MANUAL DE CONTRIBUCIÓN

Sistema de red social ElephanTalk

ElephanTalk

Versión 1.0

Manual de Desarrollo y Contribución

Este documento establece los estándares para mantener la calidad, consistencia y escalabilidad del código fuente de **ElephanTalk**.

Guía de Estilo de Código

Para asegurar la legibilidad y el mantenimiento internacional del proyecto, se siguen las siguientes convenciones:

- **Idioma:** Todo el código (nombres de variables, funciones, comentarios y commits) debe escribirse estrictamente en **Inglés**.
- **Convenciones de Nombramiento (Naming Conventions):**
  - **Variables y Funciones:** Usar `camelCase`.
    - *Bien:* `getUserData`, `isModalOpen`, `chatMessage`.
    - *Mal:* `ObtenerUsuario`, `IsModalOpen`, `chat_message`.
  - **Componentes React:** Usar `PascalCase`.
    - *Ejemplo:* `ChatWindow.jsx`, `UserProfile.jsx`.
  - **Constantes:** Usar `UPPER_SNAKE_CASE` (ubicadas en carpeta `constants`).
    - *Ejemplo:* `API_BASE_URL`, `MAX_RETRY_COUNT`.
- **Estilos (CSS):** Se utiliza **Tailwind CSS**. Evitar archivos `.css` o `.scss` externos a menos que sea estrictamente necesario; priorizar clases utilitarias en el JSX.
- **Linting:** El proyecto cuenta con reglas definidas en `.eslintrc.cjs` y `.editorconfig`. Asegúrese de que su editor respete estas configuraciones automáticamente.

Proceso de Desarrollo y Git (Branching Strategy)

Utilizamos una estrategia basada en **Feature Branching** para mantener la rama principal limpia y funcional.

Flujo de Trabajo:

1. **Rama Principal:** `main` contiene siempre la versión estable y desplegable del proyecto. No se hace *commit* directo aquí.
2. **Nuevas Funcionalidades:** Para cada tarea, crear una rama desde `main` con el prefijo `feat/` o `fix/`.
   - Ejemplo: `feat/global-chat-ui` o `fix/login-error`.
3. **Commits:** Se utiliza la especificación **Conventional Commits**.
   - Estructura: `type: description`
   - `feat`: Para nuevas funcionalidades. (*Ej:* *`feat: add websocket connection logic`*)
   - `fix`: Para corrección de errores. (*Ej:* *`fix: resolve overlapping text in mobile view`*)

    - `refactor`: Cambios de código que no arreglan bugs ni añaden features.
4. **Merge:** Una vez finalizada la tarea, se realiza un Pull Request (PR) hacia `main`.

Cómo agregar nuevas funcionalidades (Arquitectura)

La arquitectura del frontend es modular. Para añadir una nueva *feature* (ej: "Sistema de Notificaciones"), siga este flujo lógico respetando la estructura de carpetas:

1. **Definir Modelos y Esquemas (`models/` y `schemas/`):** Defina la estructura de datos (interfaces) y las validaciones necesarias (ej: Zod schemas para formularios).
2. **Crear Servicios (`services/`):** Implemente las llamadas a la API en archivos aislados. No haga `fetch` directamente en los componentes.
    - *Ejemplo:* Crear `notificationService.js` con funciones como `getNotifications()`.
3. **Gestión de Estado (`store/` o `hooks/`):**
    - Si la lógica es compleja o reutilizable, cree un **Custom Hook** en `hooks/` (ej: `useNotifications.js`).
    - Si el estado es global, agréguelo al gestor de estado en `store/` o cree un nuevo Context en `providers/`.
4. **Componentes UI (`components/`):** Cree los componentes visuales pequeños y reutilizables.
    - *Ejemplo:* `NotificationCard.jsx`.
5. **Armado de Página (`pages/`):** Ensamble los componentes y la lógica en una vista principal dentro de `pages/`.
6. **Rutas:** Registre la nueva página en el archivo de rutas principal (`App.jsx` o configuración de router).