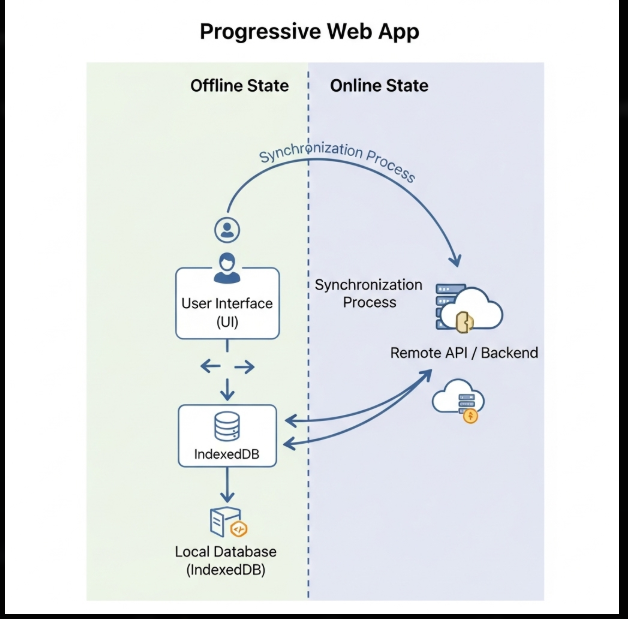
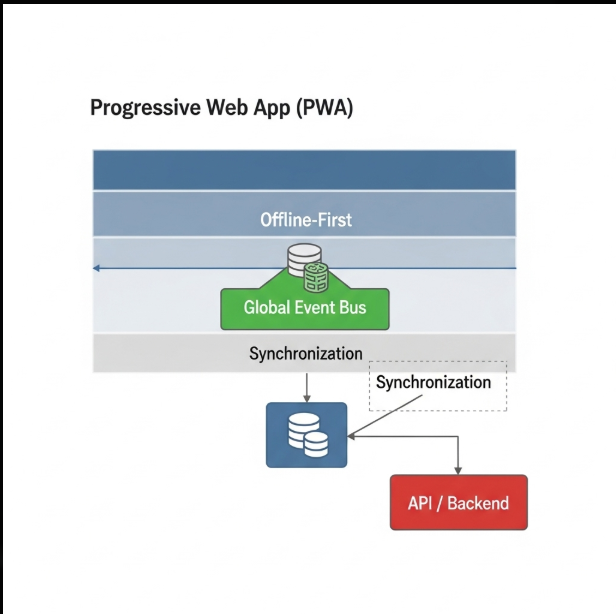
**ARQUITECTURA PWA**

****

**Explicación paso a paso**

1. **Usuario interactúa con la app**
   * Crea, modifica o elimina un recurso (usuario, grupo, tarea, etc.)
   * Todos los cambios se **guardan en IndexedDB** a través del **Repository**.
2. **Repository**
   * Encapsula operaciones CRUD locales.
   * Devuelve datos a la UI y también marca el syncStatus como pending o deleted.
3. **SyncSensor**
   * Monitorea el estado de sincronización de cada entidad.
   * Dispara eventos (itemSynced, itemFailed, itemDeleted) que actualizan la UI o alertas.
4. **NetworkSensor**
   * Detecta cuando el backend está online.
   * Activa el **Worker General**, que decide qué sincronizaciones deben correr.
5. **Worker específico**
   * Por ejemplo: userSensor, groupSensor, taskSensor.
   * Toma elementos pending o deleted y hace los requests correspondientes a la **API**.
   * Actualiza IndexedDB con la versión “synced”.
6. **API Backend**
   * Responde con los datos finales o IDs del backend.
   * Si es un nuevo registro offline, devuelve el id real del servidor.
7. **Sync Switch / Eventos**
   * Permite que la UI y otras partes de la app reaccionen al estado de sincronización.
   * Por ejemplo: marcar un usuario como “sincronizado” o mostrar error si falla.



**Explicación de la Arquitectura de Sincronización de la PWA**

El sistema utiliza una arquitectura **Offline-First** robusta, que prioriza la interacción local y garantiza una experiencia de usuario fluida, independientemente del estado de la conexión a internet. La sincronización con el servidor se maneja de forma asíncrona y transparente para el usuario.

A continuación, se detalla el flujo de trabajo paso a paso:

**1. Interacción del Usuario**

El usuario interactúa con la interfaz de la aplicación para crear, modificar o eliminar cualquier recurso (como un usuario, un grupo o una tarea).

**2. Capa de Repositorio (Offline-First)**

Todo cambio iniciado por el usuario es capturado por un **Repository** (TaskRepository, GroupRepository, etc.). El repositorio es la única capa que interactúa directamente con la base de datos local **IndexedDB**. Su función principal es:

* **Guardar cambios:** Almacena la operación localmente.
* **Marcar estado de sincronización:** Asigna un syncStatus al recurso, que puede ser pending (para creaciones o actualizaciones) o deleted (para eliminaciones).

**3. Control de Sincronización**

La lógica de sincronización se controla mediante dos componentes clave:

* **NetworkSensor**: Monitorea el estado de la conexión. Detecta cuándo la aplicación pasa de estar sin conexión a tener una conexión estable con el backend. Una vez que el backend está en línea, el sensor emite un evento server-online.
* **SyncSwitch / Eventos**: Permite a la interfaz de usuario habilitar o deshabilitar la sincronización. Esto se logra a través de un **Event Bus global** (appEvents), que permite a otros componentes (como el Worker específico) reaccionar al estado de la sincronización.

**4. Sincronización Asíncrona (Worker Específico)**

El evento server-online disparado por el NetworkSensor activa un **Worker específico** (syncTaskWorker, syncUserWorker, etc.). Este worker es el encargado de la sincronización.

1. **Consulta el repositorio:** Obtiene todos los recursos con un syncStatus de pending o deleted.
2. **Envía peticiones a la API:** Para cada recurso, realiza la petición HTTP apropiada (POST, PUT o DELETE) al **Backend API**.
3. **Actualiza IndexedDB:** Si la petición a la API es exitosa, el worker actualiza el recurso en la base de datos local:
   * Cambia su syncStatus a synced.
   * Si era una creación, actualiza el tempId con el id real del servidor.
   * Si era una eliminación, borra el registro de la base de datos.
4. **Emite eventos de sincronización:** El SyncSensor emite eventos (item-synced, item-deleted, etc.) para notificar a la aplicación que una operación de sincronización ha finalizado.

**5. Backend API**

El backend es la **fuente de verdad**. Su única responsabilidad es procesar las peticiones del worker y, en el caso de nuevas creaciones, devolver el ID permanente que se usará para futuras sincronizaciones.

**6. Notificación a la UI**

Finalmente, los eventos emitidos por el SyncSensor y el **Event Bus global** son consumidos por los componentes de la interfaz de usuario, permitiéndoles reaccionar en tiempo real, por ejemplo, mostrando una notificación de "Sincronizado" o un mensaje de error si la sincronización falla.

**Arquitectura de Sincronización**

Tu aplicación utiliza una arquitectura de "Offline-First" con un patrón de sincronización basado en **Sensores** y **Workers**. El objetivo es que las acciones del usuario (crear, actualizar, eliminar) se guarden de inmediato en la base de datos local y, cuando la conexión a internet esté disponible, se sincronicen con el servidor de manera automática y sin bloquear la interfaz.

Aquí está el rol de cada componente clave y cómo se relacionan:

**1. Los Sensores (networkSensor.ts, groupSensor.ts)**

Los sensores son los "ojos y oídos" de la aplicación. Su única responsabilidad es detectar eventos o cambios de estado y notificar a otros componentes.

* **networkSensor.ts**: Es el sensor más fundamental. Su única tarea es monitorear el estado de la conexión a Internet del dispositivo. Cuando el dispositivo se conecta o desconecta de la red, emite un evento ("online" o "offline") para que otros componentes sepan que el estado ha cambiado.
* **groupSensor.ts**: Este es el sensor de estado para la sincronización de grupos. Funciona como un **gestor de eventos**. Cuando el syncGroupWorker inicia, tiene éxito o falla, el groupSensor emite un evento ("start", "success", "failure", "itemSynced") que la interfaz de usuario puede escuchar para mostrar mensajes o actualizar su estado visual.

**2. El Orquestador (syncOrchestrator.ts)**

El orquestador es el intermediario que conecta los eventos detectados por los sensores con las acciones que deben ser ejecutadas por los workers. Su función es crucial.

* **syncOrchestrator.ts**: Contiene la función registerSyncTrigger. Esta función escucha el evento "online" del networkSensor y, en cuanto lo detecta, ejecuta la función del worker que le hayas asignado (syncPendingGroups). Este es el mecanismo que inicia la sincronización de forma automática.

**3. Los Workers (syncGroupWorker.tsx)**

Los workers contienen la lógica pesada de la aplicación, como la comunicación con el servidor y la manipulación de datos. Se activan por el orquestador y están diseñados para ser eficientes y asíncronos.

* **syncGroupWorker.tsx**: Este es el corazón de la sincronización. Su única función, syncPendingGroups, se encarga de todo el proceso de envío de datos al servidor.
  1. **Consulta la base de datos local**: Busca todos los grupos con los estados pending, updated y deleted.
  2. **Llama a los servicios**: Utiliza la lógica de un switch (implementado con if/else) para decidir qué hacer con cada grupo.
     + Si syncStatus === "pending": Llama al servicio para crear el grupo (api.post).
     + Si syncStatus === "updated": Llama al servicio para actualizar el grupo (api.put).
     + Si syncStatus === "deleted": Llama al servicio para eliminar el grupo (api.delete).
  3. **Actualiza el estado local**: Una vez que la operación en el servidor es exitosa, actualiza el estado de sincronización del grupo a "synced" en la base de datos local.

**4. Los Servicios (groupService.ts)**

Los servicios son la capa de comunicación con el backend. Encapsulan la lógica para interactuar con la API RESTful.

* **groupService.ts**: Este servicio se utiliza en el componente de gestión de grupos (groupusermanagement2.tsx) para guardar, actualizar y eliminar datos en la base de datos local. También es llamado por el worker para enviar los datos al servidor, pero de forma asíncrona.

**Flujo de la Sincronización**

Aquí tienes un resumen de cómo se conectan todos los componentes:

1. **El usuario** crea un grupo en la interfaz (groupusermanagement2.tsx) sin conexión a Internet.
2. **El groupService** guarda este grupo en la base de datos local con el syncStatus en **"pending"**.
3. **El networkSensor** detecta que el dispositivo ahora tiene conexión y emite un evento "online".
4. **El syncOrchestrator** escucha el evento "online" y llama al syncGroupWorker.
5. **El syncGroupWorker** busca los grupos con estado "pending". Al encontrar el nuevo grupo, utiliza la lógica if/else para identificar que debe ser creado en el servidor.
6. **El syncGroupWorker** llama al api.post del backend. Si es exitoso, actualiza el estado local a "synced" y llama al groupSensor.
7. **El groupSensor** emite un evento "success" o "itemSynced", que la interfaz de usuario escucha para actualizar la pantalla y confirmar que la sincronización se completó.