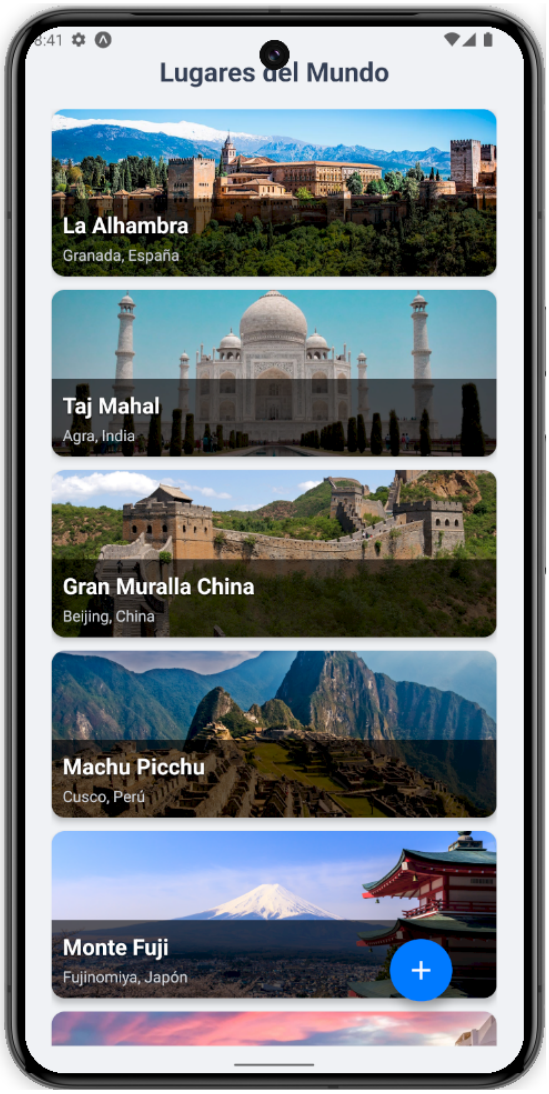
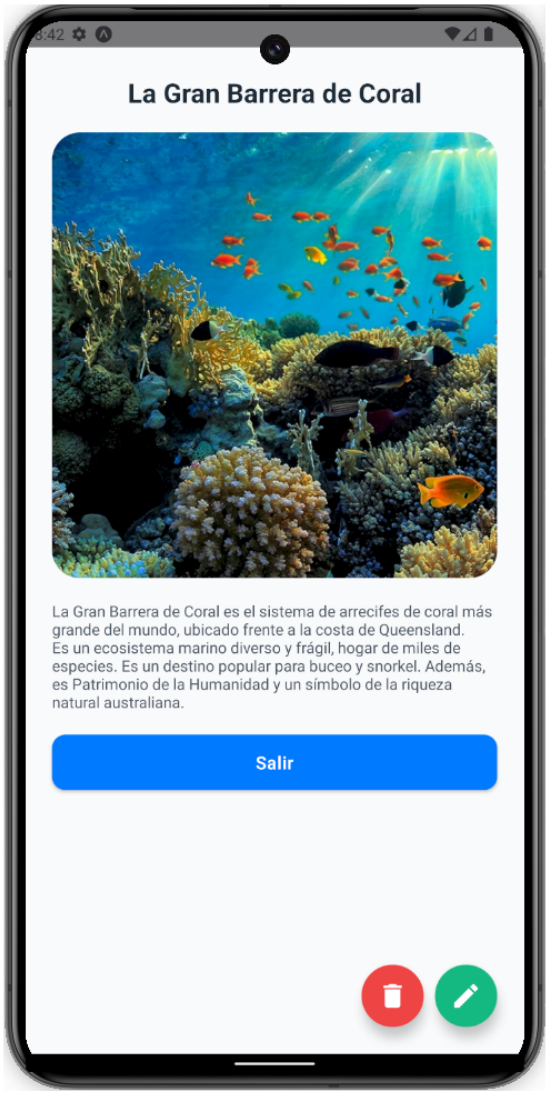
TUTORIAL 18

OPERACIONES CRUD SOBRE APIS





En este tutorial vamos a hacer una app que realice operaciones **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete) por medio de un API. Este tipo de apps son muy comunes en la práctica y veremos cómo realizarlas mediante **unidirectional data flow**, que es el modelo al que estamos acostumbrados en React Native para pasar mediante props, datos de un componente padre a sus hijos, pero no al revés.

1.- Inicio del proyecto

*En los proyectos que trabajan con acceso a datos (Apis, bases de datos, etc) es muy recomendable utilizar un **modelo,** que son tipos de datos que representan los conceptos que aparecen en la app (por ejemplo, **Lugar**). Por ese motivo, este proyecto se realizará desde su inicio en el lenguaje **TypeScript**.*

1. Crea un proyecto llamado **lugares** y súbelo a GitHub
2. Crea las siguientes carpetas en el proyecto:
    - **model:** contiene archivos con los tipos de datos que usaremos
    - **utils:** contiene archivos con funciones auxiliares para programar la app
    - **styles:** muchos de los elementos de la app tienen un estilo común, así que aquí pondremos un archivo con estilos globales
    - **components:** aquí se programarán los componentes de la app
3. Instala las siguientes librerías:
    - Componente **Image** de Expo → **npx expo install expo-image**
    - Iconos **MaterialIcons** → **npx expo install @expo/vector-icons**
    - Librería **axios** → **npm install axios**
    - Librería **ramda** → **npm install ramda**

☞ ***¿Para qué sirve esta librería?*** *Cuando tenemos una lista en una variable de estado, React-Native no se entera si la lista muta (se le añaden o eliminan valores), con lo que no se producirán renderizados. Para que se entere, los setters necesitan recibir siempre objetos nuevos. O sea, cada vez que se actualice una lista necesitamos asignar a su variable de estado una lista nueva con la modificación realizada. La librería **ramda** permite realizar operaciones sobre listas (añadir, borrar, eliminar) devolviendo siempre listas nuevas, así que es ideal para nuestro objetivo*

- Librería **react-native-uuid** → **npm install react-native-uuid**

☞ ***¿Para qué sirve esta librería?*** *Esta app maneja datos de lugares del mundo, y cuando los creemos, deberemos ponerles un id único. La librería **react-native-uuid** permite generar automáticamente números de 128 bits (llamados **uuid** por las siglas **Universally Unique IDentifier**) que por su gran tamaño, se consideran únicos*[1]

4. Crea una rama llamada **tutorial18**

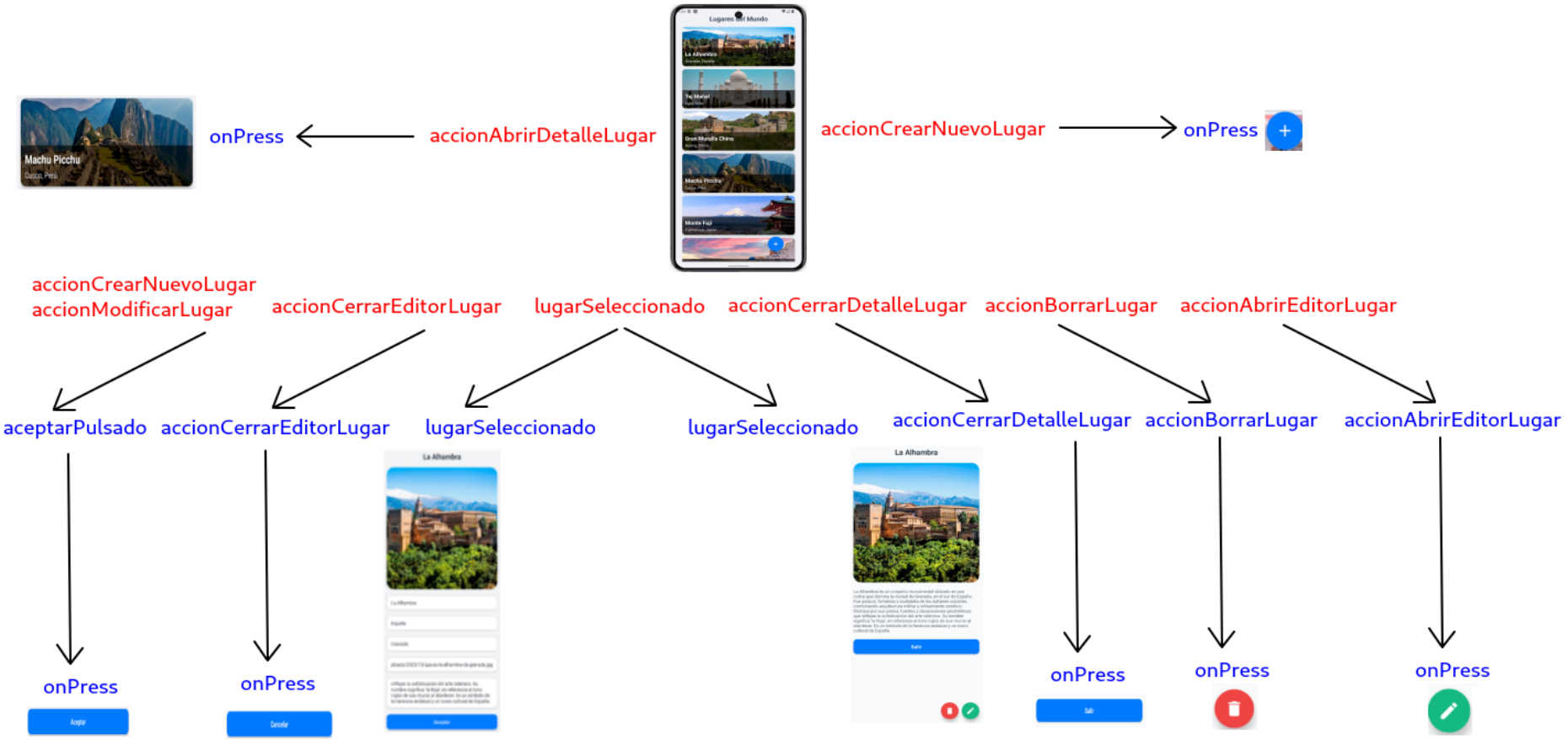
2.- El Unidirectional Data Flow

*Cuando tenemos una app como la de este tutorial, en la que van a intervenir bastante componentes y subcomponentes, es importante organizar todas las variables de entorno y funciones siguiendo un principio clave de React: el **unidirectional data flow**, que nos indica que los datos pueden pasar mediante **props** de un componente padre, pero no al revés.*

Esto significa que a la hora de hacer los componentes, tendremos que tener en cuenta qué datos y funciones recibirán de su componente padre, y del mismo modo, qué datos y funciones pasarán a sus componentes hijos

[1] Si se generan mil millones de uuids la probabilidad de encontrar dos repetidos es mucho menor que la de ganar la lotería cientos de veces seguidas

El siguiente gráfico muestra cómo se van a disponer las funciones y variables de estado (en rojo) de los componentes que intervendrán en la app y cómo pasan mediante props (en azul) a sus componentes hijos.



3.- Puesta en marcha de una Fake Api

En este proyecto usaremos una Fake API para simular un API que guarda los datos en una base de datos.

> *Una **Fake API** es un programa que pone en marcha un API que solo funciona en nuestro equipo (localhost) y lee y guarda los datos en un archivo JSON*

1. Copia en la carpeta del proyecto (donde **App.tsx**) el archivo **datos.json**
2. Ejecuta el siguiente comando para instalar el programa **json-server**

npm install json-server@0.17.4

3. Ejecuta el siguiente comando para poner en marcha **json-server** y crear una Fake API que trabaje sobre el archivo **datos.json**

npx json-server datos.json

> *El programa **json-server** crea un API local que tiene endpoints para consultar, insertar, modificar y borrar objetos del archivo JSON que se le pasa al lanzarlo.*

4. Vamos a comprobar que funciona. Abre el navegador, accede a la dirección **http://localhost:3000/lugares** y comprueba que se muestra un JSON.

4.- Definición del modelo de datos

El modelo de datos está formado por los tipos de datos que necesitaremos para programar la app, y que siempre incluye los conceptos del mundo real que aparecen en ella.

*En nuestro caso, la respuesta del api nos sugiere que necesitaremos un tipo llamado **Lugar,** del que nos interesa recoger su id, nombre, país, ciudad, foto y descripción.*

1. En **model** crea un archivo llamado **Tipos.ts.**
2. Añade a **Tipos.ts** el siguiente tipo **Lugar**

```
1.  export type Lugar = {
2.      id:string
3.      nombre:string
4.      pais:string
5.      ciudad:string
6.      foto:string
7.      descripcion:string
8.  }
```

3. Añade a **Tipos.ts** un tipo llamado **Lugares**, que sea un alias para el tipo de dato **Array<Lugar>**, y exporta ambos tipos.

```
1.  export type Lugares = Array<Lugar>
```

> *En **TypeScript** es posible asignar un nombre alternativo (llamado alias) a un tipo, para así referirnos a él de forma más simple. Como en nuestro proyecto se usará con frecuencia el tipo **Array<Lugar>**, es más cómodo llamarlo **Lugares***

4. Aunque no es imprescindible, facilita mucho tener un tipo de dato **DatosFormulario** que recoja los datos que se escriben en el formulario de alta/modificación de **Lugar.** Dichos datos son todos los de **Lugar**, menos el **id**

```
export type DatosFormulario = {
  nombre: string;
  pais: string;
  ciudad: string;
  foto: string;
  descripcion: string;
}
```

5. Para no repetir código (observa que **Lugar** y **DatosFormulario** poseen muchos campos comunes), modifica el tipo **Lugar** de forma que tenga los mismos campos que el tipo **DatosFormulario** y el **id**

```
export type Lugar = DatosFormulario & {
  id: string
}
```

*Cuando se define un tipo, el operador **&** sirve para "añadir" a un tipo los campos de otro[2].*

5.- Creación de los estilos globales

Vamos a crear algunos estilos que usaremos en varios lugares de la app.

1. En la carpeta **styles** crea un archivo llamado **GlobalStyles.ts**
2. En dicho archivo, crea y exporta un objeto **globalStyles** con estos estilos:

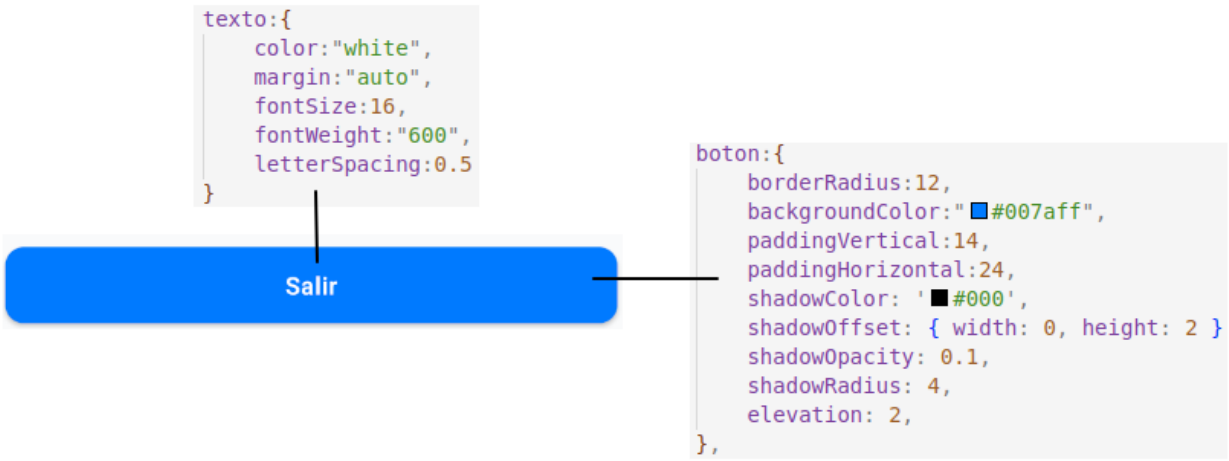
```
contenedor: {
    flex: 1,
    padding: 24,
  },
  titulo: {
    color: "#1F2937",
    textAlign: "center",
    fontSize: 24,
    fontWeight: "bold",
  },
  foto: {
    width: "100%",
    aspectRatio: 1,
    borderRadius: 25,
  },
  cuadroTexto:{
    backgroundColor:"#ffffff",
    borderWidth:1,
    borderColor:"#d1d5db",
    borderRadius:12,
    paddingVertical:12,
    paddingHorizontal:16,
    fontSize:16,
    color:"#111827",
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 1 },
    shadowOpacity: 0.05,
    shadowRadius: 4,
    elevation: 1,
}
```

[2] En realidad, **A & B** define un nuevo tipo, y un dato **x** pertenece a dicho tipo si pertenece simultáneamente al tipo **A** y al tipo **B**. Por ese motivo, el tipo **A & B** se denomina "intersection type"

6.- Creación del componente Boton

*En este proyecto se hará uso de un botón alargado de color azul en varios momentos, y por ese motivo, vamos a hacer un componente **Boton** de manera que sea reutilizable y pueda usarse en varios lugares del proyecto.*

1. En **components** crea un archivo llamado **Boton.tsx**
2. Abre **Boton.tsx** y teclea **rnfs+intro**
3. Sin mirar la solución, diseña el siguiente componente, teniendo en cuenta que:
    - El componente tendrá dos props:
        - **texto:** un **string** con el texto que se muestra en el botón
        - **onPress:** una función que se ejecuta al pulsar el botón
    - Al pulsar el botón, su color de fondo se pondrá a **#2563EB**



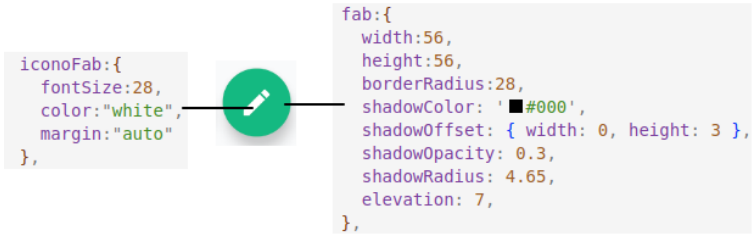

```tsx
type BotonProps = {
  texto: string;
  onPress: () => void;
}
export default function Boton({ texto, onPress }: BotonProps) {
  return (
   <Pressable
      style={ ({pressed}) =>
                    pressed ? [styles.boton, { backgroundColor: "#2563eb" }] : styles.boton }
      onPress={onPress} >
    <Text style={styles.texto}>{texto}</Text>
   </Pressable>
  )
}
```

7.- Creación del componente Fab

*También se usará un floating action button que aparecerá en varios lugares, y por ese motivo, vamos a crear un componente **Fab.tsx** que lo haga reutilizable*

1. En **components** crea un archivo llamado **Fab.tsx**
2. Abre **Fab.tsx** y teclea **rnfs+intro**
3. Sin mirar la solución, diseña el siguiente componente, teniendo en cuenta que
    - El componente tendrá tres props:
        - **icono:** Será un **string** con el nombre del icono que se mostrará
        - **onPress:** Función que se ejecuta al pulsar el botón
        - **bgColor:** Un **string** con el color de fondo del botón

- El icono será de la librería **MaterialIcons**




```
type FabProps={
    icono:string
    onPress: ()=>void
    bgColor:string
}
export default function Fab({icono, onPress,bgColor}:FabProps) {
  return (
    <Pressable style={[styles.fab,{backgroundColor:bgColor}]} onPress={onPress}>
        <MaterialIcons name={icono} style={styles.iconoFab}/>
    </Pressable>
  )
}
```

*Observa que el prop **name** del **MaterialIcons** da un error, porque **icono** es un **string** y podría contener un nombre de icono incorrecto.*

4. Modifica **FabProps** para que **icono** solo pueda tomar los valores **“edit”, “add”** y **“delete”**

```
type FabProps={
    icono: "edit"|"add"|"delete"
    onPress: ()=>void
    bgColor:string
}
```

*En **TypeScript** cuando escribimos valores separados por **|** estamos restringiendo el tipo de un campo a esos valores concretos.*

5. Cambia el tipo de **bgColor** para que en lugar de **string** sea **ColorValue.** Ese tipo admite **string** con formato de color (nombre en inglés, notación html, etc)

```
type FabProps={
    icono:"edit"|"add"|"delete"
    onPress: ()=>void
    bgColor:ColorValue
```

8.- Diseño del componente ItemLugar

*Nuestra app va a mostrar en su componente principal un **FlatList** rellenado con tarjetas para lugares del mundo famosos. Por ese motivo, el siguiente paso será diseñar un componente que sirva de tarjeta para mostrar un lugar.*

1. En **components** crea un archivo llamado **ItemLugar.tsx**
2. Abre **ItemLugar.tsx** y teclea **rnfs+intro**
3. Como ejercicio, diseña el siguiente componente sin mirar la solución.

- Tendrá un prop llamado **item**, que será un objeto de tipo **Lugar** cuyos datos se muestran en la tarjeta

- Se usa un **ImageBackground** (de la librería **expo-image**) con una imagen de fondo y dentro un contenedor que muestra en una columna semitransparente el nombre y la ciudad con el país del lugar.


```
1.  type ItemLugarProps = {
2.    item: Lugar
3.  }
4.  export default function ItemLugar({item}:ItemLugarProps) {
5.    return (
6.      <View style={styles.contenedor}>
7.        <ImageBackground source={item.foto} style={styles.imagen} contentFit={"cover"}>
8.          <View style={styles.columna}>
9.            <Text style={styles.titulo}>{item.nombre}</Text>
10.           <Text style={styles.subtitulo}>{item.ciudad}, {item.pais}</Text>
11.         </View>
12.       </ImageBackground>
13.     </View>
14.   );
15. }
```

*Puesto que esta tarjeta va a usarse en el **renderItem** de un **FlatList** para mostrar un objeto **Lugar**, vamos a añadir a **App** una función que reciba un **Lugar** y nos devuelva la correspondiente etiqueta **ItemLugar**. Hacer esto facilita el mantenimiento cuando el componente **ItemLugar** reciba más props (no solo **item**)*

4. Añade dentro de la función **App** una función llamada **getItemLugar** que reciba un objeto **Lugar** y nos devuelva una etiqueta **ItemLugar** para ese lugar

```
1.  export function getItemLugar(lugar:Lugar):React.ReactElement{
2.      return <ItemLugar item={lugar}/>
3.  }
```

*Los componentes que se pueden poner dentro de un **FlatList** son de tipo **React.ReactElement***

9.- Diseño del componente principal

*A continuación vamos a diseñar el componente principal de la app, que estará formado por una lista con tarjetas para mostrar todos los lugares que nos devuelve el api. Para ello, usaremos un **FlatList** y cada uno de sus ítems será un **ItemLugar***

1. Abre **App.tsx** y añade una variable de estado (con su setter) llamada **listaLugares**, que contendrá la lista de lugares que devuelva el api

```
export default function App() {
  const [listaLugares, setListaLugares] = useState<Lugares>([])
  // resto omitido
}
```

2. A continuación, añade las siguientes funciones vacías (de momento), que realizarán las funcionalidades de la app
   - **accionCargarLugares:** Pedirá al api todos los lugares disponibles y los guardará en **listaLugares**
   - **accionAbrirDetalleLugar:** Abre un modal que muestra los datos de un objeto **Lugar** que recibe como parámetro
   - **accionCerrarDetalleLugar:** Cierra el modal que muestra los detalles de un lugar
   - **accionAbrirEditorLugar:** Recibe un objeto **Lugar** y abre un modal con un formulario para editarlo. Si se pasa **undefined**, entonces el formulario creará un lugar nuevo con los datos rellenados en él.
   - **accionCerrarEditorLugar:** Cierra el modal de edición de lugar
   - **accionCrearNuevoLugar:** Crea un nuevo lugar con los datos rellenados en el formulario (que son recibidos como parámetro)
   - **accionModificarLugar:** Modifica el lugar cuyo **id** se pasa como parámetro, con los datos rellenados en el formulario
   - **accionBorrarLugar:** Borra el lugar cuyo **id** se pasa como parámetro

```
export default function App() {
  // inicio omitido
  function accionCargarLugares() {
    // aquí se pedirá al api cargar todos los lugares
  }
  function accionAbrirDetalleLugar(lugar: Lugar) {
    // abre un modal con una pantalla que muestra el detalle de un lugar
  }
  function accionCerrarDetalleLugar() {
    // se cierra el modal con los detalles del lugar
  }
  function accionAbrirEditorLugar(lugar?: Lugar) {
    // aquí se mostrará un modal con un formulario para crear o editar un lugar
  }
  function accionCerrarEditorLugar() {
    // se cierra el modal con el formulario de edición de un lugar
  }
  function accionCrearNuevoLugar(datos:DatosFormulario){
    // aquí se pedirá al api crear un lugar, con los datos del formulario
  }
  function accionModificarLugar(datos:DatosFormulario){
    // aquí se pedirá al api modificar un lugar, con los datos del formulario
  }
  function accionBorrarLugar(){
    // aquí se pedirá al api borrar el lugar seleccionado por el usuario
  }
}
```

3. En **App.tsx**, añade los siguientes estilos:
    - contenedor:
        - Las características del estilo global **contenedor**
        - color de fondo **#F0F2F5**
    - título:
        - Las características del estilo global **titulo**
        - Color **#344055**
        - Margen inferior **16**
    - posicionFab: Define una posición absoluta a **64** píxeles de la esquina inferior derecha

```
const styles = StyleSheet.create({
  contenedor: {
    ...globalStyles.contenedor,
    backgroundColor: '#f0f2f5',
  },
  titulo:{
    ...globalStyles.titulo,
    color:"#344055",
    marginBottom: 16
  },
  posicionFab:{
    position:"absolute",
    bottom:64,
    right:64
  }
});
```

4. En **App.tsx** añade:
    - un **Text** con el título “Lugares del Mundo” y el estilo **titulo**
    - un **FlatList** que muestre la lista almacenada en **listaLugares**. Se usará como clave primaria el **id** del lugar (pasado a **string**) y un **ItemLugar** devuelto por la función **getItemLugar** para mostrar cada uno de los datos de la lista
    - Un **Fab** con icono **add** y color de fondo **#007AFF** dentro de un contenedor con estilo **posicionFab**
    - Al pulsar el botón **Fab** se llamará a **accionMostrarEditorLugar** sin pasarle nada (con esto, el modal que se abra mostrará un formulario para crear un nuevo lugar)

```
export default function App() {
  // inicio omitido
  return (
    <View style={styles.contenedor}>
      <Text style={styles.titulo}>Lugares del Mundo</Text>
      <FlatList
        data={listaLugares}
        keyExtractor={(lugar) => lugar.id.toString()}
        renderItem={ ({item}) => getItemLugar(item)}
      />
    </View>
  );
}
```

- Añade en **App** un **Fab** con icono **“add”** y color de fondo **#007AFF** tras el **FlatList**. El **Fab** estará encerrado en un **View** con el estilo creado en el punto anterior y al pulsarlo, llamará a una lambda (o función) que ponga a **true** la variable **modalCrearVisible**

```
export default function App() {
  // inicio omitido
  return (
    <View style={styles.contenedor}>
      <Text style={styles.titulo}>Lugares del Mundo</Text>
      <FlatList
        data={listaLugares} keyExtractor={(lugar) => lugar.id.toString()}
        renderItem={({ item }) => getItemLugar(item)}
      />
      <View style={styles.posicionFab}>
        <Fab
          icono={"add"}
          bgColor={"#007aff"}
          onPress={() => accionAbrirEditorLugar()}
        />
      </View>
    </View>
  );
}
```

☞ ***¿Por qué es necesario encerrar el Fab en un View?*** *Se debe a que no hemos puesto un prop* ***style*** *al* ***Fab****, y por ese motivo, no admite estilos. Debemos posicionarlo encerrándolo en un* ***View***

5. Por último, añade a la función **App** una función **mostrarError** que reciba un mensaje y lo muestre en una ventana emergente con un título de error.

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

10.- Consulta de todos los lugares

Comenzaremos realizando la consulta de todos los lugares del mundo que hay en el api. El end point que usaremos tiene estos datos:

- ***url****: http://localhost:3000/lugares*
- ***método****: GET*

1. En **utils** crea un archivo llamado **CrudLugares.ts.** En él pondremos todas las funciones encargadas de hacer las operaciones CRUD sobre los lugares

Para facilitar el mantenimiento y la escalabilidad de la app es buena costumbre separar los accesos a datos en archivos y funciones diferentes del código de la interfaz de usuario. Por ese motivo, por cada operación CRUD tendremos una función en ***CrudLugares.ts*** *y la correspondiente función "acción" que la llama en* ***App****. Por ejemplo, para consultar los lugares tendremos:*

- ***cargarLugares:*** *Función en* ***CrudLugares.ts*** *que consulta el api*
- ***accionCargarLugares:*** *Función en* ***App.tsx*** *que llama a* ***cargarLugares***

2. Crea y exporta una función asíncrona llamada **cargarLugares**, que use **axios** para consultar el endpoint indicado anteriormente.

```
export async function cargarLugares():Promise<Lugares>{
    const url = "http://localhost:3000/lugares"
    const respuesta = await axios.get(url)
    return respuesta.data
}
```

*Como el api responde una lista de objetos que se ajustan perfectamente a nuestro tipo **Lugar**, podemos devolver directamente **respuesta.data***

3. Abre **App** y programa la función **accionCargarLugares**, que cargue la lista de lugares. Si todo va bien, la guardará en la variable de estado **listaLugares** y en caso de error, llamará a **mostrarError**

```
function accionCargarLugares(){
  cargarLugares()
    .then( lugares => setListaLugares(lugares) )
    .catch( error => mostrarError(error.toString()))
}
```

4. Usa **useEffect** para que la función **accionCargarLugares** se llame automáticamente cuando se inicie el componente **App**

```
export default function App() {
  const [listaLugares, setListaLugares] = useState<Lugares>([])
  useEffect( accionCargarLugares, [])
  // resto omitido
}
```

*Recuerda que **useEffect** sirve para llamar automáticamente a una función cuando cambian una o varias variables de estado que se pasan en una lista. Si dicha lista se deja vacía, entonces la función se llama solo una vez, cuando se inicia el componente en el que se encuentra.*

5. Ejecuta la app y comprueba que si estás en **Android** se mostrará un error, y si estás en **iOS** todo funcionará correctamente

☞ ***¿Por qué hay un error?*** *El motivo es que el servidor del api se encuentra en tu ordenador, que es un equipo diferente del móvil o el emulador. Por ese motivo, al escribir **localhost** en la dirección del api, el móvil (o el emulador) intentará conectarse a sí mismo y como no es el servidor, da ese error.*

En una app real habría que poner la ip o nombre de dominio del servidor. Sin embargo, en desarrollo es posible usar unas direcciones especiales para poder acceder al equipo local donde está el emulador:

- *Android → **10.0.2.2***
- *iOS → **localhost** (por eso, en **iOS** funciona sin tener que hacer nada)*

6. Abre **CrudLugares.ts** y crea al principio una constante llamada **IP** que guardará **10.0.2.2** si es Android o **localhost** en cualquier otro caso. Utiliza dicha variable dentro de la función **cargarLugares**

```
const IP = Platform.OS==="android" ? "10.0.2.2" : "localhost"
async function cargarLugares():Promise<Lugares>{
    const url = `http://${IP}:3000/lugares`
    const respuesta = await axios.get(url)
    return respuesta.data
}
```

Si ejecutas la app ahora, podrás ver que se muestra correctamente la lista de lugares que nos devuelve el api

11.- Activar la pulsación en una tarjeta

A continuación vamos a hacer que al pulsar en la tarjeta de un lugar, se abra un modal que nos muestre toda la información de dicho lugar.

*Para ello, deberemos modificar nuestro componente **ItemLugar** para hacer que sea pulsable y que admita como prop la función que se lanzará al pulsar en la tarjeta. Esta modificación nos llevará hacer cambios también en la función **getItemLugar***

1. Abre **ItemLugar.tsx** y modifica el tipo **ItemLugarProps** para que también contenga una función llamada **accionAbrirDetalleLugar.** Esta función recibirá un objeto **Lugar** y se encargará de abrir un modal con los datos de ese lugar.

```
type ItemLugarProps = {
  item: Lugar
  accionAbrirDetalleLugar: (lugar:Lugar) => void
}
export default function ItemLugar({item, accionAbrirDetalleLugar }:ItemLugarProps) {
  // resto omitido
}
```

*En **TypeScript** el tipo de dato que representa una función que recibe un objeto **Lugar** y no devuelve nada es **(lugar:Lugar)=>void.** El nombre del parámetro (**lugar**) es obligatorio, aunque no sirva para nada. Se puede pescribir un guión bajo (_) en lugar del nombre del parámetro, aunque queda menos legible*

```
type ItemLugarProps = {
  item: Lugar
  accionAbrirDetalleLugar: (_:Lugar) => void
}
```

2. Cambia el **View** contenedor de **ItemLugar** por un **Pressable** que al ser pulsado, lance la función **accionAbrirDetalleLugar** pasándole como parámetro el objeto **item** (que es de tipo **Lugar**) que se está mostrando en el componente

```
export default function ItemLugar({item, accionAbrirDetalleLugar }:ItemLugarProps) {
  return (
    <Pressable style={styles.contenedor} onPress={() => accionAbrirDetalleLugar (item)}>
      <ImageBackground source={item.foto} style={styles.imagen} contentFit={"cover"}>
        <View style={styles.columna}>
          <Text style={styles.titulo}>{item.nombre}</Text>
          <Text style={styles.subtitulo}>{item.ciudad}, {item.pais}</Text>
        </View>
      </ImageBackground>
    </Pressable>
  );
}
```

3. En **App** modifica la función **getItemLugar** para pasarle el prop **accionAbrirDetalleLugar** que acabamos de añadir a **ItemLugar**

```
function getItemLugar(lugar:Lugar):React.ReactElement{
 return <ItemLugar item={lugar} accionAbrirDetalleLugar ={ accionAbrirDetalleLugar }/>
}
```

4. Para comprobar que funciona, añade un **console.log** con un mensaje informativo en **accionAbrirDetalleLugar** y ejecuta la app. Si lo has hecho bien, al pulsar una tarjeta deberá mostrarse dicho mensaje.

12.- Diseño del componente DetalleLugar

*A continuación diseñaremos un componente llamado **DetalleLugar,** que será la pantalla donde mostraremos el detalle del lugar seleccionado por el usuario al pinchar la tarjeta de un lugar.*

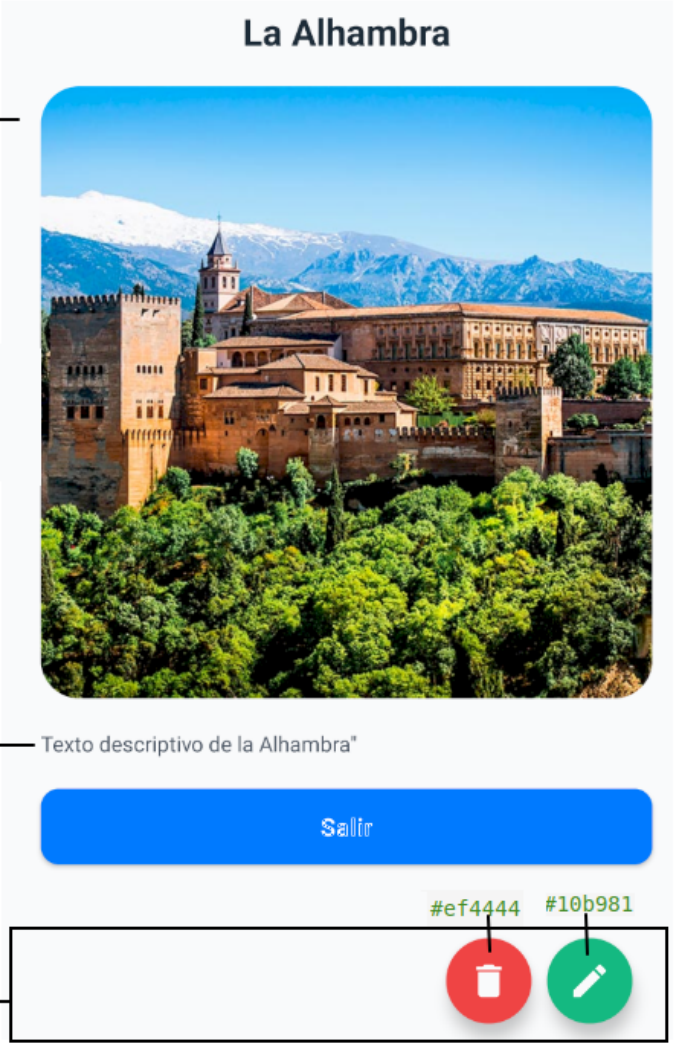
1. En **components** crea un archivo llamado **DetalleLugar.tsx**
2. Abre **DetalleLugar.tsx** y teclea **rnfs+intro**
3. Añade
4. Sin mirar la solución, crea el siguiente diseño, teniendo en cuenta:

    - Los props del componente son:

        - **lugarSeleccionado:** Es un objeto **Lugar** con el lugar seleccionado por el usuario para ver su detalle
        - **salirPulsado:** Es la función que se ejecutará cuando se pulse el botón de salir.
        - **accionAbrirEditorLugar:** Es la función que se ejecutará cuando se pulse el botón de editar el lugar
        - **accionBorrarLugar:** Es la función que se ejecutará cuando se pulse el botón de borrar el lugar.

    - La imagen tiene el estilo global **foto**
    - Haz que al pulsar los botones se llamen a las funciones recibidas en los props, con los parámetros adecuados.

```
contenedor:{
  ...globalStyles.contenedor,
    backgroundColor:"#f9fafb",
    rowGap:20
},
```




```
descripcion:{
    color:"#4b5563"
},
```

```
fila:{
  flex:1,
  flexDirection:"row",
  alignSelf:"stretch",
  justifyContent:"flex-end",
  alignItems:"flex-end",
  columnGap:10,
},
```

```
type DetalleLugarProps = {
  lugarSeleccionado: Lugar;
  salirPulsado: () => void;
  accionEditorLugar: () => void;
  accionBorrarLugar: () => void;
}
export default function DetalleLugar({lugarSeleccionado,salirPulsado,
  accionAbrirEditorLugar,accionBorrarLugar}:DetalleLugarProps) {
  // inicio omitido
  return (
    <View style={styles.contenedor}>
      <Text style={globalStyles.titulo}>{lugarSeleccionado.nombre}</Text>
      <Image
        source={lugarSeleccionado.foto}
        contentFit={"cover"}
        style={globalStyles.foto}
      />
      <Text style={styles.descripcion}>{lugarSeleccionado.descripcion}</Text>
      <Boton texto={"Salir"} onPress={salirPulsado}/>
      <View style={styles.fila}>
        <Fab
            icono={"delete"} bgColor={"#ef4444"}
            onPress={ accionBorrarLugar}/>
        <Fab
            icono={"edit"} bgColor={"#10b981"}
            onPress={ accionAbrirEditorLugar}/>
      </View>
    </View>
  );
}
```

13.- Mostrar el detalle del lugar seleccionado en un modal

Ahora vamos a hacer que la pulsación en una tarjeta abra un modal con el detalle de ese lugar. Para ello, comenzaremos abriendo un modal en la función ***accionAbrirDetalleLugar*** *y mostraremos en él el componente* ***DetalleLugar***

1. Añade a **App** esta variables de estado, con su setter:
   - **lugarSeleccionado:** su tipo será **Lugar | undefined** y será el lugar que el usuario pulsa para ver su detalle. Inicialmente será **undefined** porque el usuario no ha pulsado en ningún lugar.

```
export default function App() {
  const [lugarSeleccionado,setLugarSeleccionado] = useState<Lugar | undefined>(undefined)
  // resto omitido
}
```

> *Como en* ***TypeScript*** *los tipos no admiten* ***undefined*** *ni* ***null,*** *es necesario indicar a* ***useState*** *que el lugar seleccionado puede ser* ***undefined*** *(también se haría igual se se usara* ***null****) poniendo como tipo* ***Lugar | undefined***
>
> *Usamos aquí* ***undefined*** *en lugar de* ***null*** *porque las funciones que reciben valores opcionales usan* ***undefined*** *y así unificamos en* ***undefined*** *todos esos "tipos nulos"*

2. Modifica la función **accionAbrirDetalleLugar** para que ponga **lugarSeleccionado** con el objeto **Lugar** que recibe como parámetro.

```
function accionAbrirDetalleLugar(lugar: Lugar) {
  setLugarSeleccionado(lugar)
}
```

3. Modifica la función **accionCerrarDetalleLugar** para que ponga a **undefined** el lugar seleccionado

```
function accionCerrarDetalleLugar() {
  setLugarSeleccionado(undefined)
}
```

4. Al final de **App** añade un bloque de renderizado condicional que abra un modal en caso de que **lugarSeleccionado** no sea **undefined**. Dicho modal tapará la pantalla principal y aparecerá desde abajo deslizándose

```
export default function App() {
 // resto omitido
 return (
     // inicio omitido
      {
        lugarSeleccionado !== undefined && (
          <Modal transparent={false} animationType={"slide"}>
          </Modal>
        )
      }
    </View>
  );
}
```

5. Añade dentro del modal un componente **DetalleLugar,** rellenando sus props con los valores que ya tenemos en **App.tsx:**

```
export default function App() {
  // inicio omitido
  return (
      // inicio omitido
      {
        lugarSeleccionado!==undefined && (
          <Modal transparent={false} animationType={"slide"}>
            <DetalleLugar
              lugarSeleccionado={lugarSeleccionado}
              accionAbrirEditorLugar={accionAbrirEditorLugar}
              accionBorrarLugar={ accionBorrarLugar}
              salirPulsado={ accionCerrarDetalleLugar }/>
          </Modal>
        )
      }
    </View>
  );
}
```

6. Ejecuta la app y comprueba que ya puedes ver el detalle de cualquier lugar que selecciones, y además puedes cerrar del modal pulsando el botón de salir.

14.- Creación de un componente común para crear y editar un lugar

Nuestra app tiene dos operaciones que tienen mucho que ver: añadir un nuevo lugar y modificar datos de un lugar ya existente.

Ambas operaciones presentarán al usuario un formulario para la toma de datos de un lugar. Una lo hará mostrando un formulario en blanco, y la otra mostrando ese mismo formulario, pero con datos ya rellenados (los del lugar que vaya a ser editado)

*Por ese motivo, en este apartado vamos a diseñar un componente **EditorLugar** que nos sirva para ambos propósitos. Esto se consigue haciendo que ese componente se limite a pedir y validar los datos de un lugar (nombre, país, imagen y descripción), delegando en funciones pasadas como props la funcionalidad de los botones.*

1. En **components** crea un archivo llamado **EditorLugar.tsx**
2. Abre **EditorLugar.tsx** y teclea **rnfs+intro**
3. Añade a **EditorLugar** estos props:

    - **lugarSeleccionado:** Es un objeto opcional (puede no pasarse y valdrá **undefined** en ese caso) con el **Lugar** que se va a editar
    - **aceptarPulsado:** Es una función que recibe los datos escritos en el formulario y los procesa cuando se pulsa aceptar
    - **accionCerrarEditorLugar:** Es una función que cierra el formulario y se lanza cuando se pulse el botón de cancelar

```
type EditorLugarProps = {
  lugarSeleccionado?: Lugar;
  aceptarPulsado: (datos: DatosFormulario) => void;
  accionCerrarEditorLugar: () => void;
};
export default function EditorLugar({
  lugarSeleccionado,
  aceptarPulsado,
  accionCerrarEditorLugar,
}: EditorLugarProps) {
  // resto omitido
}
```

4. Añade a **EditorLugar** una variable de estado (con su setter) llamada **nombre** y dale como valor inicial el campo **nombre** de **lugar**, pero si **lugar** es **undefined**, le darás una cadena vacía **“”.** Aunque esto se puede hacer con la asignación condicional, usaremos estos operadores:
    - **?. → optional chaining**
    - **?? → nullish coalescing**

```
export default function EditorLugar({lugar,aceptarPulsado,accionCerrarEditorLugar,
}: EditorLugarProps) {
  const [nombre, setNombre] = useState(lugarSeleccionado?.nombre ?? "");
  // resto omitido
}
```

☞ ***¿Qué son los operadores “optional chaining” y “nullish coalescing”?*** *Como el prop* ***lugarSeleccionado*** *puede tomar el valor* ***undefined****, por seguridad* ***TypeScript*** *no permite acceder a sus campos a menos que se compruebe previamente que no es* ***undefined*** *(esto se hace para evitar los errores* ***NullPointerException****)*

El operador ***?.*** *es una de las formas que* ***TypeScript*** *ofrece para hacer esa comprobación y funciona de esta forma:*

- *si* ***lugar*** *no es* ***undefined*** *(o* ***null****), se accede al campo con normalidad*
- *en caso contrario, entonces no se accede al campo (evitando el error) y el resultado de la operación es* ***undefined***

Operador ?.	Si A no es undefined o null	Si A es undefined o null
A?.campo	A.campo	undefined

Por su parte, el operador ***??*** *permite asignar un valor en caso de que el dato sobre el que se aplica sea* ***undefined*** *o* ***null****.*

Operador ??	Si A no es undefined o null	Si A es undefined o null
A ?? valor	A	valor

Es muy frecuente que ambos operadores aparezcan combinados, como ocurre en nuestro caso:

Operadores combinados	Si A no es undefined o null	Si A es undefined o null
A?.campo ?? valor	A.campo	valor

En cualquier caso, el efecto combinado de estos operadores equivale a esta asignación condicional:

(A === null || A === undefined) ? valor : A.campo

5. Repite lo anterior para añadir estas otras variables de estado (con su setter)
    - **pais:** El país del lugar
    - **ciudad:** La ciudad del lugar
    - **foto:** Una url con la foto del lugar
    - **descripcion:** La descripción del lugar

```
export default function EditorLugar({
  lugarSeleccionado,
  aceptarPulsado,
  accionCerrarEditorLugar,
}: EditorLugarProps) {
  const [nombre, setNombre] = useState(lugarSeleccionado?.nombre ?? "");
  const [pais, setPais] = useState(lugarSeleccionado?.pais ?? "");
  const [ciudad, setCiudad] = useState(lugar lugarSeleccionado ciudad ?? "");
  const [foto, setFoto] = useState(lugarSeleccionado?.foto ?? "");
  const [descripcion, setDescripcion] = useState(lugarSeleccionado?.descripcion ?? "");
  // resto omitido
}
```

☞ ***¿Por qué tenemos que añadir todas las variables por separado y no consideramos un solo objeto de tipo DatosFormulario?*** *Se podría hacer, pero el problema es que al modificarlo (mutarlo), React-Native no se enteraría de los cambios y no se harían renderizados. Habría que trabajar con ese objeto de forma inmutable (creando un nuevo objeto tras cada modificación) y sería engorroso, o bien utilizar librerías que faciliten la programación orientada a objetos, como* ***MobX,*** *o la gestión de estados complejos, como* ***Redux***

6. Añade a **EditorLugar** una función llamada **getDatosFormulario** que devuelva un objeto **DatosFormulario** que contenga todas las variables de la interfaz

```
function getDatosFormulario():DatosFormulario{
    return {nombre,pais,ciudad,foto,descripcion}
}
```

Cuando se ponen variables en un objeto, se utiliza el nombre de las variables como clave, y el valor de las variables como valor. O sea, lo que hemos escrito es equivalente a esto (mucho más largo):

```
function getDatosFormulario():DatosFormulario {
    return {
        nombre:nombre,
        pais:pais,
        ciudad:ciudad,
        foto:foto,
        descripcion:descripcion
    }
}
```

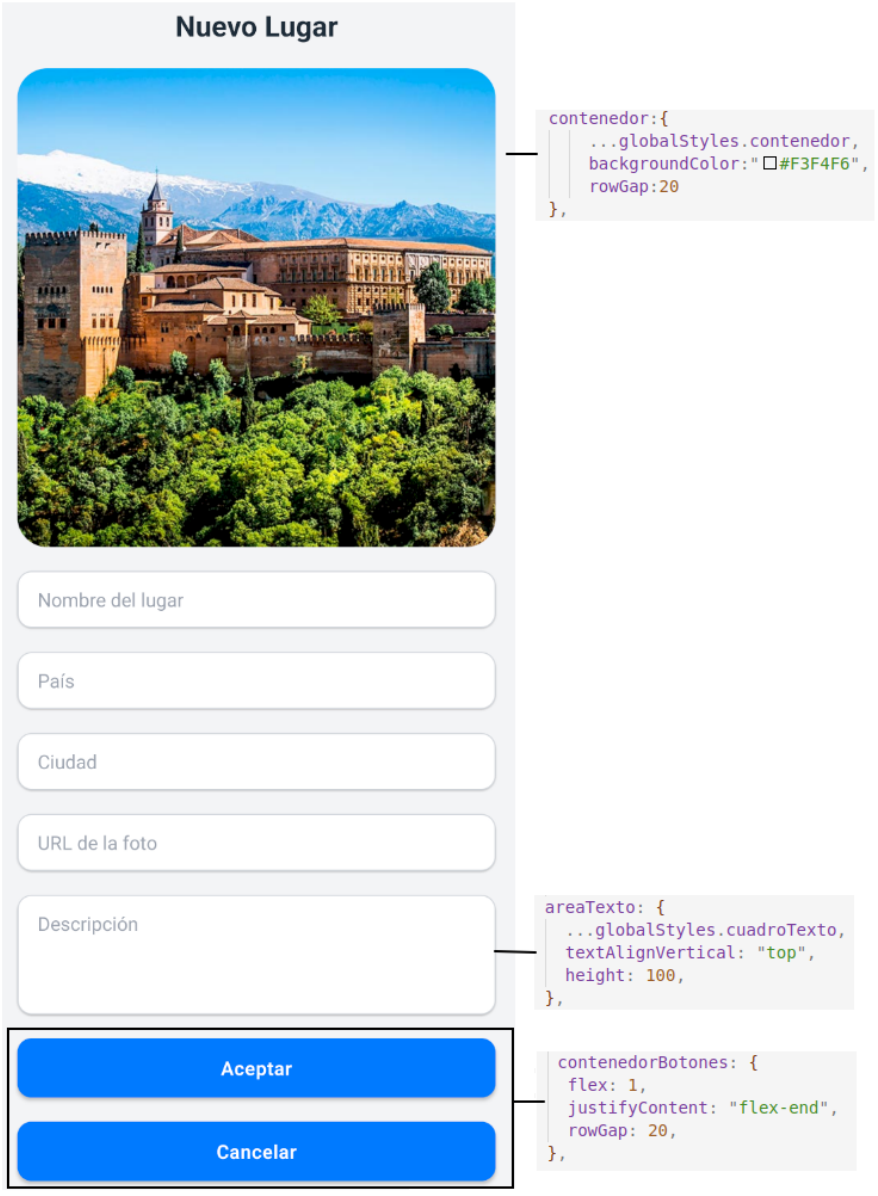
7. Antes del return de la función **EditorLugar**, crea una variable llamada **titulo** que valga **"Nuevo Lugar"** si **lugar** es **undefined**, y en caso contrario, valga el nombre del lugar

```
export default function EditorLugar({lugar,aceptarPulsado,accionCerrarEditorLugar,
}: EditorLugarProps) {
  // inicio omitido
  const titulo = lugar === undefined ? "Nuevo Lugar" : lugarSeleccionado.nombre
  return (
   // resto omitido
}
```

8. Sin mirar la solución, programa el siguiente diseño teniendo en cuenta que:
   - Todo el formulario estará dentro de un **ScrollView** que tendrá un prop llamado **contentContainerStyle={{flexGrow:1}}**

☞ ***¿Para qué hace falta ese prop?*** *Es necesario para que el* ***ScrollView*** *suba cuando aparece en pantalla el teclado para rellenar un cuadro de texto y de esa forma, el teclado no tape el cuadro de texto donde escribimos.*

- El texto del título será la variable **titulo**
- Los cuadros de texto usan el estilo global **cuadroTexto**
- La imagen usa el estilo global **foto**
- Vincula todos los cuadros de texto con sus respectivas variables de estado
- Haz que al pulsar el botón de aceptar, se llame a la función **aceptarPulsado** pasándole el objeto con todas las variables de estado devuelto por **getDatosFormulario**
- Haz que al pulsar el botón de cancelar, se llame a **accionCerrarEditorLugar**



```
export default function EditorLugar({
  lugarSeleccionado,
  aceptarPulsado,
  accionCerrarEditorLugar,
}: EditorLugarProps) {
  // inicio omitido
  return (
    <ScrollView contentContainerStyle={{ flexGrow: 1 }}>
      <View style={styles.contenedor}>
        <Text style={globalStyles.titulo}>{titulo}</Text>
        <Image source={foto} contentFit="cover" style={globalStyles.foto} />
        <TextInput
          value={nombre}
          onChangeText={setNombre}
          style={globalStyles.cuadroTexto}
          placeholder="Nombre del lugar"
          placeholderTextColor="#9ca3af"
        />
        <TextInput
          value={pais}
          onChangeText={setPais}
          style={globalStyles.cuadroTexto}
          placeholder="País"
          placeholderTextColor="#9ca3af"
        />
        <TextInput
          value={ciudad}
          onChangeText={setCiudad}
          style={globalStyles.cuadroTexto}
          placeholder="Ciudad"
          placeholderTextColor="#9ca3af"
        />
        <TextInput
          value={foto}
          onChangeText={setFoto}
          style={globalStyles.cuadroTexto}
          placeholder="URL de la foto"
          placeholderTextColor="#9ca3af"
        />
        <TextInput
          value={descripcion}
          onChangeText={setDescripcion}
          style={styles.areaTexto}
          placeholder="Descripción"
          placeholderTextColor="#9ca3af"
          numberOfLines={5}
          multiline={true}
        />
         <View style={styles.contenedorBotones}>
          <Boton
            texto={"Aceptar"}
            onPress={() => aceptarPulsado(getDatosFormulario())}
          />
          <Boton texto={"Cancelar"} onPress={accionCerrarEditorLugar} />
        </View>
      </View>
    </ScrollView>
  );
}
```

15.- Creación de un nuevo lugar

A continuación vamos a permitir que la app pueda añadir nuevos lugares. Para ello comenzaremos creando una función auxiliar que reciba los datos del formulario y solicite al api su grabación. Si no hay error, se creará un objeto ***Lugar*** *que se añadirá a la variable* ***listaLugares.***

Los datos del endpoint para grabar un lugar son:

- ***url*** → *http://localhost:3000/lugares*
- ***método*** → *POST*
- ***Parámetros:*** *Un objeto* ***Lugar*** *con el lugar que se va a registrar*

Cuando un api usa el método POST, se le pueden pasar objetos en formato json como parámetros. Dichos objetos no se ven en la url (como ocurre con los parámetros en el método GET), porque viajan en las cabeceras del protocolo http

1. Abre el archivo **CrudLugares.tsx** y añade una función asíncrona llamada **crearNuevoLugar**, que recibe un objeto **DatosFormulario** y usa **axios** para enviarlo al endpoint descrito anteriormente

```
1.  export async function crearLugar(datos:DatosFormulario):Promise<Lugar>{
2.  }
```

*Para crear un lugar, necesitamos obtener su **id** único. Vamos a utilizar la librería **react-native-uuid** para generar un identificador aleatorio único (**uuid**) a cada lugar.*

2. Crea una constante de tipo **Lugar** con todos los datos recibidos en el formulario y utiliza la función **v4** (de **react-native-uuid)** para poner el **id**

```
1.  export async function crearNuevoLugar(datos:DatosFormulario){
2.    const lugar:Lugar = {
3.      id:v4(),
4.      nombre:datos.nombre,
5.      pais:datos.pais,
6.      ciudad:datos.ciudad,
7.      foto:datos.foto,
8.      descripcion:datos.descripcion
9.    }
10. }
```

*La función **uuid.v4** nos genera un identificador único para el lugar creado. Aunque en la librería aparecen otras funciones que "parecen versiones más recientes" como **v5** o **v7**, hay que tener en cuenta que no es así, y que tienen finalidades distintas. Por ejemplo, **v5** no genera un uuid aleatorio, sino que funciona de manera más parecida a un "resumen hash" que da un identificador único para datos únicos*

3. Usa **axios** para enviar dicho objeto al endpoint con el método **POST**, y termina la función devolviendo dicho objeto.

```
1.  export async function crearNuevoLugar(datos:DatosFormulario):Promise<Lugar>{
2.      const lugar:Lugar={
3.          id:v4(),
4.          nombre:datos.nombre,
5.          pais:datos.pais,
6.          ciudad:datos.ciudad,
7.          foto:datos.foto,
8.          descripcion:datos.descripcion
9.      }
10.     const URL = `http://${IP}:3000/lugares`
11.     await axios.post(URL,lugar)
12.     return lugar
13. }
```

*Observa cómo con el método **post** podemos pasar objetos al api*

4. Añade a **App** una variable de estado llamada **modalEditarVisible** con valor incial **false** y su setter **setModalEditarVisible**. Esta variable controlará si se mostrará un modal con un **EditorLugar** para crear un nuevo lugar.

```
export default function App() {
  const [modalEditarVisible, setModalEditarVisible] = useState(false)
  // resto omitido
}
```

5. Modifica las funciones **accionAbrirEditorLugar** y **accionCerrarEditorLugar** para poner a **true** y **false** respectivamente la variable **modalCrearVisible**

```
function accionAbrirEditorLugar() {
  setModalEditarVisible(true)
}
function accionCerrarEditorLugar() {
  setModalEditarVisible(false)
}
```

6. Añade a **App** un segundo bloque de renderizado condicional que abra un modal cuando **modalEditarVisible** sea **true.** Dentro de ese modal se mostrará un **EditorLugar,** con sus props rellenados con las funciones que ya tenemos definidas en **App**

```
export default function App() {
  return (
    <View style={styles.contenedor}>
      // inicio omitido
      {
        modalEditarVisible && (
          <Modal transparent={false} animationType={"slide"}>
            <EditorLugar
              lugar = {lugarSeleccionado }
              aceptarPulsado={ accionCrearNuevoLugar}
              accionCerrarEditorLugar={ accionCerrarEditorLugar }/>
          </Modal>
        )
      }
    </View>
  )
}
```

*Observa que como **lugarSeleccionado** es en este momento **undefined**, el **EditorLugar** mostrará un formulario con campos vacíos para crear un nuevo lugar*

7. Por último, programa la función **accionCrearNuevoLugar** para crear el nuevo lugar usando la función asíncrona **crearNuevoLugar**. Si no hay errores:
    - Se cerrará el modal con el formulario de creación de nuevo lugar
    - Se usará la librería **ramda** para actualizar la variable de estado **listaLugares** con una nueva lista (no se mutará la lista que ya hay) que contenga el lugar añadido

```
function accionCrearNuevoLugar(datos:DatosFormulario){
  crearNuevoLugar(datos)
    .then( nuevoLugar => {
      setModalCrearVisible(false)
      const nuevaLista = R.append(nuevoLugar, listaLugares)
      setNuevaLista(nuevaLista)
    })
    .catch(error => mostrarError(error.toString()))
}
```

☞ ***¿Es necesario usar la librería ramda?*** *Como ya sabemos, el cambio en la lista no se detecta automáticamente y el setter necesita recibir una lista nueva con el lugar que hemos añadido. Tenemos dos formas de conseguirla:*

- *Usando la librería **ramda:** La función **R.append** recibe un objeto y una lista, y nos devuelve una lista nueva en el que el objeto ha sido añadido.*

```
1.  const nuevaLista = R.append(lugar, listaLugares)
2.  setListaLugares(nuevaLista)
```

- *Con **desestructuración**: Consiste en usar **…listaLugares** para "pegar" todos los elementos de la lista original en una nueva lista, y añadir después el elemento nuevo.*

```
1.  const nuevaLista = [...listaLugares, lugar]
2.  setListaLugares(nuevaLista)
```

8. Ejecuta la app y comprueba que puedes añadir nuevos lugares

16.- Modificar un lugar existente

*Vamos ahora a realizar la modificación de un lugar. Para ello vamos a aprovecharnos del componente **EditorLugar** que ya tenemos hecho y lo que haremos será pasarle el objeto **lugarSeleccionado** a su prop **lugar** cuando pulsemos editar.*

*Al igual que antes, comenzaremos creando una función auxiliar en **CrudLugares.ts** llamada **modificarLugar** El endpoint de nuestro api para modificar un lugar es:*

- ***url:** http://localhost:3000/lugares/**{id del lugar}***
- ***método:** PUT (para reemplazar completamente el objeto) o PATCH (para reemplazar solo algunos campos)*
- ***parámetros:** El objeto **Lugar** que va a ser modificado*

1. Abre **CrudLugares.ts** y crea una función asíncrona llamada **modificarLugar,** que reciba el **id** del lugar que se va a modificar y un objeto con datos del formulario, y lo pase al endpoint descrito. Se devolverá el objeto **Lugar** con las modificaciones realizadas.

```
1.  export async function modificarLugar(idLugarModificado:string,datos:DatosFormulario):Promise<Lugar>{
2.  }
```

2. Crea en dicha función un objeto de tipo **Lugar** cuyo **id** sea el del lugar que queremos modificar y sus datos sean los del formulario

```
1.  export async function modificarLugar(idLugarModificado:string,datos:DatosFormulario):Promise<Lugar>{
2.      const lugar:Lugar={
3.          id:idLugarModificado,
4.          nombre:datos.nombre,
5.          pais:datos.pais,
6.          ciudad:datos.ciudad,
7.          foto:datos.foto,
8.          descripcion:datos.descripcion
9.      }
10. }
```

3. Finaliza la función llamando al endpoint y pasándole ese objeto. La función devolverá dicho objeto **Lugar**

```
export async function modificarLugar(idLugarModificado:string,datos:DatosFormulario):Promise<Lugar>{
    const lugar:Lugar={
        id:idLugarModificado,
        nombre:datos.nombre,
        pais:datos.pais,
        ciudad:datos.ciudad,
        foto:datos.foto,
        descripcion:datos.descripcion
    }
    const url=`http://${IP}:3000/lugares/${lugar.id}`
    await axios.put(url,lugar)
    return lugar
}
```

4. En **App** programa la función **accionModificarLugar** de esta forma:

    - Se comprueba si **lugarSeleccionado** no es **undefined** (de ser así, no se podría consultar su **id** con seguridad)
    - Después, se llama a la función **modificarLugar** para que el api se encargue de actualizar el lugar
    - Si todo va bien:
        - Hace que **lugarSeleccionado** sea el objeto **lugar**, para que así la pantalla de detalle vea la actualización.
        - Se usa el método **map** de **listaLugares** para obtener una lista nueva con el lugar actualizado, y se pasa a **setListaLugares**
        - Se pone a **false** la variable **modalEditarVisible** para cerrar el modal de edición de lugar

```
function accionModificarLugar(datos:DatosFormulario){
  if(lugarSeleccionado!==undefined){
   modificarLugar(lugarSeleccionado.id, datos)
    .then( lugarModificado => {
      setLugarSeleccionado(lugarModificado)
      const nuevaLista = listaLugares.map(
         lugar => lugar.id===lugarModificado.id? lugarModificado : lugar
      )
      setListaLugares(nuevaLista)
      setModalEditarVisible(false)
    })
    .catch(error => mostrarError(error.toString()))
  }
}
```

> ☞ ***¿Se podría hacer con la librería ramda?*** *Si, es posible, pero vamos a ver que es un poco más difícil. Para obtener una nueva lista en la que un elemento de una lista original ha cambiado, hay dos posibilidades:*
>
> - *Usar la librería **ramda:** Con la función **R.findIndex** se encuentra la posición del elemento cuyo **id** es **lugar.id** (eso lo hace **R.propEq(lugar.id,"id")** en la lista de lugares. Después, la función **R.update** devuelve una nueva lista en la que el objeto de la posición indicada es reemplazado por otro.*

```
const posicion = R.findIndex(R.propEq(lugar.id,"id"), listaLugares)
const nuevaLista = R.update(posicion, lugar, listaLugares)
setListaLugares(nuevaLista)
```

- *El método **map** de **listaLugares**: Consiste en transformar **listaLugares** en una nueva lista en la que cada elemento se queda como está, excepto el que tiene el id del lugar modificado, que es reemplazado por **lugar***

```
const nuevaLista = listaLugares.map( l => l.id===lugar.id? lugar : l)
setListaLugares(nuevaLista)
```

4. En **App** añade una variable de entorno llamada **modalEditarVisible** con valor inicial **false** y su setter. Esta variable controlará si se mostrará un modal con los datos para editar el lugar guardado en **lugarSeleccionado**

```
export default function App() {
  const [modalEditarVisible, setModalEditarVisible] = useState(false)
  // resto omitido
}
```

5. En **App** modifica el último bloque de renderizado condicional de forma que el prop **aceptarPulsado** sea **accionCrearNuevoLugar** si **lugarSeleccionado** es **undefined**, y **accionModificarNuevoLugar** en caso contrario

```
export default function App() {
  // inicio omitido
  return (
    // inicio omitido
      {
        modalEditarVisible && (
          <Modal transparent={false} animationType={"slide"}>
            <EditorLugar
              lugar={lugarSeleccionado}
              aceptarPulsado={lugarSeleccionado===undefined ?
                                                accionCrearNuevoLugar : accionModificarLugar}
              accionCerrarEditorLugar={accionCerrarEditorLugar}
              />
          </Modal>
        )
      }
    </View>
  );
}
```

6. Ejecuta la app y comprueba que se pueden modificar los lugares

17.- Borrar el lugar seleccionado

*Para terminar, vamos a darle funcionalidad al botón de borrado. Comenzaremos creando en **CrudLugares.ts** una función asíncrona que borre el lugar, y después la llamaremos al pulsar el correspondiente botón, no sin antes, advertir al usuario con una ventana emergente.*

Los datos del endpoint que nuestro api proporciona para borrar un lugar son:

- ***url:** http://localhost:3000/lugares/**{id del lugar}***
- ***método:** DELETE*

1. Abre **CrudLugares.ts** y añade una función asíncrona que reciba un objeto **Lugar** y lance una petición al api para borrarlo

```
async function borrarLugar(lugar:Lugar){
    const url = `http://${IP}:3000/lugares/${lugar.id}`
    await axios.delete(url)
}
```

*El método **delete** del protocolo http normalmente se utiliza en las apis para solicitar al servidor el borrado de una entidad. No todas las api lo admiten, y otras pueden usar otros métodos (como get o post) para realizar el borrado. Por ese motivo, siempre hay que consultar bien la documentación del api.*

2. Dentro de la función **App**, programa la función **accionBorrarLugar** para que muestre una ventana emergente de confirmación, y en caso de aceptar el borrado, se llame a una función llamada **realizarBorrado** que ahora mismo se dejará vacía.

```
function accionBorrarLugar(){
  Alert.alert(
    `¿Desea borrar ${lugarSeleccionado?.nombre}`,
    "Un lugar eliminado no podrá ser recuperado",
    [
      {text:"Si, eliminar", onPress:realizarBorrado},
      {text:"No, cancelar"}
    ]
  )
}
function realizarBorrado(){
}
```

3. Programa la función **realizarBorrado** de esta forma:
   - Comprueba que **lugarSeleccionado** no sea **undefined** (si lo fuese, no podríamos pasarlo a **realizarBorrado**, que solo recibe un **Lugar**)
   - Después llama a la función asíncrona **borrarLugar**, y en caso de que no se produzca un error, usamos la función **without** de la librería **ramda** para <u>eliminar</u> el objeto **Lugar** de **listaLugares** creando una nueva lista (o sea, no se muta la variable **listaLugares** quitándole el lugar, sino que se crea una lista nueva sin el lugar eliminado) y finalmente ponemos a **null** la variable **lugarSeleccionado** (esto lo hacemos para cerrar el modal)

```
function realizarBorrado(){
   if(lugarSeleccionado!==undefined){
     borrarLugar(lugarSeleccionado)
       .then( () => {
         const nuevaLista = R.without([lugarSeleccionado],listaLugares)
         setListaLugares(nuevaLista)
         setLugarSeleccionado(undefined)
       })
       .catch(error => mostrarError(error.toString()))
   }
 }
```

☞ ***¿Es necesario usar la librería ramda?*** *Como ya sabemos, React-Native no se entera si una lista muta (se le añaden o eliminan valores), con lo que no se producirían renderizados. Para que se entere, los setters necesitan siempre objetos nuevos. En nuestro caso, necesitamos una <u>lista nueva</u> sin el valor que necesitamos, y tenemos dos formas de conseguirla:*

- *Usando la librería **ramda:** Es una librería que realiza operaciones sobre listas (añadir, borrar, eliminar) devolviendo siempre listas nuevas, así que es ideal para nuestro objetivo. La función **R.without** recibe una lista con los valores que se desean eliminar de una lista, y nos devuelve una lista nueva sin ellos.*

```
3.  const nuevaLista = R.without([lugarSeleccionado], listaLugares)
4.  setListaLugares(nuevaLista)
```

- *Con el método **filter** de las listas: Este método recibe un lambda y nos devuelve una nueva lista con los objetos de la original para los que el lambda devuelve true. Podemos usarlo en nuestro caso para quedarnos con todos los lugares cuyo id no sea el del lugar que queremos borrar*

```
1.  const nuevaLista = listaLugares.filter( lugar => lugar.id !== lugarSeleccionado.id)
2.  setListaLugares(nuevaLista)
```

4. Ejecuta la app y comprueba que puedes borrar lugares (haz antes una copia de seguridad del archivo **datos.json** por si quieres conservarlos)
5. Ejecuta la app y comprueba que el buscador funciona correctamente
6. Haz un commit titulado “finalizado el tutorial 18”
7. Sitúate en la rama **main** y mezcla en ella la rama **tutorial18**
8. Haz un push