📱 [Mobile] 🚦 Dashboard de Tareas Colaborativas

Contexto

Imagina que estás construyendo una aplicación de gestión de tareas similar a Trello o Asana. Tu objetivo es desarrollar un "dashboard" donde un usuario pueda ver una lista de tareas, filtrarlas por estado y marcarlas como completadas. La aplicación debe ser resiliente a la falta de conexión y ofrecer una experiencia de usuario fluida.

Esta prueba está diseñada para evaluar tu capacidad para manejar un estado complejo, implementar una arquitectura robusta con un enfoque **offline-first**, y tu dominio de la integración con componentes nativos de UI.

- **Tiempo estimado:** 6 horas.
- **Uso de IA:** Se permite y recomienda el uso de asistentes de IA (GitHub Copilot, etc.) para acelerar el desarrollo. **Debes documentar detalladamente su uso en el README.**

🛠️ Stack Tecnológico

- **Framework:** Expo o React Native CLI.
- **Lenguaje:** TypeScript (obligatorio, con tipado estricto).
- **Manejo de estado:** Puedes elegir la solución de manejo de estado que prefieras, pero se recomienda usar una de estas opciones por alineación con el stack y facilidad de evaluación:
  - **Zustand**
  - **Redux Toolkit (Redux)**
  - **Context**
- Si eliges una alternativa distinta (ej. Jotai, Recoil, MobX, React Query como “state manager”, etc.), debes incluir en el **README**: por qué la elegiste

  cómo organizaste el estado (dominio/UI/server cache)

  cómo garantizas mantenibilidad y testabilidad.
- **Persistencia de datos: WatermelonDB** o **Realm** para la base de datos local (elegir uno). Esto es clave para el enfoque offline-first.
- **Navegación:** React Navigation.
- **API Externa:** Usa `<https://dummyjson.com/todos` > para obtener la lista inicial de tareas.

📦 Requisitos Funcionales

1. Arquitectura y Sincronización de Datos (Offline-First)

- **Sincronización Inicial:** Al abrir la app por primera vez, debe consumir los datos de `<https://dummyjson.com/todos>` y guardarlos en la base de datos local (WatermelonDB o Realm).
- **Lectura desde la BD local:** Toda la UI debe leer los datos exclusivamente desde la base de datos local, no directamente de la API (excepto en la sincronización). Esto garantiza que la app funcione sin conexión.
- **Modificaciones en Offline:** El usuario debe poder marcar tareas como "completadas" o "pendientes" incluso sin conexión a internet. Estos cambios se guardan localmente.
- **Mecanismo de "Pull-to-Refresh":** Al hacer "pull-to-refresh" en la lista de tareas, la app debe volver a consultar la API para obtener nuevas tareas y actualizar la base de datos local.

2. Funcionalidades de la UI

- **Pantalla del Dashboard:**
  - Mostrar la lista de tareas obtenidas de la base de datos local.
  - Implementar filtros para ver: "Todas", "Completadas", "Pendientes".
  - Cada ítem de la lista debe mostrar el título de la tarea y un checkbox o switch para cambiar su estado (`completed`).
- **Feedback Visual:** La UI debe reflejar inmediatamente los cambios de estado de una tarea, mostrando la interacción del usuario de forma instantánea.

3. ⭐ Requisito Senior Avanzado: Componente de UI Nativo

Debes crear y renderizar un componente de UI nativo llamado `AvatarView` que muestre las iniciales de un usuario con un fondo de color generado a partir de su nombre.

- **Funcionalidad:**
  - El componente recibirá una propiedad `name` desde React Native (ej: `name="Santiago Lopez"`).
  - La lógica nativa debe:
    i. Extraer las iniciales del nombre (ej: "SL").
    ii. Generar un color de fondo único basado en el hash del nombre.
    iii. Renderizar una vista circular (un `View` nativo) con el color de fondo y las iniciales centradas.
- **Implementación:**
  - **Android:** Crea un `SimpleViewManager` o `ViewManager` para exponer tu `AvatarView` nativo (escrito en Kotlin/Java).

- **iOS:** Crea un `RCTViewManager` para exponer tu componente (escrito en Swift/Objective-C).

- **Objetivo:** A diferencia de un módulo que solo ejecuta una acción, esto evalúa tu capacidad para **integrar vistas nativas personalizadas dentro del árbol de componentes de React**, una habilidad senior fundamental para optimizar el rendimiento y acceder a APIs de UI que no existen en React Native.

4. ➕ (OPCIONAL) Módulo Nativo: Adjuntar foto a una tarea (Cámara + guardado local)

Implementar un flujo para **tomar una foto con la cámara nativa**, guardarla localmente en el dispositivo y **vincularla a una tarea**.

Alcance funcional

1. **UI**
   - En el detalle de una tarea (o en cada item del listado) agregar un botón: **"Adjuntar foto"**.
   - Si la tarea ya tiene foto adjunta, mostrar una **miniatura** (o un placeholder con opción "Reemplazar foto").
2. **Bridge / Native Module (obligatorio para este plus)**
   - Crear un módulo nativo expuesto a JS, por ejemplo: `CameraModule`.
   - Debe abrir la **cámara nativa** y permitir tomar una foto.
   - Debe manejar el caso de **cancelación** (usuario cierra la cámara sin tomar foto) sin crashear.
3. **Permisos**
   - Solicitar permisos necesarios:
     - iOS: cámara (y describir uso en `Info.plist`).
     - Android: cámara (y permisos runtime si aplica según versión).
   - Manejar estado "denegado" mostrando feedback claro al usuario.
4. **Guardado local**
   - La foto debe guardarse en un directorio persistente dentro del sandbox de la app (no es necesario que aparezca en la galería del sistema).
   - El módulo debe retornar a JS un objeto con datos mínimos:
     - `uri` (string) del archivo guardado (ej. `file://...` o `content://...`)
     - `fileName` (string, opcional)
     - `width`, `height` (opcionales)
     - `size` (opcional)
5. **Persistencia offline-first**
   - Persistir la referencia en la base de datos local, por ejemplo:
     - un campo `attachmentUri` en `Task`, o

    - una entidad/tabla `TaskAttachment` relacionada por `taskId`.
  - La UI debe renderizar la imagen **leyendo desde la BD local**.

🚦 Criterios de Evaluación

- **Arquitectura Offline-First:** Correcta implementación del patrón de sincronización y lectura desde una base de datos local. Es el punto más importante de la prueba.
- **Estructura de Carpetas (Scaffolding):** Organización del código siguiendo principios de Clean Architecture o una estructura modular clara.
- **Calidad del Código:** Código legible, mantenible, y un uso avanzado de TypeScript (tipos, genéricos, etc.).
- **Pruebas Unitarias:** Cubrir la lógica de negocio, especialmente las interacciones con la base de datos (ej: marcar una tarea como completada, lógica de sincronización).
- **Implementación del Componente de UI Nativo:** Correcta creación y exposición de la vista nativa para ambas plataformas.

🔗 Entregables

- **Un repositorio público en GitHub** con todo el código fuente.
- **Un archivo README.md** que debe incluir:
  a. Instrucciones detalladas para instalar y ejecutar el proyecto (incluyendo cualquier paso de configuración nativa).
  b. Una explicación de la **arquitectura offline-first** y por qué elegiste WatermelonDB o Realm.
  c. Una sección obligatoria: **"Uso de IA"**, explicando qué herramientas usaste, para qué tareas y cómo tu supervisión fue clave para el resultado final.

¡Mucha suerte! Esta prueba es un reto, pero es una excelente oportunidad para demostrar tu dominio completo del ecosistema de React Native.