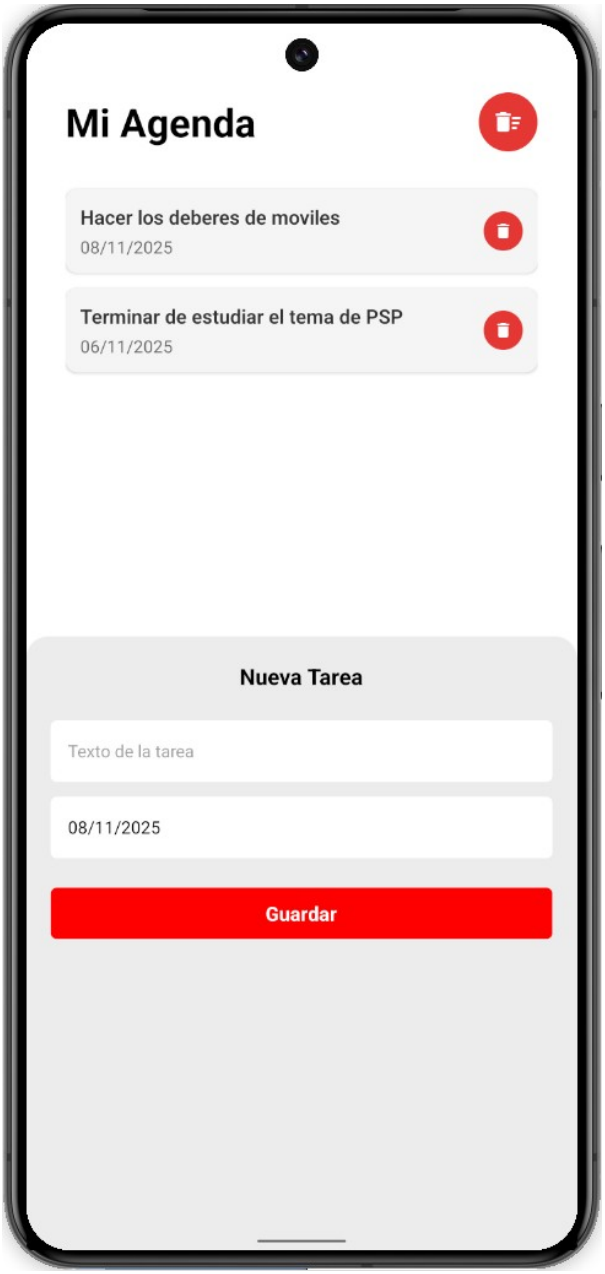
TUTORIAL 22

BASES DE DATOS LOCALES



En este tutorial vamos a realizar una sencilla agenda de tareas guardando los datos en una base de datos local **sqlite**. Veremos cómo crear la base de datos y realizar operaciones CRUD utilizando el lenguaje SQL

1.- El Expo Bare *Workflow*

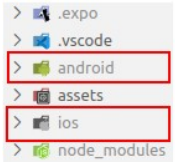
*Durante todo el curso hemos utilizado **Expo** para trabajar fácilmente con **React-Native,** y concretamente, su modalidad **Expo Managed Workflow.** En ella, hemos usado herramientas y comandos para gestionar ciertas complejidades de las apps (como poder ver todas las versiones a la vez con el comando **npx expo start**), aunque a cambio perdemos la posibilidad de usar librerías nativas (que son aquellas que no están programadas al 100% en **JavaScript**, sino que incluyen partes que dependen del sistema operativo concreto).*

*La librería que vamos a usar en este tutorial, **react-native-sqlite-storage**, es una de ellas y no funcionaría en un proyecto de los que estamos acostumbrados. Necesitamos crear el proyecto y pasarlo al **Expo Bare Workflow**, que es un modo de trabajo en el que las versiones de Android e iOS aparecen diferenciadas y se compilan de forma diferente.*

1. Crea un proyecto llamado **rn_agenda** y súbelo a **GitHub**
2. Crea una rama llamada **tutorial22**
3. Abre la terminal y teclea el comando **npx expo prebuild**

> *El comando **npx expo prebuild** sirve para convertir un proyecto **Expo Managed Workflow** a un proyecto **Expo Bare Workflow**.*

4. Observa que en el proyecto aparecen las carpetas **android** e **ios**, que son las versiones nativas (en **Java/Kotlin** y **Swift**) del proyecto



*Una vez que hemos pasado el proyecto al **Expo Bare Workflow** podemos hacer cosas como estas:*

- *Instalar librerías nativas (como la de este tutorial)*
- *Abrir las versiones nativas (con **Android Studio** o **XCode**) para añadir código nativo* (en **Java/Kotlin** y **Swift**)

5. Instala la siguiente librería
   - **react-native-sqlite-storage → npm install react-native-sqlite-storage**

> ☞ ***¿Para qué sirve esta librería?** Esta librería sirve para trabajar sobre una base de datos local **sqlite** en nuestra app y poder crearla y lanzarle sentencias SQL*

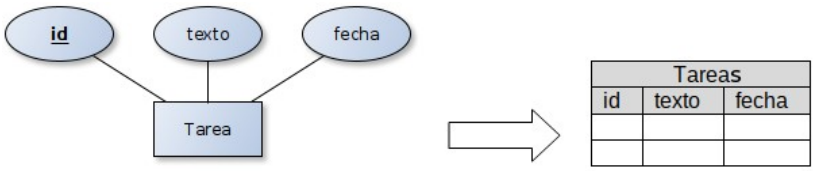
6. Instala además estas librerías:
   - **expo-vector-icons → npx expo install @expo/vector-icons**
   - **dayjs → npm install dayjs**
7. Para ejecutar la app, ahora tenemos que compilar y lanzar cada versión específica, según estos comandos:
   - **Android → npx expo run:android**

- **iOS → npx expo run:ios**

2.- Modelo de datos

*Siempre que tenemos que hacer acceso a datos, es muy útil hacer un modelo que refleje las entidades que maneja la app. En nuestro caso, la app es una agenda muy sencilla que almacena solamente tareas, donde cada tarea tiene un **id**, un **texto** descriptivo y una **fecha***

Puesto que trabajaremos directamente sobre una base de datos relacional, tenemos que realizar un diseño previo de la base de datos, que en este caso es muy sencillo:



*En la app deberemos crear un tipo llamado **Tarea** que represente una tarea de la base de datos, y por comodidad, crearemos un tipo **Tareas** que será un alias para el tipo **Array<Tarea>***

1. Crea una carpeta llamada **model** y en ella un archivo llamado **Tarea.ts**
2. En **Tarea.ts** crea y exporta los tipos **Tarea** y **Tareas**

```
type Tarea = {
    id:number
    texto:string
    fecha:string // la guardaremos como una cadena de texto en formato día/mes/año
}
type Tareas = Array<Tarea>
export {Tarea, Tareas}
```

3.- Abrir la base de datos

Lo primero que vamos a hacer es abrir la base de datos para poder trabajar con ella.

*En bases de datos montadas en servidores (como **MySQL**) hablamos de **conectarnos** a la base de datos, puesto que hay que abrir una conexión con un puerto de un servidor. En cambio, en bases de datos locales (que solo constan de un archivo, como **sqlite**) hablamos de **abrir** la base de datos.*

1. Crea una carpeta llamada **database**
2. Crea en dicha carpeta un archivo llamado **BaseDatos.ts**
3. En **BaseDatos.ts** llama a la función **SQLite.enablePromise** pasándole **true**

```
import SQLite from 'react-native-sqlite-storage'
SQLite.enablePromise(true)
```

*Esta función hace que la librería[1] trabaje con **Promises** y de esa forma se puedan usar los mecanismos habituales de llamadas a funciones asíncronas.*

4. Crea (pero **no** exportes) una variable llamada **baseDatos** cuyo tipo será **SQLite.SQLiteDatabase** o **null**, e inicialízala con **null**

```
1.  let baseDatos: SQLite.SQLiteDatabase | null = null
```

*Esta variable es el objeto que va a guardar la base de datos una vez abierta. Inicialmente vale **null** porque la base de datos aún no está abierta. Se deja privado en el archivo para que la app no acceda directamente a él, sino que lo haga por medio de la función **getBaseDatos** que crearemos a continuación*

5. Crea y exporta una función asíncrona llamada **getBaseDatos** que haga esto:
   - Si **baseDatos** no es nula, devuelve **baseDatos**
   - En caso contrario, inicializa **baseDatos** con lo que nos devuelve **SQLite.openDatabase** para abrir la base de datos, y retorna **baseDatos**

```
1.  export async function getBaseDatos():Promise<SQLite.SQLiteDatabase>{
2.      if(baseDatos == null ){
3.          baseDatos = await SQLite.openDatabase({
4.              name:"agenda.db",
5.              location:"default",
6.          })
7.      }
8.      return baseDatos
9.  }
```

*La función **SQLite.openDatabase** recibe un objeto con el nombre (**name**) de la base de datos y su ubicación (**location**) en el dispositivo. Al pasarle **default**, se toma la ubicación por defecto que use el sistema operativo.*

4.- Crear las tablas de la base de datos

Una vez que podemos abrir la base de datos, podemos crear las tablas con las que vamos a trabajar, y eso debemos la primera vez que se usa la app.

*Para lanzar sentencias **SQL** a la base de datos usaremos el método **executeSQL** del objeto **baseDatos***

1. En la carpeta **database** crea un archivo llamado **TareaDAO.ts**

*Se suele llamar **DAO (Data Access Object)** a los objetos que realizan operaciones crud sobre una tabla de una base de datos relacional.*

*En este archivo vamos a crear funciones para realizar sobre la tabla **Tareas** todas las acciones que necesita hacer nuestra app*

[1] Por defecto, la librería usa **callbacks,** que son funciones que se pasan como parámetro con las acciones a realizar cuando finaliza la tarea asíncrona.

2. En **TareaDAO.ts** crea una función asíncrona llamada **crearTablas**, que lanzará a la base de datos la correspondiente sentencia **create table.** Las funciones que usará son:
    - **getBaseDatos:** Permite obtener la base de datos ya abierta (o la abre si aún no lo está)
    - **executeSQL:** Recibirá una sentencia **create table**

```
export async function crearTablas(){
    const sql=`
        CREATE TABLE IF NOT EXISTS TAREAS(
            ID INTEGER PRIMARY KEY AUTOINCREMENT,
            TITULO TEXT NOT NULL,
            FECHA TEXT NOT NULL
        )
    `
    const bd = await getBaseDatos()  // abre la base de datos (o la obtiene ya abierta)
    await bd.executeSql(sql) // lanza la sentencia sql a la base de datos
}
```

*Los tipos de datos que admite **sqlite** son:*

- ***INTEGER*** → *número entero (también reconoce **INT, TINYINT, BIGINT**)*
- ***TEXT*** → *cadena de texto (también reconoce **CHAR, VARCHAR, CLOB**)*
- ***REAL*** → *número con decimales (también reconoce **DOUBLE, FLOAT**)*
- ***NUMERIC*** → *es **INTEGER** o **REAL** según el valor (también reconoce y tiene asociados **BOOLEAN, DATE, DECIMAL**)*
- ***BLOB*** → *datos en binario*

3. Abre **App.tsx** y crea una función **mostrarError** que reciba un mensaje y muestre en una ventana emergente dicho mensaje, con título “Error”

```
function mostrarError(mensaje:string){
  Alert.alert("Error",mensaje)
}
```

4. Añade una función llamada **accionCrearTablas**, que llame a la función **crearTablas** y si todo va bien, muestre un mensaje en la terminal y si hay un error, abra una ventana emergente con el mensaje

```
function accionCrearTablas(){
  crearTablas()
    .then(() => console.log("Tablas creadas"))
    .catch(error => mostrarError(error.message))
}
```

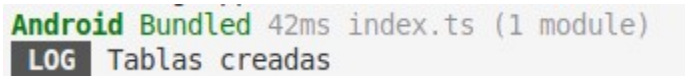
☞ ***¿Por qué se pone error.message?*** *Se debe a que los errores que lanza la librería de la base de datos devuelven un objeto en cuyo campo **message** se encuentra la descripción del error*

5. Añade un efecto que llame a **accionCrearTablas** una sola vez, al crear el componente principal **App**

```
export default function App() {
  useEffect( accionCrearTablas, [] )
  // resto omitido
```

```
4.  }
```

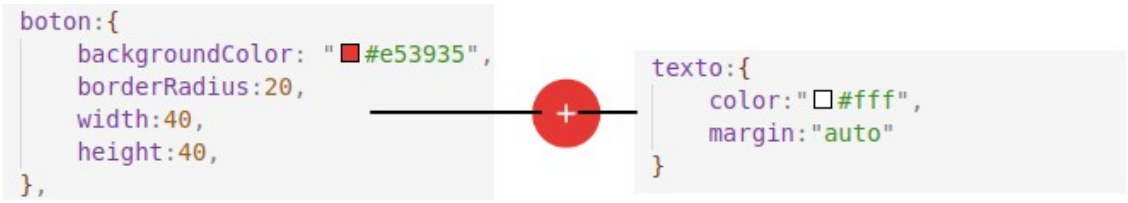
6. Ejecuta la app y comprueba que la base de datos se abre y aparece en el terminal el mensaje “Tablas creadas”



5.- Diseño del componente BotonCircular

Ahora que ya tenemos creada la base de datos, vamos a pasar a la parte de crear los componentes que intervienen en la app. Comenzaremos por un botón de apariencia circular.

1. Crea una carpeta **components** y en ella un archivo **BotonCircular.tsx**
2. Abre **BotonCircular.tsx** y teclea **rnfs+intro**
3. Sin mirar la solución, diseña el siguiente componente, añadiéndole estos props:
    - **icono:** Es un **string** que solo puede tomar los valores **add, delete** y **delete-sweep**
    - **sizeIcono:** Es un número que indica el tamaño del icono
    - **sizeBoton:** Es un número que indica el tamaño del contenedor
    - **onPress:** Es una función que se ejecuta al pulsar el botón




```
1.  type BotonCircularProps = {
2.      icono :"add" | "delete" | "delete-sweep"
3.      sizeIcono:number
4.      onPress : () => void
5.  }
6.  export default function BotonCircular({icono, sizeIcono, onPress}:BotonCircularProps) {
7.    const estilo ={
8.          width:sizeBoton,
9.          height:sizeBoton,
10.         borderRadius: sizeBoton/2
11.   }
12.   return (
13.     <Pressable style={[styles.boton, estilo]} onPress={onPress}>
14.         <MaterialIcons
15.             name={icono}
16.             size={sizeIcono}
17.             color={"red"}
18.             style={styles.texto}
19.         />
20.     </Pressable>
21.   )
22. }
```

6.- Diseño del componente ItemTarea

*La app usará un **FlatList** para mostrar todas las tareas disponibles en la base de datos. Por ese motivo, es necesario crear previamente un componente llamado **ItemTarea** que servirá para mostrar cada tarea de la lista.*

1. Crea en **components** un archivo llamado **ItemTarea.tsx**
2. Abre **ItemTarea.tsx** y teclea **rnfs+intro**
3. Sin mirar la solución, crea el componente **ItemTarea** de manera que tenga estos dos props:
    - **item:** Es un objeto **Tarea**
    - **accionBorrarTarea:** Es la función que se lanza al pulsar el botón de borrar. Su objetivo es recibir el **id** de una tarea y borrarla


```tsx
type ItemTareaProps = {
    item: Tarea,
    accionBorrarTarea: (idTarea:number) => void
}
export default function ItemTarea({item, accionBorrarTarea}:ItemTareaProps) {
  return (
    <View style={styles.contenedor}>
        <View style={styles.columnaTexto}>
            <Text style={styles.texto}>{item.texto}</Text>
            <Text style={styles.fecha}>{item.fecha}</Text>
        </View>
        <BotonCircular
            icono={"delete"}
            sizeBoton={32}
            sizeIcono={16}
            onPress={() => accionBorrarTarea(item.id)}
        />
    </View>
  )
}
```

7.- Diseño del componente principal

*Vamos a programar el componente principal, que estará formado por un **FlatList** que muestre todas las tareas que hay en la agenda y un botón para añadir una nueva tarea*

1. Abre **App.tsx** y añade estas variables de estado (con sus setters):
    - **listaTareas:** Será la lista de objetos **Tarea** que hay en la base de datos
    - **modalVisible:** Será un **boolean** que indicará si mostrar un modal con un formulario para crear una nueva tarea

```
export default function App() {
  const [listaTareas, setListaTareas] = useState<Tareas>([])
  const [modalVisible, setModalVisible] = useState(false)
  useEffect( accionCrearTablas , [] )
  // resto omitido
}
```

2. Añade a **App** las siguientes funciones (por ahora vacías), en las que se realizarán las funciones de la app:
    - **accionNuevaTarea:** Recibe un texto y una fecha y registra en la base de datos una tarea con ellas.
    - **accionBorrarTarea:** Recibe el **id** de una tarea y la borra
    - **accionBorrarTodo:** Borra todas las tareas de la base de datos

```
function accionNuevaTarea(texto:string, fecha:string){
  // registra una tarea en la base de datos
}
function accionBorrarTarea(id:number){
  // borra la tarea con el id pasado como parámetro
}
function accionBorrarTodo(){
  // borra todas las tareas de la base de datos
}
```

3. Crea una función llamada **getItemTarea**, que reciba un objeto **Tarea** y devuelva la etiqueta **ItemTarea** que se mostrará en el **FlatList**. Se le pasará en su prop la función **accionBorrarTarea**

```
function getItemTarea(tarea:Tarea){
  return <ItemTarea item={tarea} accionBorrarTarea={accionBorrarTarea}/>
}
```

4. Sin mirar la solución, haz que la función **App** devuelva el siguiente diseño:


```
export default function App() {
  // inicio omitido
  return (
    <View style={styles.contenedor}>
      <View style={styles.fila}>
        <Text style={styles.titulo}>Mi Agenda</Text>
        <BotonCircular icono={"delete-sweep"} sizeBoton={48} sizeIcono={24}
          onPress={accionBorrarTodo} />
```

```
        </View>
        <View style={styles.fab}>
          <BotonCircular icono={"add"} sizeBoton={48}  sizeIcono={36}
            onPress={() => setModalVisible(true)}
          />
        </View>
        <FlatList style={{flex:1, borderWidth:1}} data={listaTareas}
          keyExtractor = {item => item.id.toString()} renderItem={ ({item}) => getItemTarea(item)}
          ListEmptyComponent={ () => <Text style={{textAlign:"center"}}>No hay tareas</Text>}
        />
      </View>
    )
}
```

*Se pone **zIndex:1000** al botón para que el **FlatList** no lo tape*

9.- Apertura del modal

Vamos a hacer que al pulsar el botón de nueva tarea se abra un modal que tenga dos partes: una inferior donde se mostrará un formulario (que diseñaremos en el punto siguiente) y otra superior que sirva para cerrar el modal al pulsar en ella.

1. En **App.tsx** añade los siguientes estilos

```
exteriorModal:{
  flex:1,
},
interiorModal:{
  position: "absolute",
  bottom: 0,
  width: "100%",
  height: "50%", // queremos que la parte inferior del modal ocupe el 50% de la pantalla
  borderTopLeftRadius: 18,
  borderTopRightRadius: 18,
  padding: 20,
  backgroundColor: "#ececeff",
}
```

*Recuerda que el componente **Modal** ocupa toda la pantalla, y lo que hacemos es dividirla en dos mitades que tendrán los estilos anteriores*

2. Al final de la función **App** añade un bloque de renderizado condicional que muestre un componente **Modal** dividido en un **Pressable** y un **View** que tengan los estilos anteriores. Al pulsar la zona superior, se ocultará el modal

```
export default function App() {
  // inicio omitido
  return (
      // inicio omitido
      {
        modalVisible && (
          <Modal animationType={"slide"} transparent={true}>
            <Pressable style={styles.exteriorModal} onPress={()=>setModalVisible(false)}/>
            <View style={styles.interiorModal}>
            </View>
          </Modal>
        )
      }
    </View>
  )
}
```

8.- Diseño del componente NuevaTarea

*A continuación, vamos a hacer un componente **NuevaTarea** que contenga un formulario para sirva para que el usuario escriba los datos de una nueva tarea*

3. En **components** crea un archivo llamado **NuevaTarea.tsx**
4. Abre **NuevaTarea.tsx** y pulsa **rnfs+intro**
5. Añade a **NuevaTarea** el prop **accionNuevaTarea,** que será una función que recibe el título y fecha de una tarea y la crea en la base de datos

```
type NuevaTareaProps = {
    accionNuevaTarea: (texto:string, fecha:string) => void
}
export default function NuevaTarea({accionNuevaTarea, setModalVisible}:NuevaTareaProps) {
  // resto omitido
}
```

6. Añade a **NuevaTarea** las siguientes variables de estado (y sus setter):
    - **textoTarea:** Es el texto de la tarea que se va a crear
    - **fechaTarea:** Es el texto de la fecha de la tarea que se va a crear. Por defecto, será la fecha de hoy en formato día/mes/año

```
export default function NuevaTarea({accionNuevaTarea}:NuevaTareaProps) {
    const [textoTarea, setTextoTarea] = useState("")
    const [fechaTarea, setFechaTarea] = useState(dayjs(Date()).format("DD/MM/YYYY"))
    // resto omitido
}
```

7. Sin mirar la solución, haz que **NuevaTarea** devuelva el siguiente diseño:


```
export default function NuevaTarea({accionNuevaTarea}:NuevaTareaProps) {
   // inicio omitido
   return (
    <View style={styles.contenedor}>
      <Text style={styles.texto}>Nueva Tarea</Text>
      <View style={styles.cuadroTexto}>
        <TextInput
            placeholder={"Texto de la tarea"}
            value={textoTarea} on
            onChangeText={setTextoTarea}
        />
      </View>
      <View style={styles.cuadroTexto}>
        <TextInput value={fechaTarea} onChangeText={setFechaTarea}/>
      </View>
      <Pressable style={styles.boton} onPress={() => accionNuevaTarea(textoTarea,fechaTarea)}>
        <Text style={styles.textoBoton}>Guardar</Text>
```

```
      </Pressable>
    </View>
  )
}
```

9.- Creación de una nueva tarea

*Ahora que ya está terminado el diseño de la app, vamos a programar la funcionalidad completando las funciones vacías que hemos creado en **App.** Esas funciones llamarán a funciones auxiliares que se encargarán de realizar las tareas en la base de datos.*

1. Abre **TareaDAO.ts**, crea y exporta una función asíncrona llamada **crearTarea** que reciba un texto y una fecha (ambos **string)** y devuelva un objeto **Tarea** (que será la tarea que se creará en la base de datos, con todos sus campos)

```
export async function crearTarea(texto:string, fecha:string):Promise<Tarea>{
   // aquí crearemos una tarea en la bd y devolveremos un objeto Tarea con la tarea creada
}
```

2. Dentro de la función, crea un **string** con la sentencia para insertar la tarea, teniendo en cuenta que el **id** es autoincrementado. Todos los datos que sean variables se pondrán como interrogantes **?** en la sentencia

```
export async function crearTarea(texto:string, fecha:string):Promise<Tarea>{
    const sql = `INSERT INTO TAREAS (TITULO,FECHA) VALUES (?,?)`
}
```

*El uso de interrogantes permitirá inyectar de forma segura los valores que faltan en la sentencia, y así evitar ataques de **SQL Injection***

3. Lanza la sentencia usando **executeSql** de esta forma:
   - Pasa a **executeSql** la sentencia y una lista con las variables que se van a sustituir en los interrogantes
   - Recoge en una variable llamada **listaResultados** el resultado de la función (esto lo hacemos porque necesitamos recuperar el **id** que nos genera el **autoincrement**)

```
export async function crearTarea(texto:string, fecha:string):Promise<Tarea>{
    const sql = `INSERT INTO TAREAS(TITULO,FECHA) VALUES (?,?)`
    const bd = await getBaseDatos()
    const listaResultados = await bd.executeSql(sql, [texto,fecha])
}
```

*La función **executeSql** permite lanzar varias sentencias **SQL** a la vez, y por eso devuelve una lista con los resultados de cada una de las sentencias lanzadas.*

*Los resultados obtenidos contienen información útil sobre la ejecución de una sentencia, como por ejemplo, el id generado, que se obtiene con el campo **insertId***

4. Haz que la función devuelva un objeto **Tarea** cuyos campos sean el texto y fecha recibidos como parámetros, y el **id** sea el campo **insertId** del primer objeto de la lista de resultados

```
export async function crearTarea(texto:string, fecha:string):Promise<Tarea>{
```

```
    const sql = `INSERT INTO TAREAS(TITULO,FECHA) VALUES (?,?)`
    const bd = await getBaseDatos()
    const listaResultados = await bd.executeSql(sql)
    return {
        id:listaResultados[0].insertId,
        texto:texto,
        fecha:fecha
    }
}
```

> ☞ ***¿Por qué no validamos la hora?*** *No hemos validado que la hora tenga un formato correcto porque en el tutorial siguiente la seleccionaremos de un diálogo*

5. Abre **App.tsx** y completa la función **accionNuevaTarea** para que llame a **crearTarea** y si todo va bien, actualice **listaTareas** con el objeto **Tarea** recibido tras la ejecución asíncrona y se cerrará el modal. Como sabes, la actualización de la lista debe hacerse creando una lista nueva.

```
function accionNuevaTarea(texto:string, fecha:string){
  crearTarea(texto,fecha)
    .then( tarea => {
      const nuevaLista = [tarea, ...listaTareas]
      setListaTareas(nuevaLista)
      setModalVisible(false)
    })
    .catch(error => mostrarError(error.message))
}
```

10.- Borrar una tarea

*Vamos a programar ahora la funcionalidad de borrar una tarea. Es muy similar al caso de creación de tareas, y bastará con crear una sentencia de borrado a la que inyectaremos el **id** de la tarea que queremos borrar.*

1. Abre **TareaDAO.ts**, crea y exporta una función asíncrona llamada **borrarTarea** que reciba el id de una tarea y borre de la base de datos la tarea que tiene dicho id.

```
export async function borrarTarea(idTarea:number){
    const sql=`DELETE FROM TAREAS WHERE ID=?`
    const bd = await getBaseDatos()
    await bd.executeSql(sql,[idTarea])
}
```

2. Abre **App.tsx** y programa la función **accionBorrarTarea** de manera que se llame a **borrarTarea** y si todo va bien, borre de **listaTareas** la tarea que tiene ese id (ya sabes que eso debe hacerse creando una nueva lista con **filter**)

```
function accionBorrarTarea(id:number){
  borrarTarea(id)
    .then( () => {
      const nuevaLista = listaTareas.filter( t => t.id!==id)
      setListaTareas(nuevaLista)
    })
}
```

11.- Borrar todas las tareas

El siguiente paso consistirá en borrar todas las tareas de la base de datos, lo cual es tan sencillo como lanzar la correspondiente sentencia de borrado.

1. Abre **TareaDAO.ts**, crea y exporta una función asíncrona llamada **borrarTodo** que borre de la base de datos todas las tareas

```
export async function borrarTodo(){
    const sql=`DELETE FROM TAREAS`
    const bd = await getBaseDatos()
    await bd.executeSql(sql)
}
```

2. Abre **App.tsx** y programa la función **accionBorrarTodo** de manera que se muestre una ventana emergente para confirmar el borrado. Si se acepta, se llamará a una función **realizarBorrado** donde se llamará a **borrarTodo**. En caso de que se borren todas las tareas, se actualizará **listaTareas** con una lista vacía.

```
function accionBorrarTodo(){
  Alert.alert(
    "¿Deseas borrar todas las tareas?",
    "Esta acción no se puede deshacer y borrará todas las tareas",
    [
      {text:"Si, eliminar", onPress: realizarBorrado},
      {text:"No, cancelar"}
    ]
  )
}
function realizarBorrado(){
  borrarTodo()
    .then( () => setListaTareas([]))
    .catch(error => mostrarError(error.message))
}
```

12.- Consultar todas las tareas

Lo único que falta para terminar la app es hacer que al iniciarse la app, se rellene la lista de tareas con todas las tareas almacenadas en la base de datos. Para hacer esto, primero programaremos una función que realice dicha consulta y después añadiremos un efecto que se ejecute al iniciarse la app para llamarla.

1. Abre **TareaDAO.ts**, crea y exporta una función asíncrona llamada **consultarTareas** que consulte todas las tareas de la base de datos, y nos devuelva una lista de objetos **Tarea** con todas ellas.

```
export async function consultarTareas():Promise<Tareas>{
    const lista:Tareas = [] // aquí guardaremos la lista de tareas
    return lista
}
```

2. Añade a continuación la sentencia para consultar todas las tareas de la base de datos, ordenadas por fecha descendente

```
export async function consultarTareas():Promise<Tareas>{
    const lista:Tareas =[] // aquí guardaremos la lista de tareas
    const sql=`SELECT * FROM TAREAS ORDER BY FECHA DESC`
    return lista
}
```

3. Abre la base de datos y lanza la sentencia sql sobre ella, recogiendo en una variable llamada **resultado** el primer resultado devuelto por **executeSql**

```
export async function consultarTareas():Promise<Tareas>{
    const lista:Tareas =[] // aquí guardaremos la lista de tareas
    const sql=`SELECT * FROM TAREAS ORDER BY FECHA DESC`
    const bd = await getBaseDatos()
    const [resultado] = await bd.executeSql(sql)
    return lista
}
```

*Al escribir **[resultado]** estamos recogiendo en una variable llamada **resultado** el primer elemento de la lista que devuelve **executeSql**. Esto se denomina "desestructurar la lista"*

4. Crea una variable llamada filas que sea igual al campo rows de resultado

```
export async function consultarTareas():Promise<Tareas>{
    const lista:Tareas =[] // aquí guardaremos la lista de tareas
    const sql=`SELECT * FROM TAREAS ORDER BY FECHA DESC`
    const bd = await getBaseDatos()
    const [resultado] = await bd.executeSql(sql)
    const filas = resultado.rows
    return lista
}
```

*El campo **rows** del resultado es un objeto que contiene las filas consultadas, y escribiendo **filas(i)** obtenemos los datos de la fila encapsulados en un objeto.*

5. Usa un bucle **for** para recorrer **filas** (su tamaño es **filas.length**) y añade a la variable **lista** el objeto que encapsula los datos de cada fila recorrida, siendo sus claves, los nombres de las columnas de la tabla.

```
export async function consultarTareas():Promise<Tareas>{
    const lista:Tareas =[] // aquí guardaremos la lista de tareas
    const sql=`SELECT * FROM TAREAS ORDER BY FECHA DESC`
    const bd = await getBaseDatos()
    const [resultado] = await bd.executeSql(sql)
    const filas = resultado.rows
    for(let i = 0;i<filas.length;i++){
        const tarea: Tarea = {
            id:filas.item(i).ID,
            texto:filas.item(i).TITULO,
            fecha:filas.item(i).FECHA,
        }
        lista.push(tarea)
    }
    return lista
}
```

*Al escribir **filas.item(i)** estamos obteniendo un objeto con los datos de la fila de la tabla. Los nombres de las claves de ese objeto son las correspondientes columnas, y cuando se creó la tabla (ver la sentencia **create table**) se pusieron en mayúscula*

6. Abre **App.tsx** y añade una función llamada **accionConsultarTareas**, que llamará a **consultarTareas** y rellenará **listaTareas** con la lista obtenida

```
function accionConsultarTareas(){
  consultarTareas()
```

```
    .then(lista => setListaTareas(lista))
    .catch(error => mostrarError(error.message))
}
```

7. En **App,** después del efecto que crea las tablas, añade un efecto que llame a **accionConsultarTareas** y que se ejecute solo una vez, al crearse el componente principal

```
export default function App() {
  const [listaTareas, setListaTareas] = useState<Tareas>([])
  const [modalVisible, setModalVisible] = useState(false)
  useEffect( accionCrearTablas , [] )
  useEffect( accionConsultarTareas , [] )
  // resto omitido
}
```

8. Haz un commit titulado “finalizado el tutorial 22”
9. Sitúate en la rama **main** y mezcla en ella la rama **tutorial22**
10. Haz un **push**