo renderizados e atualizados.
  - Disponível para Chrome, Firefox e Edge.
- **Zustand Devtools (Middleware):**
  - Se estiver usando Zustand para gerenciamento de estado global, integre o middleware `devtools` para inspecionar o estado das suas stores e as ações despachadas, similar ao Redux DevTools.
  - Exemplo de configuração:

```
import { create } from 'zustand';
import { devtools } from 'zustand/middleware';

const useMyStore = create(devtools((set) => ({
  count: 0,
  inc: () => set((state) => ({ count: state.count + 1 })),
})));
```

- **Debugging de React Native (`app-mobile`):**
  - **Flipper (https://fbflipper.com/):** Plataforma de debugging extensível para iOS, Android e React Native. Permite inspecionar logs, rede, layout, e usar plugins específicos.

    - **Debugger do React Native (via Chrome DevTools ou standalone):** Permite usar o debugger do Chrome para código React Native.
    - **Expo Go App:** Facilita o debugging durante o desenvolvimento com Expo, mostrando logs e erros diretamente no dispositivo ou emulador.
- **Debugging do `core-p2p-lib` (e Lógica Node.js para Testes):**
    - Se estiver executando partes do core ou testes em um ambiente Node.js, use o debugger embutido do Node.js ou o debugger do seu IDE (VS Code, WebStorm).
    - Coloque `debugger;` no seu código para criar um breakpoint programático.
    - Inicie o Node.js com a flag `--inspect` ou `--inspect-brk` e conecte o debugger do Chrome (via `chrome://inspect`) ou do seu IDE.

12.2. Logging

Logs eficazes são indispensáveis para entender o comportamento da aplicação e diagnosticar problemas, especialmente em sistemas distribuídos como P2P.

- **Níveis de Log:** Adote níveis de log consistentes:
    - `DEBUG`: Informações detalhadas úteis apenas para desenvolvimento e debugging profundo (ex: dumps de objetos, traços de execução detalhados).
    - `INFO`: Informações gerais sobre o progresso da aplicação e eventos importantes (ex: inicialização de módulo, conexão P2P estabelecida, addon carregado).
    - `WARN`: Indica uma situação potencialmente problemática que não impede o funcionamento normal, mas que deve ser investigada (ex: falha ao conectar a um peer opcional, timeout em uma operação não crítica).
    - `ERROR`: Um erro que impediu uma operação de ser concluída ou que colocou a aplicação em um estado instável. Deve incluir contexto suficiente para diagnóstico (ex: stack trace, parâmetros da operação).
- **Formato dos Logs:**
    - Inclua timestamps.
    - Indique a origem do log (módulo, função, nome do addon).
    - Seja consistente no formato para facilitar a filtragem e análise.
    - Exemplo: `[2023-10-27T10:30:00Z] [INFO] [NetworkManager] Peer connected: 12D3KooW...`
- **No Frontend (`app-web`, `app-mobile`):**
    - Use `console.debug()`, `console.info()`, `console.warn()`, `console.error()`.
    - Considere o uso de uma biblioteca de logging mais robusta para o frontend (ex: `loglevel`, `pino-pretty` para formatação) se precisar de mais controle sobre níveis e saídas, especialmente em produção (embora em P2P, os logs de produção são mais complexos de coletar centralmente).
- **No `core-p2p-lib`:**
    - Use `console` ou uma biblioteca de logging (ex: `debug` para logs condicionais baseados em variáveis de ambiente, ou `pino` para logging estruturado e performático).
    - Para `js-libp2p`, `js-ipfs`, `OrbitDB`, eles geralmente usam a biblioteca `debug`. Você pode habilitar seus logs definindo a variável de ambiente `DEBUG` (ex: `DEBUG=libp2p*`, `DEBUG=ipfs*`, `DEBUG=orbitdb*`, ou `DEBUG=*` para todos).
- **Visualização de Logs:**

- No navegador: Aba Console do DevTools.
- Em React Native: Expo Go, Flipper, ou o terminal onde o Metro bundler está rodando.
- Em Node.js: O terminal onde o processo está rodando.

12.3. Problemas Comuns e Soluções (WIP - A ser construído pela equipe)

Esta seção será um repositório vivo de problemas comuns encontrados durante o desenvolvimento e suas soluções ou dicas de debugging. Contribuições da equipe são bem-vindas.

- **Problemas de Conexão P2P (libp2p, WebRTC):**
  - **Sintoma:** Peers não se descobrem, conexões diretas falham.
  - **Dicas:**
    - Verifique a configuração dos nós de bootstrap.
    - Certifique-se de que os servidores STUN/TURN estão configurados e acessíveis (se estiver usando WebRTC e atrás de NATs complexos).
    - Use `DEBUG=libp2p*` para ver logs detalhados do libp2p.
    - Verifique firewalls e configurações de rede.
    - Teste com peers na mesma rede local primeiro (mDNS pode ajudar).
- **Problemas de Sincronização (OrbitDB, IPFS):**
  - **Sintoma:** Dados não são sincronizados entre peers, ou há conflitos inesperados.
  - **Dicas:**
    - Verifique se os peers estão conectados e replicando os mesmos bancos de dados OrbitDB (mesmo endereço/nome).
    - Use `DEBUG=ipfs*` e `DEBUG=orbitdb*` para logs.
    - Inspecione o log de oplog do OrbitDB para entender as operações e merges.
    - Certifique-se de que as permissões de escrita nos bancos de dados OrbitDB estão corretas.
- **Addons Nativos Não Carregam ou Falham:**
  - **Sintoma:** Addon não aparece na lista, ou erros no console ao tentar carregar/usar.
  - **Dicas:**
    - Verifique o `manifest.json` do addon por erros de sintaxe ou caminhos incorretos no `entryModule`.
    - Verifique se o bundle do addon está acessível (ex: se hospedado no IPFS, o CID está correto e o conteúdo está disponível).
    - Verifique os logs do `AddonManager` no Core para erros durante o carregamento ou sandboxing.
    - Verifique se o addon está solicitando as permissões corretas na `CoreAPI`.
- **Problemas de Performance:**
  - **Sintoma:** UI lenta, alto uso de CPU/memória.
  - **Dicas:**
    - Use o Profiler do React DevTools para identificar gargalos de renderização.
    - Use o Performance Profiler do navegador para analisar o uso de CPU.
    - Otimize listas longas com virtualização.
    - Verifique se há operações P2P ou criptográficas excessivas na thread principal.

(A equipe deve adicionar mais itens a esta lista à medida que encontra e resolve problemas.)

13. Recursos Adicionais

Esta seção fornece links para documentação externa, canais de comunicação da equipe e outros recursos úteis para o desenvolvimento.

13.1. Links para Documentação de Tecnologias e Bibliotecas

- **React:**
  - Documentação Oficial: https://react.dev/
- **React Native:**
  - Documentação Oficial: https://reactnative.dev/
- **Expo:**
  - Documentação Oficial: https://docs.expo.dev/
- **TypeScript:**
  - Documentação Oficial: https://www.typescriptlang.org/docs/
- **js-libp2p:**
  - Site Principal: https://libp2p.io/
  - Documentação: https://docs.libp2p.io/
  - Implementação JavaScript: https://github.com/libp2p/js-libp2p
- **IPFS / js-ipfs / Helia:**
  - Site Principal IPFS: https://ipfs.tech/
  - Documentação IPFS: https://docs.ipfs.tech/
  - js-ipfs (Legado): https://github.com/ipfs/js-ipfs
  - Helia (Nova geração de IPFS em JS): https://github.com/ipfs/helia
- **OrbitDB:**
  - Site Principal: https://orbitdb.org/
  - GitHub (Documentação na API.md): https://github.com/orbitdb/orbit-db
- **WebTorrent:**
  - Site Principal: https://webtorrent.io/
  - GitHub: https://github.com/webtorrent/webtorrent
- **Tamagui (Design System):**
  - Site Principal: https://tamagui.dev/
- **NativeWind (Alternativa Design System):**
  - Site Principal: https://www.nativewind.dev/
- **Zustand (Gerenciamento de Estado):**
  - GitHub: https://github.com/pmndrs/zustand
- **React Navigation:**
  - Site Principal: https://reactnavigation.org/
- **TanStack Query (React Query):**
  - Site Principal: https://tanstack.com/query/latest
- **npm (Gerenciador de Pacotes):**
  - Site Principal: https://www.npmjs.com/
- **ESLint:**
  - Site Principal: https://eslint.org/
- **Prettier:**
  - Site Principal: https://prettier.io/

- **Jest (Testes):**
  - Site Principal: https://jestjs.io/
- **Playwright (Testes E2E):**
  - Site Principal: https://playwright.dev/
- **Stremio Addon SDK (para referência de addons externos):**
  - Documentação API: https://github.com/Stremio/stremio-addon-sdk/blob/master/docs/api.md

13.2. Canais de Comunicação da Equipe

- **Gerenciamento de Tarefas e Issues:** [Link para o GitHub Issues, Jira, Trello, ou similar]
- **Chat da Equipe:** [Link para o Slack, Discord, Microsoft Teams, ou similar]
- **Reuniões Semanais:** [Detalhes sobre o dia/hora e link para a vídeo chamada]
- **Repositório Principal do Código:** [Link para o repositório no GitHub/GitLab - a ser definido]
- **Documentação do Projeto (Orientação Geral):** `projeto-rede-social-p2p.md` (neste mesmo repositório)

13.3. Contatos Chave no Projeto

- **Líder Técnico / Arquiteto Principal:** [Nome e Contato]
- **Referência Frontend (Web/Mobile):** [Nome e Contato]
- **Referência Backend/P2P (Core):** [Nome e Contato]
- **Referência UX/UI Design:** [Nome e Contato]

(Esta seção deve ser atualizada com os links e nomes corretos à medida que o projeto avança e a equipe é formada.)

--- Fim do Guia de Desenvolvimento ---