Documentação Técnica de Frontend - Athus App

Este documento é um guia detalhado sobre a arquitetura, os componentes e os processos do aplicativo Athus, desenvolvido com React Native e TypeScript. Nosso objetivo é oferecer uma referência clara e direta para os desenvolvedores, facilitando a compreensão, a manutenção e as futuras expansões do projeto.

1. Introdução

1.1. Propósito do Documento

O principal objetivo desta documentação é ser um guia técnico completo para o aplicativo Athus. Queremos que ela ofereça uma visão aprofundada da sua estrutura, funcionalidades e das tecnologias que o impulsionam. Assim, tanto os novos membros da equipe poderão se familiarizar rapidamente com o projeto, quanto os desenvolvedores atuais terão uma fonte confiável para consultas e referências. Este documento cobre desde a visão geral do projeto até os detalhes mais específicos de componentes, gerenciamento de estado e comunicação com o backend, tudo para otimizar o desenvolvimento, a depuração e a manutenção do aplicativo.

1.2. Visão Geral do Projeto

O Athus App é um aplicativo frontend criado para empoderar pessoas em situação de vulnerabilidade social, oferecendo um espaço simples e acessível para divulgarem seus serviços. Ele foi pensado para atender a trabalhadores autônomos e informais, como jardineiros, pedreiros, eletricistas, diaristas e costureiras, que muitas vezes não conseguem divulgar seus talentos por falta de acesso a plataformas tradicionais. O Athus App preenche essa lacuna, permitindo que esses profissionais criem perfis, apresentem seus trabalhos e sejam encontrados por quem precisa, gerando renda, autonomia e dignidade. Além disso, o aplicativo também serve a contratantes e clientes que buscam serviços práticos, confiáveis e que desejam apoiar a economia local e quem mais precisa. Tudo isso é entregue com uma experiência de usuário fluida e responsiva em dispositivos móveis. As tecnologias que dão vida ao Athus App incluem:

- **Nome do Aplicativo:** Athus App O aplicativo Athus tem como propósito principal gerar oportunidades para pessoas em situação de vulnerabilidade social, permitindo que elas divulguem seus serviços de forma simples e acessível. Ele

oferece um espaço inclusivo e intuitivo para profissionais autônomos e informais divulgarem seus trabalhos e atraírem clientes, gerando renda, autonomia e dignidade. Trabalhadores autônomos e informais em situação de vulnerabilidade social (jardineiros, pedreiros, eletricistas, diaristas, costureiras, etc.) que buscam divulgar seus serviços e atrair clientes. Também inclui contratantes e clientes em geral que procuram serviços práticos, confiáveis e desejam apoiar a geração de renda local.
- **Tecnologias Principais:** React Native, TypeScript, Expo, Axios, React Navigation, AsyncStorage, entre outras bibliotecas listadas no `package.json`.
- **Versão Atual:** 1.0.0 (conforme `package.json`)

2. Arquitetura do Frontend

Esta seção explora a estrutura e o design do frontend do Athus App, revelando como ele foi construído para ser robusto e fácil de manter.

2.1. Visão Geral da Arquitetura

O Athus App foi concebido como um Single Page Application (SPA), aproveitando o poder do React Native e do Expo. Isso significa que conseguimos desenvolver uma única base de código que funciona perfeitamente em diversas plataformas, como iOS, Android e Web. A organização das pastas do projeto foi pensada para ser modular, o que facilita muito na hora de encontrar e dar manutenção em qualquer parte do código. Os diretórios mais importantes que você vai encontrar são:

- `app`: Contém as telas e a navegação principal do aplicativo, organizada pelo Expo Router.
- `components`: Armazena componentes de UI reutilizáveis, como botões, inputs, etc.
- `constants`: Define constantes globais, como cores, tamanhos de fonte e outras configurações.
- `context`: Gerencia o estado global da aplicação através de Context APIs, como `AuthContext` e `ThemeContext`.
- `hooks`: Contém hooks personalizados para encapsular lógicas reutilizáveis.
- `services`: Responsável pela comunicação com o backend, incluindo a configuração da instância do Axios e a lógica de autenticação.
- `types`: Define as interfaces e tipos TypeScript para garantir a segurança e consistência do código.

O fluxo de dados no aplicativo segue uma lógica reativa: quando o estado em contextos ou componentes muda, essas alterações são automaticamente refletidas nas interfaces

de usuário que dependem desses dados. A comunicação com o backend é centralizada nos serviços, que cuidam de todas as chamadas de API, autenticação e tratamento de erros de forma transparente.

2.2. Gerenciamento de Estado

No Athus App, o gerenciamento de estado é feito principalmente com a **Context API do React**. Por exemplo, o `AuthContext` (que você encontra em `context/AuthContext.tsx`) cuida de tudo relacionado à autenticação do usuário, como os dados do usuário logado (`user`) e se algo está carregando (`loading`). Ele também oferece funções para `login`, `logout`, `signUp` e `verifyEmail`, que se comunicam com os serviços de autenticação do backend e atualizam o estado do usuário em todo o aplicativo.

O `ThemeContext` (localizado em `context/ThemeContext.tsx`) é outro exemplo de gerenciamento de estado via Context API, responsável por controlar o tema visual do aplicativo. A escolha da Context API permite um gerenciamento de estado leve e eficiente para as necessidades do aplicativo, evitando a complexidade de bibliotecas de gerenciamento de estado mais robustas quando não são estritamente necessárias.

2.3. Comunicação com o Backend (APIs)

A comunicação do frontend com as APIs do backend é um ponto central no Athus App. Para isso, utilizamos o diretório `services` e a biblioteca **Axios** para todas as requisições HTTP. O arquivo `services/api.ts` é onde configuramos a instância do Axios, incluindo a `baseURL` do nosso backend (`http://felipemariano.com.br:8080/ProjetoAthus`).

Endpoints Utilizados:

Os principais endpoints consumidos pelo frontend incluem:

- `/auth/login`: Para autenticação de usuários.
- `/auth/signup`: Para registro de novos usuários.
- `/auth/verify-email`: Para verificação de e-mail.
- `/auth/refresh`: Para renovação de tokens de autenticação.

Formato dos Dados:

O formato dos dados esperados nas requisições e respostas é **JSON**.

Autenticação:

A autenticação é baseada em **tokens JWT (JSON Web Tokens)**. Após o login, o `accessToken` e o `refreshToken` são armazenados no `AsyncStorage`. O `accessToken` é enviado no cabeçalho `Authorization` como `Bearer Token` em todas as requisições subsequentes, exceto nas rotas de autenticação. Um interceptor de requisição no `services/api.ts` adiciona automaticamente o token às requisições.

Tratamento de Erros:

O tratamento de erros de API é implementado através de um interceptor de resposta no `services/api.ts`. Em caso de erro `401 Unauthorized` (não autorizado), o interceptor tenta automaticamente renovar o `accessToken` utilizando o `refreshToken`. Se a renovação for bem-sucedida, a requisição original é repetida com o novo token. Caso contrário, ou se o `refreshToken` estiver ausente, o usuário é deslogado e redirecionado para a tela de login, garantindo que a sessão do usuário seja mantida ou redefinida de forma segura.

3. Componentes

Esta seção mergulha nos componentes reutilizáveis do Athus App, que são como os tijolos da nossa interface de usuário. A ideia por trás da criação e do uso desses componentes é simples: queremos que o código seja fácil de reutilizar, modular e, acima de tudo, fácil de manter. Cada componente é pensado para ser independente e ter uma única função, o que torna o desenvolvimento e a correção de erros muito mais tranquilos.

3.1. Documentação de Componentes Individuais

Componente: `Button`

- **Propósito:** O `Button` é um dos nossos componentes mais importantes, ele é a base para qualquer interação clicável no aplicativo. Você vai encontrá-lo em diversas telas, seja para confirmar uma ação, navegar para outra parte do app ou enviar um formulário.
- **Props:** O componente `Button` é bem flexível e aceita as seguintes propriedades para você personalizar o comportamento e a aparência dele:

Propriedade	Tipo	Padrão	Descrição
`title`	`string`	-	O texto exibido dentro do botão.

Propriedade	Tipo	Padrão	Descrição
`onPress`	`() => void`	-	Função de callback que é executada quando o botão é pressionado.
`loading`	`boolean`	`false`	Se `true`, exibe um indicador de carregamento (`ActivityIndicator`) e desabilita o botão.
`disabled`	`boolean`	`false`	Se `true`, o botão é desabilitado e seu estilo é alterado para indicar inatividade.
`variant`	`'primary' | 'secondary' | 'outline'`	`'primary'`	Define o estilo visual do botão. As opções são `primary` (fundo principal), `secondary` (fundo secundário) e `outline` (borda com fundo transparente).
`style`	`ViewStyle`	-	Objeto de estilo opcional para o contêiner do botão, permitindo personalização adicional.
`textStyle`	`TextStyle`	-	Objeto de estilo opcional para o texto do botão, permitindo personalização adicional.

- **Estado Interno:** O `Button` é um componente "burro" no bom sentido! Ele não guarda nenhum estado interno; tudo o que ele faz é controlado pelas `props` que você passa para ele. Simples e direto!
- **Eventos:** O componente emite o evento `onPress` quando é tocado pelo usuário.
- **Exemplos de Uso:**

  ```typescript import { Button } from '@/components/Button';
  ```

```
console.log('Botão Confirmar pressionado')} />

// Botão primário padrão
// Botão secundário
console.log('Botão Cancelar pressionado')}
console.log('Botão Ver Detalhes pressionado')} variant="outline

// Botão com estilo outline
// Botão desabilitado
console.log('Botão Enviar pressionado')} di
console.log('Botão Carregando pressionado')} loading={true} />

// Botão com indicador de carregamento
// Botão com estilos personalizados
console.log('Botão Customizado pressionado')}
  style={{ backgroundColor: 'purple', borderRadius: 20 }}
  textStyle={{ color: 'yellow', fontSize: 18 }}
/>
```

- **Variações:** O componente `Button` é um camaleão! Ele se adapta com as seguintes variações visuais, tudo controlado pela prop `variant`:
    - `primary`: O estilo padrão, com um fundo preenchido pela nossa cor principal (`colors.primary`). É o botão que chama a atenção!
    - `secondary`: Um estilo com fundo preenchido pela cor secundária. Perfeito para ações menos prioritárias.
    - `outline`: Aqui, o botão tem uma borda e um fundo transparente, usando a cor primária para a borda e o texto. Ideal para um visual mais discreto. E tem mais! O botão também pode ficar `disabled` (desabilitado), com um fundo `colors.lightGray` para mostrar que não está ativo. E se algo estiver carregando, ele entra no modo `loading`, trocando o texto por um `ActivityIndicator` – assim, o usuário sabe que algo está acontecendo nos bastidores.
- **Dependências:** O componente `Button` depende de `TouchableOpacity`, `Text`, `StyleSheet`, `ActivityIndicator` do `react-native` e das cores definidas em `@/constants/colors`.
- **Capturas de Tela/Imagens:** [**Inserir capturas de tela ou imagens do componente Button em suas diferentes variações**]

4. Estilização e Design System

Esta seção explora como o visual do Athus App é construído e mantido, garantindo uma experiência de usuário consistente e agradável. Vamos mergulhar nos detalhes de como estilizamos nossos componentes e mantemos a coesão visual.

4.1. Guia de Estilo CSS

No Athus App, a estilização é feita principalmente com o `StyleSheet.create` do React Native. Isso nos permite criar estilos que ficam "dentro" de cada componente, facilitando a organização. As cores, por exemplo, são todas centralizadas no arquivo `constants/colors.ts`, o que garante que a paleta de cores seja sempre a mesma em todo o aplicativo e que seja fácil de atualizar. Essa abordagem modular significa que cada componente cuida da sua própria aparência, mas sempre seguindo um conjunto de cores e padrões globais.

- **Convenções de Nomenclatura:** As convenções de nomenclatura para estilos seguem o padrão camelCase, comum em JavaScript/TypeScript. Por exemplo, `buttonPrimary`, `textOutline`.
- **Organização de Arquivos CSS:** Os estilos são definidos diretamente nos arquivos dos componentes (`.tsx`), dentro de objetos `StyleSheet.create`, o que mantém o estilo e a lógica do componente juntos, facilitando a compreensão e a manutenção.

4.2. Design System (se aplicável)

Embora o Athus App não use um Design System formal com uma biblioteca de componentes dedicada, ele segue princípios de design que garantem uma consistência visual incrível. As cores são definidas em `constants/colors.ts`, que funciona como nosso "token de design" para a paleta de cores do aplicativo. A tipografia é cuidadosamente gerenciada com fontes personalizadas (Poppins), como você pode ver no `package.json` (`@expo-google-fonts/poppins`), e é aplicada de forma consistente em todo o app. Nossos componentes reutilizáveis, como o `Button`, são os blocos de construção que incorporam esses princípios de design, garantindo que tudo tenha a mesma cara e sensação.

4.3. Acessibilidade

A acessibilidade é um pilar no desenvolvimento do Athus App, principalmente por causa do nosso público-alvo. Mesmo que não tenhamos uma seção de acessibilidade explícita no código que analisamos, o React Native e o uso de componentes nativos já nos dão

uma base sólida. Por padrão, o React Native já incorpora muitas das diretrizes de acessibilidade do iOS e Android, como navegação por teclado e leitores de tela. Para garantir que o aplicativo seja o mais acessível possível, é super importante que os desenvolvedores sigam as melhores práticas do React Native, usando corretamente `accessibilityLabel`, `accessibilityHint`, `role` e `tabIndex` para elementos interativos, e que se certifiquem de que o contraste de cores é adequado para todos os usuários. Assim, garantimos que o Athus App seja realmente para todos!

5. Testes

Para garantir que o Athus App seja de alta qualidade e super estável, é essencial ter uma estratégia de testes bem robusta. Embora o repositório atual não tenha testes explícitos, a forma como desenvolvemos em React Native e TypeScript nos leva a pensar em testes unitários, de integração e de interface (UI).

5.1. Testes Unitários

Os testes unitários são como lupas que focam na menor parte do nosso código, verificando cada função, componente sem estado ou lógica específica de forma isolada. Para um projeto React Native com TypeScript, as ferramentas que mais indicamos são o **Jest** e a **React Native Testing Library**. O Jest é um gigante no mundo dos testes JavaScript, e a React Native Testing Library é perfeita para simular como o usuário interage com os componentes, garantindo que tudo funcione como deveria.

Exemplos de Testes Unitários para o Athus App:

- **Componentes:** Testar se o componente `Button` renderiza corretamente com diferentes `props` (e.g., `title`, `variant`, `disabled`, `loading`).
- **Funções de Utilitário:** Testar funções auxiliares em `hooks` ou `services` que realizam cálculos ou transformações de dados.
- **Contextos:** Testar se os contextos (`AuthContext`, `ThemeContext`) fornecem os valores corretos e se as funções de atualização de estado funcionam como esperado.

5.2. Testes de Integração

Os testes de integração são como pontes que conectam diferentes partes do nosso código, garantindo que elas conversem bem entre si. No Athus App, isso pode envolver:

- **Componentes e Contextos:** Testar se um componente que consome dados de um contexto se comporta corretamente quando o estado do contexto muda.

- **Componentes e APIs:** Testar a interação de um componente com as chamadas de API, simulando respostas de sucesso e erro do backend.
- **Navegação:** Testar se a navegação entre telas funciona conforme o esperado, passando os parâmetros corretos.

5.3. Testes de Interface (UI)

Os testes de UI são essenciais para ter certeza de que a interface do usuário está se comportando visualmente como esperamos em diferentes dispositivos e situações. Embora sejam um pouco mais complexos de configurar, eles são super importantes para garantir uma ótima experiência para o usuário. Ferramentas como **Detox** ou **Appium** podem ser usadas para fazer testes de ponta a ponta (end-to-end) em aplicativos React Native.

Considerações para Testes de UI no Athus App:

- **Fluxos de Usuário:** Testar o fluxo completo de login, cadastro, visualização de serviços e contato com profissionais.
- **Responsividade:** Garantir que a interface se adapta bem a diferentes tamanhos de tela e orientações.
- **Interações:** Verificar se os elementos interativos (botões, campos de entrada) respondem corretamente aos toques e gestos.

5.4. Estratégia de Mocking

Para testes de integração e unitários que dependem de coisas "de fora" (como APIs ou `AsyncStorage`), é super importante usar mocking. Isso nos permite simular o comportamento desses recursos, garantindo que nossos testes sejam rápidos, confiáveis e não dependam de fatores externos. Bibliotecas como `jest-mock-axios` ou as próprias ferramentas de mocking do Jest podem ser usadas para simular respostas de API e dados do `AsyncStorage`.