TUTORIAL 20

OPERACIONES CRUD SOBRE GRAPHQL

En este tutorial vamos a hacer una app que realice operaciones CRUD sobre un API **GraphQL** de actores y películas**.** Como veremos, este tipo de API es mucho más flexible que las API tradicionales y va a permitir que nuestra app trabaje con los datos que necesita exactamente, aumentando así la eficiencia en el consumo de recursos

1.- GraphQL

Las API tradicionales ofrecen siempre el mismo formato de respuesta a cada tipo concreto de petición (endpoint). Esto hace que muchas veces se incluyan en las respuestas datos que no necesitamos. Por ejemplo, en el proyecto del tiempo, la respuesta del api tenía muchísimos campos y solo necesitábamos unos pocos.

*Con objeto de poder pedir al API solo los datos que necesitamos, y obtener un JSON con ellos, Facebook desarrolló **GraphQL**, que es un api unificada que admite sentencias para hacer operaciones CRUD (al estilo de las consultas, inserciones, etc del lenguaje SQL) y devuelve sus resultados.*

*Las API GraphQL solo ofrecen un endpoint (llamado **graphql)** y usan el método **POST** para recibir un objeto con la operación a realizar.*

En este proyecto vamos a montar una Fake API GraphQL y realizaremos una app que realice operaciones CRUD sobre ella.

1. Crea un proyecto llamado **rn_actores** y súbelo a **GitHub**
2. Instala las siguientes librerías:
    - Componente **SafeAreaView** → **npx install react-native-safe-area-context**
    - Componente **BouncyCheckbox** → **npm install react-native-bouncy-checkbox**
    - Componente **Image** de Expo → **npx expo install expo-image**
    - Iconos **MaterialIcons** → **npx expo install @expo/vector-icons**
    - Librería **dayjs** → **npm install dayjs**
    - Librería **graphql-request** → **npm install graphql-request**

☞ ***¿Para qué sirve esta librería?*** *Aunque GraphQL es un API y podríamos trabajar con ella usando **axios**, la librería **graphql-request** simplifica trabajar con GraphQL*

3. Crea una rama llamada **tutorial20**

2.- Instalación de una Fake API GraphQL

Al igual que hicimos en el tutorial de operaciones crud sobre APIs tradicionales, vamos a montar en nuestro equipo local una "Fake Api GraphQL", que va a ser un servidor que ofrezca un api GraphQL para realizar operaciones CRUD sobre un archivo JSON.

☞ ***¡Muy importante!*** *Al contrario de json-server, la fake api json-graphql-server solo trabaja en memoria y no realiza modificaciones sobre el archivo de datos. Esto es intencionado, porque su objetivo es solo proporcionar un entorno de pruebas.*

1. Copia el archivo **datos.json** en la carpeta del proyecto
2. Instala **json-graphql-server** con este comando:

npn install json-graphql-server

3. Lanza el servidor con este comando

npx json-graphql-server datos.json

3.- El entorno GraphiQL

*El servidor que hemos instalado permite acceder a un entorno de prueba llamado **GraphiQL** que nos permite generar y probar las sentencias **GraphQL** antes de incluirlas en nuestra app.*

1. Abre el archivo **datos.json** sobre el que trabaja el API y observa su estructura:


```
{
    "actors": [
        {
            "name": "Tom Hanks",
            "birthdate": "1956-07-09",
            "active": "true",
            "imageUrl": "",
            "bio": "Es fanático de los trenes y tiene su propia colección de modelos a escala.",
            "movies": ["Forrest Gump", "Toy Story"]
        },
        {
            "name": "Emma Watson",
            "birthdate": "1990-04-15",
            "active": "true",
            "imageUrl": "",
            "bio": "Graduada en Literatura Inglesa por la Universidad de Brown, además de actriz.
            "movies": ["Harry Potter y la piedra filosofal", "Harry Potter y la cámara secreta"]
        },
```


*Como puedes ver, los nombres de los campos están en **inglés**, y eso es muy importante, porque el servidor va a utilizar dichos nombres para generar automáticamente palabras clave que aparecerán en las consultas, como **allActors***

4. Abre el navegador y entra en esta dirección:

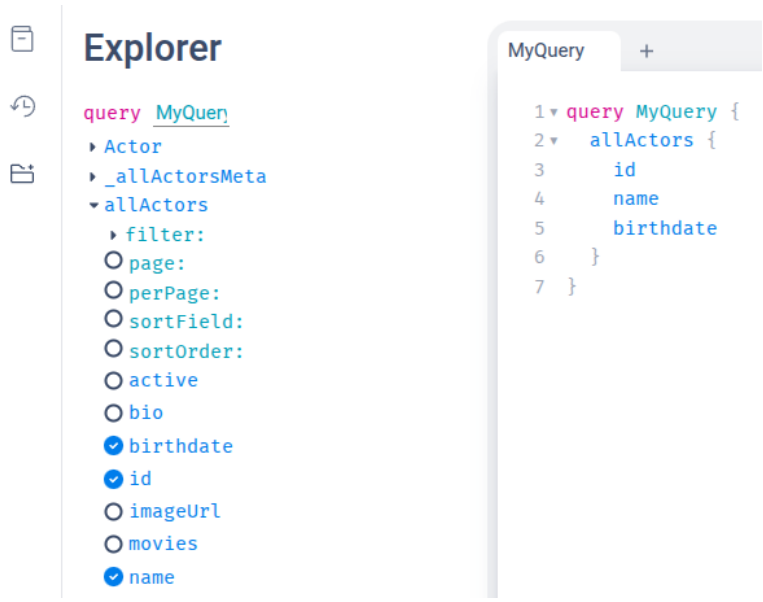
http://localhost:3000/graphql

*El entorno que aparece se denomina **GraphiQL** y es muy útil para probar y construir sentencias **GraphQL***

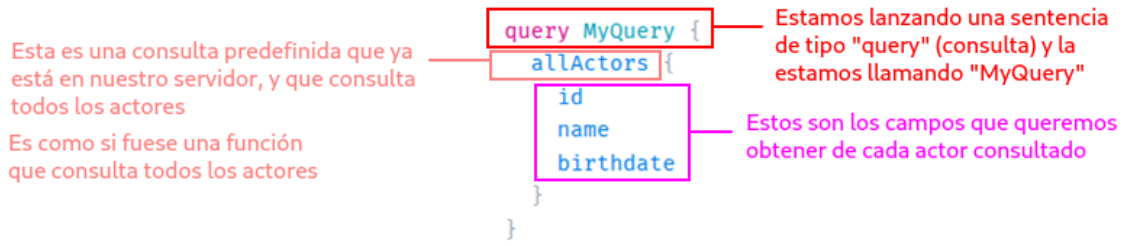
2. Pulsa el icono 

*La ventana que aparece es el **explorador**, donde podremos escribir nuestras sentencias o bien construirlas de forma gráfica*

3. Vamos a construir gráficamente una consulta que consulte todos los actores. Para ello, pulsa en el explorador sobre **allActors** y a continuación, pincha los campos que quieras que se muestren como resultado de la consulta. Por ejemplo, para consultar, el id, nombre y fecha de nacimiento, haríamos esto:



☞ ***¿Qué significa el código de la consulta?*** *El código que hemos generado con el asistente tiene tres partes con el siguiente signficado:*



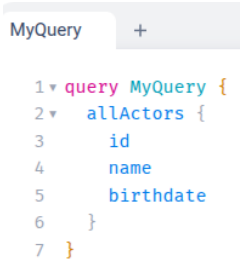
4. Pulsa el botón ▶ y comprueba que aparece un JSON con el resultado de la consulta

*Usando este asistente gráfico construiremos todas las consultas (**query**) y operaciones de modificación (**mutation**) necesarias para nuestra app. Es posible incluir condiciones y todo tipo de filtrado, ordenación, paginación, etc a las consultas.*

4.- Lanzar la consulta con Postman

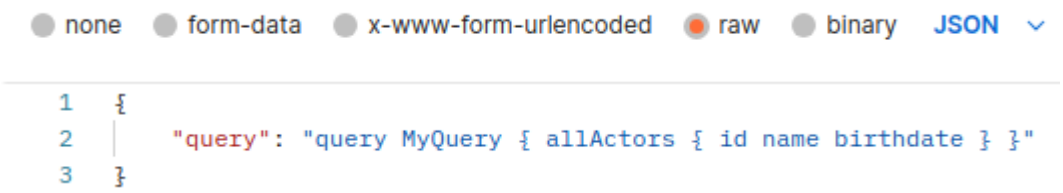
*El programa **Postman** siempre es muy útil para llamar a APIs y analizar sus respuestas. Vamos a ver cómo podemos lanzar la consulta creada en el punto anterior usando **Postman** y analizar la respuesta del api. Esto nos ayudará mucho para saber cómo extraer los datos en nuestra app*

1. En **GraphiQL** selecciona el código de la consulta del apartado anterior y cópiala en el portapapeles



2. Abre el programa **Postman**
3. En la dirección escribe **http://localhost:3000/graphql** y elige el método **POST**

4. En las opciones de parámetros, pulsa **body** y elige **raw** y **json**
5. Escribe un objeto **JSON** que contenga un campo llamado **query** y en él, pega un **string** con la consulta **GraphQL** sin incluir ningún salto de línea



6. Pulsa el botón **Send** y observa que aparece el resultado de la consulta

*Observa que se devuelve un objeto con un campo **data** que posee, dentro de su campo **allActors**, la lista de actores*

5.- Modelo de datos

*Como nuestro programa realiza operaciones sobre datos estructurados (los actores), es buena idea usar **TypeScript** y realizar tipos que representen los conceptos que intervienen en la app.*

*Si miramos la imagen del componente principal, veremos que se muestra una lista en la que solo aparecen las fotos de los actores. Por optimizar recursos, y puesto que **GraphQL** nos deja consultar lo que queramos, vamos a crear un tipo llamado **Actor** que contenga solo el **id** y la url con la foto del actor. De esta forma, si hubiese una cantidad grande de actores, en la memoria del componente principal solo tendríamos el id y foto del actor, sin necesidad de cargar datos innecesarios como la fecha de nacimiento o la lista de películas.*

*Una vez que el usuario pincha un actor, consultaremos el API para obtener el resto de datos necesarios, y para eso, crearemos un tipo llamado **ActorCompleto**. Por tanto, nuestro modelo está formado por dos tipos:*

- ***Actor:** Contiene solo el **id** y la url de la foto de un actor, que es lo mínimo que hace falta para poder mostrarlo en la pantalla principal de la app*
- ***ActorCompleto:** Contiene todos los datos del actor, y es lo que es necesario para las pantallas de detalle y modificación del actor.*

*Aunque no es imprescindible, es muy cómodo tener un tipo de dato que represente el conjunto de datos que aparecen en la pantalla para dar de alta o modificar un actor (que son todos los datos del actor, menos el id). Por eso, crearemos un tipo **DatosFormulario** con dichos campos.*

1. Crea una carpeta llamada **model** y dentro de ella un archivo llamado **Tipos.ts**
2. Crea y exporta el tipo **Actor**, con los campos **id** y **urlFoto**

```
export type Actor = {
    id:string
    urlFoto:string
}
```

3. Crea y exporta el tipo **Actores**, que será un alias para el tipo **Array<Actor>** y así simplificar su escritura cuando haya que trabajar con listas de actores

```
export type Actores = Array<Actor>
```

4. Crea y exporta el tipo **ActorCompleto**, que incluya todos los campos posibles de un actor

```
export type ActorCompleto= Actor & {
    nombre:string
    biografia:string
    fechaNacimiento:string
    activo:boolean
    peliculas:Array<string>
}
```

*Cuando se define un tipo, el operador **&** sirve para "añadir" a un tipo los campos de otro[1].*

5. Crea el tipo **DatosFormulario**, que tenga todos los campos que hacen falta para crear o modificar un actor en el formulario de entrada de datos. Esos campos son todos los de **ActorCompleto**, excepto el id

```
1.  export type DatosFormulario = Omit<ActorCompleto,"id">
```

*En **TypeScript,** se usa **Omit<A,x>** para crear un tipo de dato que tiene todos los campos del tipo **A** excepto el campo **x**[2]*

6. Por último, vamos a crear un tipo **Pelicula** y su correspondiente versión **Peliculas** como alias para **Array<Pelicula>.** Como de una película solo se trabaja con un título, no necesitamos crear un objeto, sino que nos basta con que **Pelicula** sea un alias para el tipo **string**

```
1.  export type Pelicula = string
2.  export type Peliculas = Array<String>
```

*Aunque parezca que el tipo **Pelicula** es innecesario, está bien contemplarlo por si en el futuro se amplía dicho tipo y se necesitan añadir más características de las películas (año, director, foto, etc)*

6.- Consulta de todos los actores

*Vamos a comenzar consultando todos los actores y mostrándolos en un **FlatList,** de manera que se vean sus imágenes en la interfaz principal.*

*Para consultar todos los actores usaremos la librería **graphql-request**, que incluye funciones para definir y lanzar la consulta de forma cómoda usando **GraphQL**.*

1. Crea una carpeta llamada **helpers** y allí un archivo llamado **CrudActores.ts**
2. En dicho archivo, crea una constante llamada **IP** que sea:
   - **10.0.2.2** → si estamos en android
   - **localhost** → en cualquier otro caso

```
1.  const IP = Platform.OS==="android" ? "10.0.2.2" : "localhost"
```

Estas ip son provisionales y solo funcionan cuando probamos la app en un emulador. Si usamos el móvil real debemos usar la ip real del servidor

3. Crea otra constante llamada **URL**, que contenga la dirección del endpoint

```
1.  const URL = `http://${IP}:3000/graphql`
```

4. Crea una función asíncrona **consultarActores** cuyo objetivo sea consultar la lista de todos los actores, y funcione de esta forma:

[1] 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"
[2] Si hay que excluir varios campos, por ejemplo **x** e **y**, escribiremos **Omit<A,x | y>**

- Usa la función **gql** para pasarle un **string** con la consulta de los campos **id** y **urlFoto** del actor
- Usa la función asíncrona **request** para enviar al api la consulta anterior
- Devuelve el campo **allActors** que nos devuelve la respuesta del api

```
1.  export async function consultarActores():Promise<Actores>{
2.      const sentencia = gql`
3.          query  consultarActores{
4.              allActors {
5.                  id
6.                  imageUrl
7.              }
8.          }
9.      `
10.     const respuesta =  await request(URL,sentencia)
11.     return respuesta.allActors
12. }
```

Como vemos, la "función"[3] ***gql*** *permite pasarle un* ***string*** *con la consulta y la función* ***request*** *la pasa al api, retornando de forma asíncrona un* ***json*** *con la respuesta.*

5. Como último paso, antes de dar por finalizada esa función, renombra el campo **imageUrl** de la consulta para que coincida con el nombre **urlFoto** que hemos usamos en el modelo de datos

```
1.  export async function consultarActores():Promise<Actores>{
2.      const sentencia = gql`
3.          query  consultarActores{
4.              allActors {
5.                  id
6.                  urlFoto:imageUrl
7.              }
8.          }
9.      `
10.     const respuesta =  await request(URL, sentencia)
11.     return respuesta.allActors
12. }
```

☞ ***¿Se podría hacer con la librería axios?*** *Sí, porque* ***GraphQL*** *es un api. Tendríamos que usar el método* ***post*** *para pasar al api un objeto con un campo llamado* ***query*** *cuyo valor sea la consulta. Suele hacerse creando una variable llamada* ***query*** *y pasando a* ***axios.post*** *un objeto que la contenga como campo, así:*

```
1.  export async function consultarActores():Promise<Actores>{
2.      const query = `
3.          query consultarActores {
4.              allActors {
5.                  id
6.                  imageUrl: urlFoto
7.              }
8.          }
9.      `}
10.     const respuesta =  await axios.post(URL,{query})
11.     return respuesta.data.data.allActors
12. }
```

6. Abre **App.tsx**, borra todo lo que contiene y teclea **rnfs+intro**
7. Crea estas variables de estado (con sus setter):

[3] En realidad **gql** no es una función tradicional (observa que no lleva paréntesis), sino que es un concepto de **TypeScript** denominado **tagged template literal**

- **listaActores:** Es la lista de actores (contiene objetos **Actor**) y se inicializa con una lista de actores vacía
- **actorSeleccionado:** Es el actor seleccionado por el usuario, ya completado. Podrá admitir un objeto **ActorCompleto** o **undefined** (que es el valor inicial)

```
export default function App() {
  const [listaActores, setListaActores] = useState<Actores>([])
  const [actorSeleccionado, setActorSeleccionado] = useState<ActorCompleto|undefined>(undefined)
  // resto omitido
}
```

8. Crea en **App** una función llamada **mostrarError**, que reciba un mensaje y lo muestre en una ventana emergente

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

9. Crea una función llamada **inicializarListaActores** y haz que en ella se use **consultarActores** para pedirle al api la lista de actores. En caso de que todo vaya bien, se usará esa lista para inicializar **listaActores.** Si se produce un error, se llamará a **mostrarError**

```
function inicializarListaActores(){
  consultarActores()
    .then( lista => setListaActores(lista) )
    .catch( error => mostrarError(error.toString()) )
}
```

10. Haz que **inicializarListaActores** se rellene automáticamente cuando se inicia el componente **App**

```
export default function App() {
  const [listaActores, setListaActores] = useState<Actores>([])
  const [actorSeleccionado, setActorSeleccionado] = useState<ActorCompleto | null>(null)
  useEffect(inicializarListaActores, [])
  // resto omitido
}
```

7.- Consultar todos los datos de un actor

*Ahora que ya tenemos en **App** una lista con objetos **Actor,** vamos a programar las funciones necesarias para consultar todos los datos de un objeto **Actor** y obtener así un objeto **ActorCompleto** que se guardará en **actorSeleccionado***

*Para poder esto, en la consulta necesitamos introducir el **id** del actor cuyos datos queremos buscar y para ello aprovecharemos que **GraphQL** permite definir variables en la consulta, que luego se sustituyen por variables de verdad.*

> ***GraphQL** permite definir variables de los siguientes tipos:*
>
> - ***Int** → número entero*
> - ***Float** → número con decimales*
> - ***String** → cadena de texto*
> - ***Boolean** → valor true/false*

- ***ID*** → *identificador único*
- ***Listas*** → *se encierra el tipo entre corchetes. Por ejemplo,* ***[Int]***
- ***Objetos*** → *primero, creamos el tipo del objeto de forma similar a* ***TypeScript***

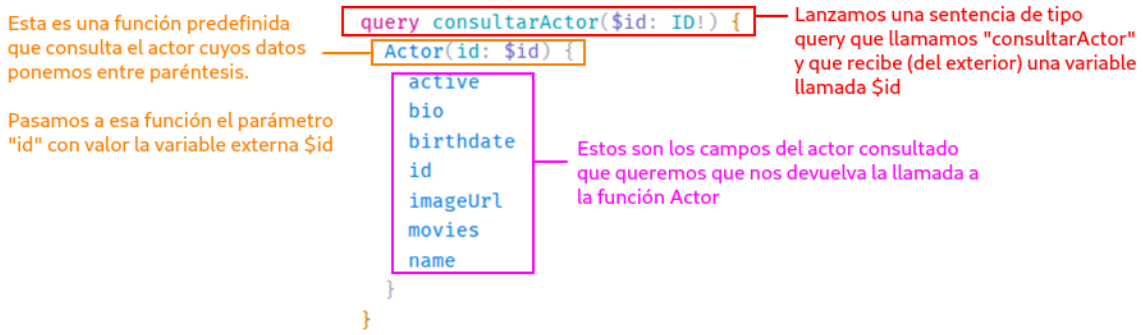
Para definir una variable se utiliza la notación ***$variable : tipo****, y si el tipo lleva* ***!*** *significa que la variable es obligatoria.*

1. Accede a **GraphiQL** con el navegador y abre el explorador
2. En la la zona de la izquierda, pon al nombre de la consulta **consultarActor**
3. A continuación pulsa **Actor** y marca todos los campos

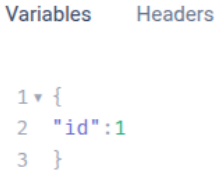
4. En el código de la consulta, abre unos paréntesis después del nombre de la consulta (como si la consulta fuese una función) y escribe dentro las variables que necesitará la consulta, que en este caso solo es **$id** de tipo **ID!**

5. Por último, observa que junto a la palabra **Actor** aparecen unos paréntesis que encierra un **id.** Borra las comillas dobles vacías que tiene, y en su lugar, escribe la variable **$id**

☞ ***¿Qué significa el código de la consulta?*** *Nuevamente, podemos distinguir tres partes en la sentencia:*



6. Para poder lanzar en **GraphiQL** una consulta que lleva variables, es necesario escribir en la zona de variables un objeto con el nombre y valor de ellas (ya sin el **$**). Escribe en la zona **Variables** un objeto que asigne el valor **1** al **id**



7. Lanza la consulta y comprueba que se obtiene el actor cuyo **id** vale **1**
8. Copia en el portapapeles la consulta que se ha generado
9. Abre **CrudActores.ts** y crea y exporta una función asíncrona llamada **completarActor,** que reciba un objeto **Actor** y devuelva el correspondiente objeto **ActorCompleto** usando la consulta obtenida en el punto anterior, y renombrando sus campos para que coincidan con el modelo

```
export async function completarActor(actor:Actor):Promise<ActorCompleto>{
    const consulta = gql`
            query consultarActor($id: ID!) {
                Actor(id: $id) {
                    activo: active
                    biografia: bio
                    fechaNacimiento: birthdate
                    id
                    urlFoto: imageUrl
                    peliculas: movies
                    nombre: name
                }
            }
        `
    const variables = { id:actor.id }
    const respuesta = await request(URL, consulta, variables)
    return respuesta.Actor
}
```

*Como puede verse, para llamar a una consulta que define variables pasamos a la función **request** como segundo parámetro un objeto con los nombres y valor de las variables.*

☞ ***¿Por qué es necesario crear las variables en la consulta?¿No se pueden poner directamente en el string?*** *En principio, podríamos pensar que el mecanismo de variables de **GraphQL** es innecesario, porque podríamos inyectar el valor **actor.id** directamente en la consulta, como cualquier otra variable que se inserta en un string:*

```
const consulta = gql`
  query consultarActor{
      Actor(id: ${actor.id}) {
          activo: active
          biografia: bio
          fechaNacimiento: birthdate
          id
          urlFoto: imageUrl
          peliculas: movies
          nombre: name
      }
  }`
```

*El problema de esta forma es que es **vulnerable a ataques SQL injection** y un atacante podría llegar a ejecutar consultas maliciosas o no autorizadas*

10. En **App** crea una función llamada **seleccionarActor**, que utilice la función **completarActor** para recuperar los datos de un objeto **Actor** que recibe como parámetro y si no hay errores, actualice la variable **actorSeleccionado**

```
function seleccionarActor(actor:Actor){
  completarActor(actor)
    .then( actorCompleto => setActorSeleccionado(actorCompleto))
    .catch( error => mostrarError(error.toString()) )
}
```

☞ ***¿Para qué sirve esta función?*** *Como hemos dicho, la app ahorra memoria mostrando solo objetos **Actor** en la interfaz, y completándolos cuando se selecciona uno. Así pues, cuando el usuario seleccione un objeto **Actor** en la interfaz, se llamará a **seleccionarActor** para consultar el api y obtener un objeto **ActorCompleto**, que se guardará en **actorSeleccionado***

8.- Diseño del componente para mostrar un actor

*El componente principal de la app contiene un **FlatList** donde se mostrarán las fotos de todos los actores de la lista. Por tanto, comenzaremos creando el componente **VisorActor,** donde se mostrará cada objeto **Actor** del **FlatList***

1. Crea una carpeta llamada **components** y en ella un archivo llamado **VisorActor.tsx**
2. Abre **VisorActor.tsx** y teclea **rnfs+intro**
3. Añade a la función **VisorActor** estos dos props
    - **actor:** Es el objeto **Actor** que se muestra en el componente
    - **seleccionarActor:** Función que se ejecuta al pulsar en el componente, y sirve para cargar el resto de datos del actor y marcarlo como actor seleccionado en **App**

```
type VisorActorProps={
    actor:Actor,
    seleccionarActor: (actor:Actor) => void
}
export default function VisorActor({actor,seleccionarActor}:VisorActorProps) {
  // resto omitido
}
```

4. Añade a **VisorActor** una variable de estado llamada **errorCarga** que inicialmente valdrá **false** y se pondrá a **true** si se produce un error al cargar la imagen del actor

```
export default function VisorActor({actor, seleccionarActor }:VisorActorProps) {
  const [errorCarga, setErrorCarga] = useState(false)
  // resto omitido
}
```

5. A continuación, crea dos variables llamadas **imagenError** e **imagenCarga** que carguen las imágenes **error.jpg** y **loading.jpg** de la carpeta **assets**

```
export default function VisorActor({actor, seleccionarActor }:VisorActorProps) {
  const [errorCarga, setErrorCarga] = useState(false)
  const imagenError = require("../assets/error.jpg")
  const imagenCarga = require("../assets/loading.jpg")
  // resto omitido
}
```

6. Diseña el componente **VisorActor** de forma que:
    - Un **Pressable** encierre un **Image**
    - Si la variable **errorCarga** es **false**, se mostrará en el **Image** la imagen del actor, y si es **true,** se mostrará la imagen cargada en **imagenError**
    - Durante la carga se mostrará en el **Image** la imagen cargada en **imagenCarga** (esto se hace con el prop **placeholder** del **Image**)
    - Si se produce un error durante la carga, se pondrá a **true** la variable **errorCarga** (esto se hace pasando una lambda con dicha acción al prop **onError** del **Image**)
    - Tras cargarse la foto, se producirá un efecto de entrada (**fade-in)** de **800** milisegundos de duración (esto se hace con el prop **transition** del **Image**)

- Al pulsar en el **Pressable** se llamará a **completarActor** pasándole el objeto **actor**
- La imagen tendrá un tamaño de 150x200

```
export default function VisorActor({actor, seleccionarActor }:VisorActorProps) {
  // inicio omitido
  return (
    <Pressable onPress={ () => seleccionarActor (actor) }>
      <Image
        source={errorCarga? imagenError : actor.urlFoto}
        placeholder={imagenCarga}
        style={{ width:150, height: 200}}
        transition={800}
        onError={()=> setErrorCarga(true)}
    />
    </Pressable>
  )
}
```

*Aunque parece que ya está terminado, hay un detalle que queda por hacer. Si un actor no tiene foto, se verá la imagen de error y la variable de estado **errorCarga** quedará a **true** para siempre. Eso hace que si más adelante, con la app modificamos la foto y ya no da error, la foto no se vea porque errorCarga mantiene a true su valor.*

4. Añade un efecto que ponga a **false** la variable **errorCarga** cuando cambie el prop **actor**

```
export default function VisorActor({actor,seleccionarActor}:VisorActorProps) {
  const [errorCarga, setErrorCarga] = useState(false)
  const imagenError = require("../assets/error.jpg")
  const imagenCarga = require("../assets/loading.jpg")
  useEffect( () => setErrorCarga(false), [actor])
  // resto omitido
}
```

9.- Diseño del componente para mostrar una película

*En la pantalla del detalle de un actor mostraremos sus películas en un **FlatList** y por tanto, necesitamos un componente para mostrarlas. Dicho componente va a ser un **chip**, que es un botón alargado de esquinas redondeadas que contiene un texto y usualmente, un icono para cerrarlo.*

1. En **components** crea un archivo llamado **VisorPelicula.tsx**
2. Abre **VisorPelicula.tsx** y teclea **rnfs+intro**
3. Añade a la función **VisorActor** estos dos props
   - **pelicula:** Es el objeto **Pelicula** que se muestra en el componente
   - **accionBorrarPelicula:** Es una función que se lanza al pulsar el icono de cerrar integrado en el componente, y su misión será borrar la película

```
type VisorPeliculaProps = {
    pelicula:Pelicula,
    accionBorrarPelicula: (pelicula:Pelicula) => void
}
export default function VisorPelicula({pelicula, accionBorrarPelicula}:VisorPeliculaProps) {
  // resto omitido
}
```

4. Sin mirar la solución, diseña el componente **VisorPelicula** de esta forma:

- El icono a la derecha es un **MaterialIcons** de nombre **“cancel”,** tamaño **16** y color **#E0E0FF**
- Al pulsar el componente se llamará a **accionBorrarPelicula** pasando la película definida en el prop **pelicula**


```
export default function VisorPelicula({pelicula, accionBorrarPelicula}:VisorPeliculaProps) {
  return (
    <View style={styles.contenedor}>
      <Text style={styles.texto}>{pelicula}</Text>
      <Pressable onPress={() => accionBorrarPelicula(pelicula)}>
        <MaterialIcons name={"cancel"} size={16} color={"#E0E0FF"}/>
      </Pressable>
    </View>
  )
}
```

10.- Diseño del componente Toolbar

*Nuestra app tendrá una sencilla **Toolbar** donde se mostrará el título de la app y un icono de añadir que al pulsarlo abrirá un modal para crear un nuevo actor. Vamos a programar dicho componente.*

1. En **components** crea un archivo llamado **Toolbar.tsx**
2. Abre **Toolbar.tsx** y teclea **rnfs+intro**
3. Agrega a**Toolbar** un prop llamado **abrirFormularioNuevoActor**, que será la función que se ejecutará al pulsar el icono de añadir, y que sirve para abrir un formulario que permita crear un nuevo actor.

```
type ToolbarProps = {
    abrirFormularioNuevoActor : () => void
}
export default function Toolbar({abrirFormularioNuevoActor}:ToolbarProps) {
  // resto omitido
}
```

4. Sin mirar la solución, diseña el componente **Toolbar** de esta forma:

```
export default function Toolbar({ abrirFormularioNuevoActor }: ToolbarProps) {
  return (
    <View style={styles.container}>
      <Text style={styles.titulo}>Todo Actores</Text>
      <Pressable onPress={ abrirFormularioNuevoActor }>
        <MaterialIcons name={"add"} size={24} color={"white"} />
      </Pressable>
    </View>
  );
}
```

11.- Diseño del componente principal

Vamos a integrar los componentes creados, para así dar forma al componente principal de la app.

1. Abre **App.tsx,** borra todo lo que contiene y teclea **rnfs+intro**
2. Añade a **App** una función vacía llamada **abrirFormularioNuevoActor**

```
function abrirFormularioNuevoActor(){
  // aquí abriremos un modal para crear un nuevo actor
}
```

3. Añade a **App** una función llamada **getEtiquetaActor**, que reciba un objeto **Actor** y devuelva la etiqueta **VisorActor** con sus props rellenados de esta forma:
   - **actor:** El objeto **Actor** recibido como parámetro
   - **seleccionarActor:** La función que completa y el objeto **Actor** y pone como seleccionado al correspondiente objeto **ActorCompleto**

```
function getEtiquetaActor(actor:Actor){
  return <VisorActor actor={actor} seleccionarActor={ seleccionarActor }/>
}
```

*Recuerda que este tipo de funciones son las que nos proporcionan los componentes de los ítems que se muestran en el prop **renderItem** de un **FlatList** cuando dichos componentes necesitan recibir varios props (como en este caso, donde **VisorActor** necesita los props **actor** y **seleccionarActor**)*

4. Añade a **App** estos componentes:
   - Un **Toolbar** que reciba la función **abrirFormularioNuevoActor**
   - Un **FlatList** que muestre la lista de actores en **3** columnas. Como clave primaria se usará el **id** de cada actor, y la función **getEtiquetaActor** nos dará el componente donde se mostrará cada actor (el **renderItem**)

```
export default function App() {
  // inicio omitido
  return (
    <SafeAreaView style={styles.contenedor}>
      <Toolbar abrirFormularioNuevoActor={ abrirFormularioNuevoActor }/>
      <FlatList
        data={listaActores}
        keyExtractor={ actor => actor.id}
        renderItem={ ({item}) => getEtiquetaActor(item)}
        numColumns={3}
      />
    </SafeAreaView>
  )
}
```

5. Ejecuta la app y comprueba que aparece la interfaz principal con todos los actores cargados y mostrándose sus fotos.
6. Añade a **App** las siguientes funciones vacías, que servirán para realizar, más adelante, las funcionalidades de la app
    - **accionCrearActor:** Será la función que realizará la acción de crear un actor a partir de los datos rellenados en el formulario. Por ese motivo, recibe un objeto **DatosFormulario**
    - **accionModificarActor:** Será la función que realizará la acción de modificar el actor guardado en **actorSeleccionado** con los datos rellenados en el formulario. Para ello, recibe el **id** del actor que hay que modificar, y también objeto **DatosFormulario** con los datos que el usuario rellenados en el formulario.
    - **accionBorrarActor:** Será la función que realizará la acción de borrar el actor guardado en **actorSeleccionado**

```
function accionCrearActor(datos:DatosFormulario){
  // aquí solicitaremos al api la creación de un actor con los datos del formulario
}
function modificarActor(idActor:String, datos:DatosFormulario){
  // aquí solicitaremos al api modificar actorSeleccionado con los datos del formulario
}
function accionBorrarActor(){
  // aquí solicitaremos al api borrar actorSeleccionado
}
```

12.- El componente para mostrar un actor

*En esta app vamos a utilizar un único componente **EditorActor** para las cuatro acciones que hace la app:*

- *Crear un nuevo actor*
- *Mostrar el detalle de un actor seleccionado por el usuario*
- *Editar el actor seleccionado por el usuario*
- *Borrar el actor seleccionado por el usuario*

*Conseguiremos esto añadiendo a **EditorActor** estos props:*

- ***actorSeleccionado:** será un objeto opcional de tipo **ActorCompleto.** Cuando no se pase, la pantalla entrará en modo de "creación de nuevo actor" y si no, permitirá ver, modificar y borrar el actor.*
- ***accionCrearActor:** Función usada para crear un actor, recibiendo todos los datos del formulario recogidos en un objeto de tipo **DatosFormulario***
- ***accionModificarActor:** Función para modificar un actor con todos los datos introducidos en el formulario. Recibe el **id** del actor que se va a modificar y un objeto **DatosFormulario**.*
- ***accionBorrarActor:** Función usada para borrar un actor a partir de su **id***
- ***setModalVisible:** Función que sirve para cerrar la ventana modal*

1. En **components** crea un archivo llamado **EditorActor.tsx**
2. Abre **EditorActor.tsx** y teclea **rnfs+intro**
3. Añade a **EditorActor** los props que hemos comentado anteriormente

```
type EditorActorProps = {
  actorSeleccionado ?: ActorCompleto
  accionCrearActor: (datos:DatosFormulario) => void
  accionModificarActor: (idActor:string, datos:DatosFormulario) => void
  accionBorrarActor: (idActor: string) => void
  setModalVisible: React.Dispatch<React.SetStateAction<boolean>>
};
export default function EditorActor({
  actorSeleccionado,accionCrearActor,accionModificarActor,accionBorrarActor,
  setModalVisible}: EditorActorProps) {
  // resto omitido
}
```

4. Añade a **EditorActor** variables de estado para todos los datos del formulario: nombre, fecha de nacimiento, biografía, activo, url de la foto y lista de películas. Todos ellos tomarán su valor inicial del objeto **actorSeleccionado**, y si este fuese **undefined,** se inicializarán con valores vacíos (false para **activo**).

```
export default function EditorActor({
   actorSeleccionado,accionCrearActor,accionModificarActor,accionBorrarActor,
   setModalVisible}: EditorActorProps) {
   const [nombre,setNombre] = useState(actorSeleccionado?.nombre ?? "")
   const [fechaNacimiento, setFechaNacimiento] = useState(actorSeleccionado?.fechaNacimiento ?? "")
   const [activo, setActivo] = useState(actorSeleccionado?.activo ?? false)
   const [biografia, setBiografia] = useState(actorSeleccionado?.biografia ?? "")
   const [urlFoto, setUrlFoto] = useState(actorSeleccionado?.urlFoto ?? "")
   const [peliculas, setPeliculas] = useState<Array<string>>(actorSeleccionado?.peliculas ?? [])
   // 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 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***

5. En **EditorActor** crea dos funciones vacías llamadas **accionBorrarPelicula** (recibe un objeto **Pelicula**) y **accionNuevaPelicula** (no recibe nada)

```
function accionBorrarPelicula() {
  // aquí borraremos una película de la lista de películas del actor
}
function accionNuevaPelicula(pelicula:Pelicula){
  // aquí añadiremos una nueva película a la lista de películas del actor
}
```

En este tutorial vamos a dejar estas dos funciones vacías, porque nos centraremos en trabajar con los actores sin tener en cuenta sus películas. En el próximo tutorial actualizaremos la lista de películas del actor, creando y borrando películas.

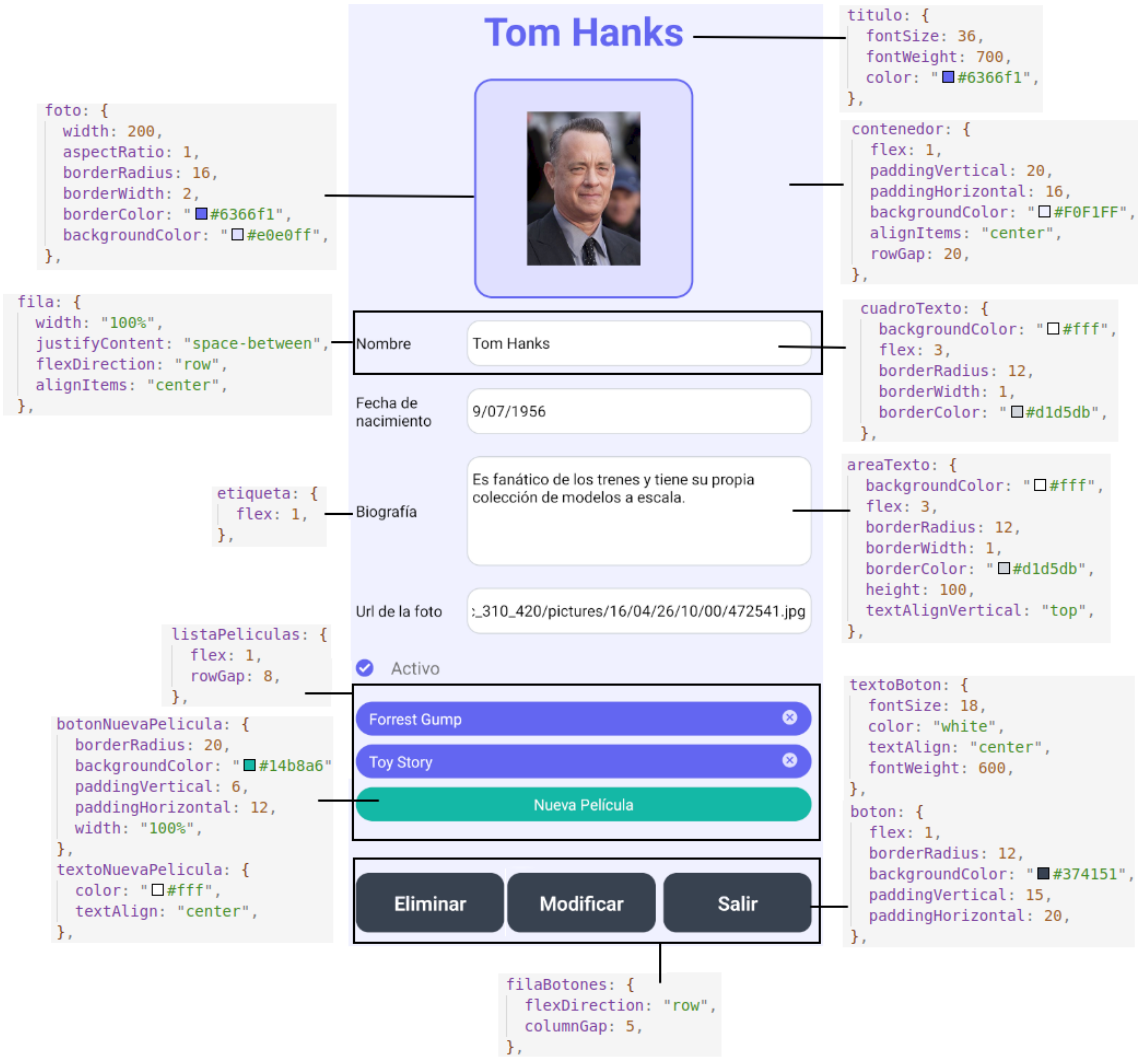
6. Añade a **EditorActor** una función **getEtiquetaPelicula** que reciba una película y nos devuelva su correspondiente etiqueta **VisorPelicula**. Se rellenará el prop **accionBorrarPelicula** de dicha etiqueta con la función **accionBorrarPelicula** y como clave primaria se usará el título de la película

```
function getEtiquetaPelicula(pelicula: Pelicula) {
  return (
    <VisorPelicula
        pelicula={pelicula} key={película} accionBorrarPelicula={accionBorrarPelicula} />
  );
}
```

7. Sin mirar la solución, haz que la función **EditorActor** devuelva el siguiente componente, teniendo en cuenta que:

    - Los datos que se ven en la imagen deberán ser sustituidos por los datos de las variable de entorno
    - El componente que muestra la imagen del actor tiene un borde, un color de fondo y usa **scale-down** en el prop **contentFit** para reducir la imagen de forma que entre en el componente manteniendo la relación de aspecto (cociente ancho/alto de la foto)
    - Si **actorSeleccionado** es **undefined,** el título que se mostrará será “Nuevo Actor”, y si no, se mostrará el nombre del actor.
    - Si **actorSeleccionado** es **undefined,** solo se mostrará un botón para crear un nuevo actor. En caso contrario, se mostrarán botones para modificar y borrar el actor.
    - En todos los casos, se mostrará el botón de salir
    - La fecha de nacimiento se mostrará en formato **día/mes/año**
    - La lista de películas será una lista de componentes **VisorPelicula** generados dinámicamente (con **map**

        - Toda la lista estará encerrada en un **View** que ocupará todo el espacio posible de la pantalla, y dentro contendrá un **ScrollView** para mostrar las películas que no quepan.
        - Se mostrará un **VisorPelicula** para cada película del actor, y dicha etiqueta se obtendrá con la función **getEtiquetaPelicula**
        - Tras la lista de películas se colocará el botón para añadir una nueva película.
        - Al pulsar los botones del fondo se llamará a las funciones correspondientes de los props
        - A la función **accionCrearActor** se le pasará un objeto que contenga todas las variables de estado (dicho objeto se convierte automáticamente en un **DatosFormulario** porque las variables de estado se llaman igual que los campos del tipo **DatosFormulario**)
        - A la función **accionModificarActor** se le pasará el **id** de **actorSeleccionado** y el mismo objeto **DatosFormulario** pasado a la función **accionCrearActor**




```tsx
export default function EditorActor({
  actorSeleccionado,crearActor, modificarActor,borrarActor,setModalVisible,
}: EditorActorProps) {
  // inicio omitido
  return (
    <View style={styles.contenedor}>
      <Text style={styles.titulo}>
        {actorSeleccionado?.nombre ?? "Nuevo Actor"}
      </Text>
      <Image
        style={styles.foto}
        source={urlFoto}
        contentFit={"scale-down"}
      />
      <View style={styles.fila}>
        <Text style={styles.etiqueta}>Nombre</Text>
        <TextInput
          placeholder="Nombre del actor"
          value={nombre}
          onChangeText={setNombre}
          style={styles.cuadroTexto}
        />
      </View>
      <View style={styles.fila}>
        <Text style={styles.etiqueta}>Fecha de nacimiento</Text>
        <TextInput
          placeholder="Fecha de nacimiento del actor"
          value={fechaNacimiento!== ""?dayjs(fechaNacimiento).format("D/MM/YYYY")} : ""
          onChangeText={setFechaNacimiento}
          style={styles.cuadroTexto}
```

```
        />
      </View>
      <View style={styles.fila}>
        <Text style={styles.etiqueta}>Biografía</Text>
        <TextInput
          placeholder="Biografía del actor"
          multiline={true}
          numberOfLines={3}
          value={biografia}
          onChangeText={setBiografia}
          style={styles.areaTexto}
        />
      </View>
      <View style={styles.fila}>
        <Text style={styles.etiqueta}>Url de la foto</Text>
        <TextInput
          placeholder="Url de la foto del actor"
          value={urlFoto}
          onChangeText={setUrlFoto}
          style={styles.cuadroTexto}
        />
      </View>
      <View style={styles.fila}>
        <BouncyCheckbox
          size={16}
          fillColor={"#6366F1"}
          unFillColor={"white"}
          isChecked={activo}
          text={"Activo"}
          textStyle={{ textDecorationLine: "none" }}
        />
      </View>
      <ScrollView>
        <View style={styles.listaPeliculas}>
          {peliculas.map((pelicula) => getEtiquetaPelicula(pelicula))}
          <Pressable style={styles.botonNuevaPelicula} onPress={accionNuevaPelicula}>
            <Text style={styles.textoNuevaPelicula}>Nueva Película</Text>
          </Pressable>
        </View>
      </ScrollView>
      <View style={styles.filaBotones}>
        {actorSeleccionado !== undefined && (
          <>
            <Pressable
              style={styles.boton}
              onPress={() => borrarActor(actorSeleccionado.id)}
            >
              <Text style={styles.textoBoton}>Eliminar</Text>
            </Pressable>
            <Pressable
              style={styles.boton}
              onPress={() =>
              accionModificarActor(
                       id, {nombre,biografia,urlFoto,fechaNacimiento,activo,películas})}
              )>
              <Text style={styles.textoBoton}>Modificar</Text>
            </Pressable>
          </>
        )}
        {
          actorSeleccionado === undefined && (
            <Pressable
              style={styles.boton}
              onPress={() =>
                  accionCrearActor({nombre,biografia,urlFoto,fechaNacimiento,activo,películas})}
            >
              <Text style={styles.textoBoton}>Crear</Text>
            </Pressable>
          )
        }
        <Pressable style={styles.boton} onPress={() => setModalVisible(false)}>
          <Text style={styles.textoBoton}>Salir</Text>
        </Pressable>
```

```
      </View>
    </View>
  );
}
```

13.- Consultar el detalle de un actor

Ahora que ya tenemos diseñados todos los componentes de la interfaz, vamos a usarlos para ir realizando las operaciones CRUD sobre los actores. Comenzaremos mostrando el detalle de un actor cuando se pulse en su imagen.

1. Abre **App.tsx**
2. Modifica la función **seleccionarActor** para que se abra un modal una vez que se obtiene el **ActorCompleto**

```
function seleccionarActor(actor:Actor){
  completarActor(actor)
    .then( actorCompleto => {
      setActorSeleccionado(actorCompleto)
      setModalVisible(true)
    })
    .catch( error => mostrarError(error.toString()) )
}
```

3. Al final de la función **App**, coloca un bloque de renderizado condicional que abra un modal que muestre dentro un **EditorActor** pasando en sus props todas las funciones que hay definidas hasta este momento en **App**:
    - **actorSeleccionado:** El **ActorCompleto** que ha sido seleccionado
    - **setModalVisible:** La función que muestra/oculta el modal
    - **accionCrearActor:** La función que recibe todos los datos del formulario y crea un actor nuevo
    - **accionModificarActor:** La función que recibe todos los datos del formulario y modifica el actor seleccionado
    - **accionBorrarActor:** La función que recibe el **id** de un actor y lo borra

```
export default function App() {
  // inicio omitido
  return (
    <SafeAreaView style={styles.contenedor}>
      <Toolbar crearNuevoActor={crearNuevoActor}/>
      <FlatList
        data={listaActores}
        keyExtractor={ actor => actor.id}
        renderItem={ ({item}) => getEtiquetaActor(item)}
        numColumns={3}
      />
      {
        modalVisible && (
          <Modal animationType={"slide"} transparent={true}>
            <EditorActor
              actorSeleccionado={actorSeleccionado}
              setModalVisible={setModalVisible}
              accionCrearActor={accionCrearActor}
              accionModificarActor={accionModificarActor}
              accionBorrarActor={accionBorrarActor}/>
          </Modal>
        )
      }
    </SafeAreaView>
  )
}
```

4. Ejecuta la app y comprueba que al pulsar sobre un actor, se abre un modal con un **EditorActor** que muestra todos los datos del actor seleccionado.

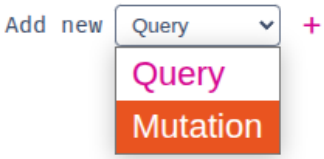
14.- Modificar el actor seleccionado

*Vamos a hacer que funcione el botón de modificar de la pantalla de detalle del actor. Para eso, programaremos la correspondiente función **modificarActor** que use el api, y luego la llamaremos en la función **accionModificarActor** que hemos puesto en el componente **App**.*

*En primer lugar, usaremos **GraphiQL** para construir gráficamente la sentencia de modificación, que va a ser una "mutación"*

*En **GraphQL** se llama **query** a las consultas y **mutations** a las operaciones de inserción, modificación y borrado*

1. Abre **GraphiQL** y pulsa el explorador
2. En la lista desplegable, elige **Mutation** y pulsa el botón **+**



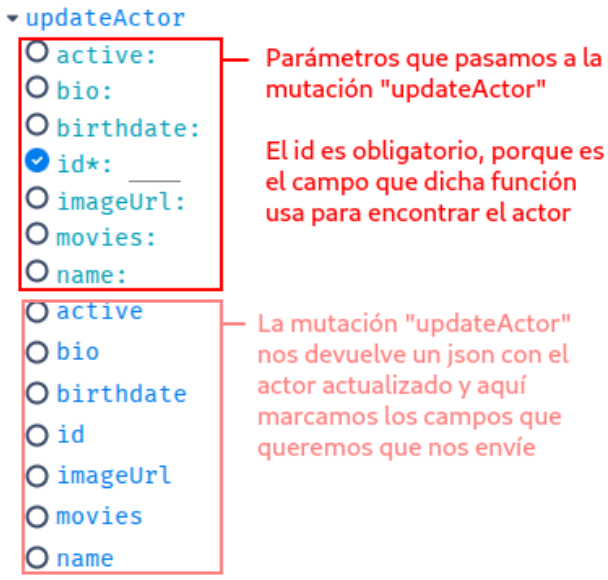
Observa que aparecen cinco "mutaciones" ya definidas:

- ***createActor*** → *Crea un nuevo actor*
- ***createManyActor*** → *Crea una lista de actores*
- ***updateActor*** → *Actualiza un actor*
- ***deleteActor*** → *Borra un actor*
- ***removeActor*** → *Borra el actor de una relación con otra entidad (en nuestro proyecto esto no tiene ninguna utilidad)*

Tienes que ver las "mutaciones" como funciones que ya trae nuestro servidor. El Fake Api que manejamos solo admite estas (que detecta automáticamente a partir del archivo JSON). En un servidor real, podríamos crear nuestras propias mutaciones.

3. Llama a la mutación **modificarActor**

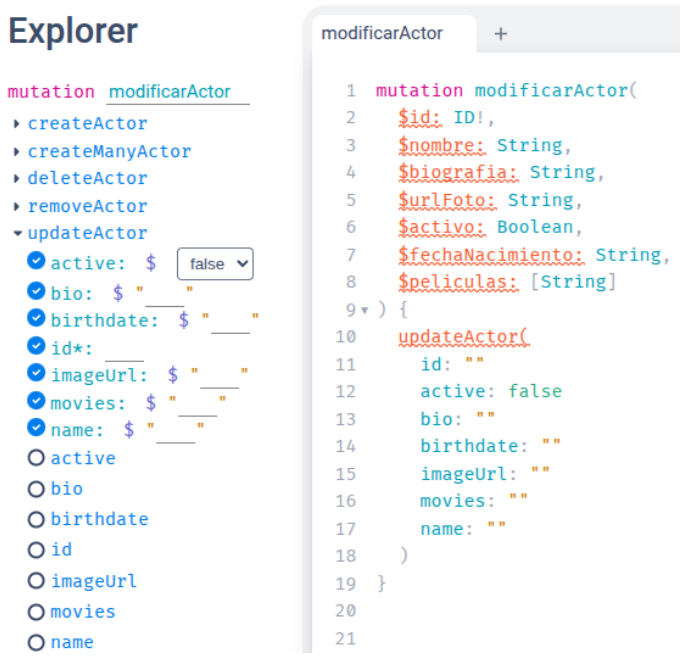
4. Selecciona la mutación **updateActor** y comprueba que la zona de la izquierda se divide en estas dos zonas:



5. En la zona del código, añade paréntesis y crea variables para todos los parámetros que le vamos a enviar al servidor (id, nombre, biografía, fechaNacimiento, urlFoto, activo y películas). Como son muchas variables, puedes escribir cada una en un renglón diferente

> ☞ ***¿No podríamos enviar un objeto de tipo DatosFormulario que las contenga todas?*** *Sí, pero no en el fake api que estamos usando. En un servidor real podríamos definir el tipo* ***DatosFormulario*** *y pasar una variable de ese tipo*

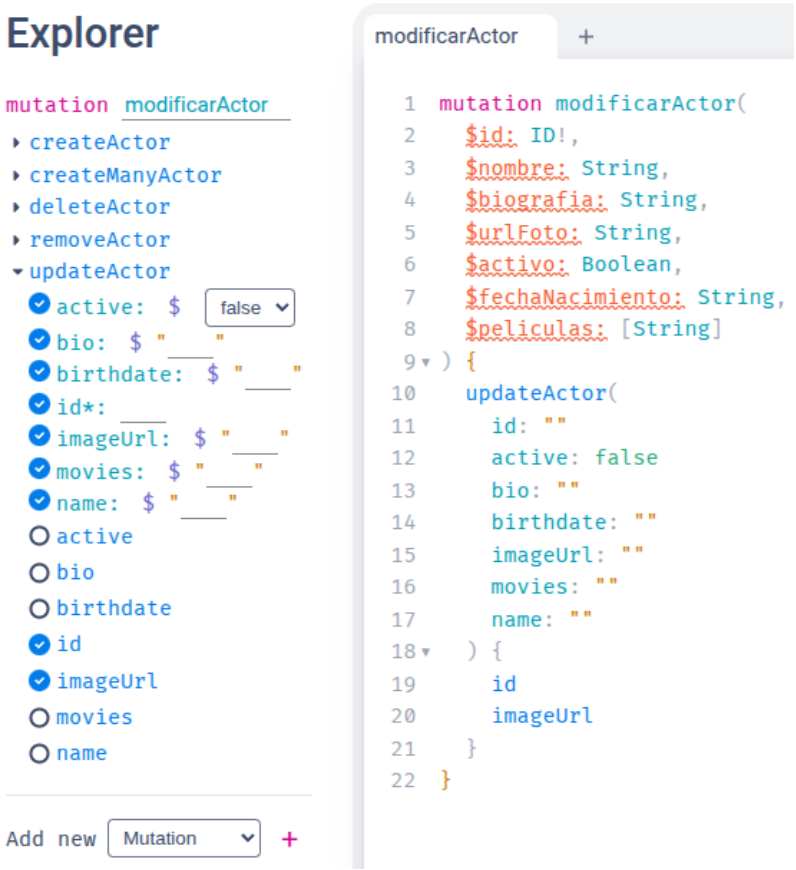
6. En la zona de la izquierda, marca todas las casillas de color azul claro



*Marcamos esas casillas porque vamos a pasar a **updateActor** todos los parámetros que puede recibir.*

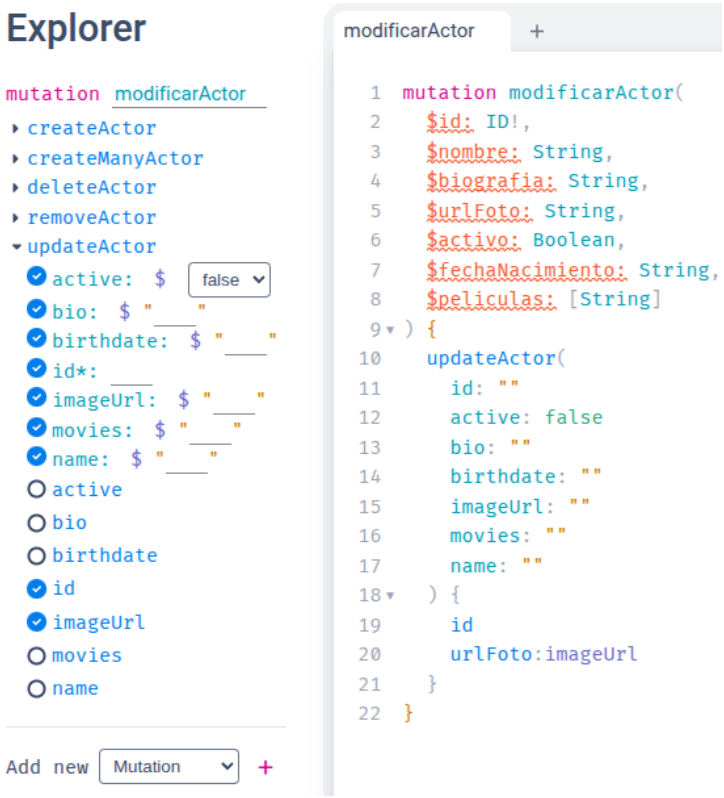
- ***updateActor** coge el id del actor y actualiza los campos que le pasamos como parámetros (en nuestro caso, todos)*

7. Marca las casillas azul oscuro **id** e **imageUrl**



*Marcamos esas casillas porque **updateActor** nos devuelve el actor actualizado, y con ellas seleccionamos los campos de dicho actor que queremos que formen la respuesta del api*

8. Renombra el campo **imageUrl** de la respuesta del api para que coincida con el campo **urlFoto** del tipo **Actor** de nuestro modelo



☞ ***¿Por qué queremos que el api nos devuelva los campos id y urlFoto del actor modificado?*** *Esos campos coinciden con nuestro tipo **Actor** y recordemos, la variable **listaActores** contiene objetos de ese tipo**.** Al consultar esos campos, el api nos devuelve un objeto **Actor** que reemplazaremos en **listaActores***

9. Selecciona el código de la consulta generada y pégalo en el portapapeles
10. Abre **CrudActores.ts** y crea una función asíncrona llamada **modificarActor**, que reciba el **id** de un actor (que va a ser el actor que se va a modificar) y un objeto de tipo **DatosFormulario** con los datos de la modificación que se hará al actor que tenga ese **id.** La función devolverá un objeto de tipo **ActorCompleto** con los datos del modificado

```
export async function modificarActor(idActor:string,datos:DatosFormulario):Promise<Actor>{
  // aquí el api actualizará el actor con id = idActor
}
```

11. Crea una constante con la sentencia de actualización que has creado:

```
export async function modificarActor(idActor:string,datos:DatosFormulario):Promise<Actor>{
  const sentencia = gql`
      mutation modificarActor(
        $id: ID!,
        $nombre: String,
        $biografia: String,
        $urlFoto: String,
        $activo: Boolean,
        $fechaNacimiento: String,
        $peliculas: [String]
    ) {
      updateActor(
        id: $id
        active: $activo
        bio: $biografia
        birthdate: $fechaNacimiento
        imageUrl: $urlFoto
        movies: $peliculas
        name: $nombre
      ) {
        id
        urlFoto:imageUrl
      }
    }
  `
}
```

> ☞ ***¿Por qué se usa la comilla ` ?*** *Además de insertar variables, ese tipo de comilla nos permite que la cadena de texto ocupe varias líneas.*

12. Crea un objeto **variables** con todos los datos del objeto **datos** y además, el id recibido como parámetro en **idActor**

```
export async function modificarActor(idActor:string,datos:DatosFormulario):Promise<Actor>{
  const sentencia = gql`
      // sentencia omitida
  `
  const variables = {
    ...datos,
    id:idActor
  }
}
```

> *Recuerda que cuando ponemos **…datos** es lo mismo que si escribimos en ese lugar todas las parejas clave-valor de **datos** una tras otra*

13. Termina la función llamando a la función asíncrona **request,** pasándole la url del api, la variable con la sentencia y el objeto con las variables que se pasarán al api. La función devolverá el campo **updateActor** de la respuesta del api

```
export async function modificarActor(idActor:string,datos:DatosFormulario):Promise<Actor>{
  const sentencia = gql`
      // sentencia omitida
  `
  const variables = {
    ...datos,
    id:idActor
  }
  const respuesta = await request(URL,sentencia,variables)
  return respuesta.updateActor
}
```

> *El json que devuelve el api siempre contiene un campo con el nombre de la función que ha llamado (en este caso, **updateActor**)*

14. Por último, vamos a programar la función **accionModificarActor** de forma que, si **actorSeleccionado** no es **undefined** (esto significa que hay un actor seleccionado por el usuario), se llame a la función **modificarActor.** En caso de que todo vaya bien:
    - Se usará **map** para crear una nueva lista de objetos **Actor** en la que el actor modificado es actualizado
    - Se actualizará **listaActores** con dicha lista
    - Se pondrá **actorSeleccionado** a **undefined** (y así es como si ya no tuviéramos ningún actor seleccionado para mostrar o editar)
    - Se cerrará la ventana modal del detalle de actor

```
function accionModificarActor(idActor:string, datos:DatosFormulario){
  modificarActor(idActor,datos)
    .then( actorModificado => {
      const nuevaLista = listaActores.map(
            actor => actor.id === actorModificado.id? actorModificado : actor)
      setListaActores(nuevaLista)
      setModalVisible(false)
     })
    .catch( error => mostrarError(error.toString()))
}
```

*El método **map** de la lista crea una nueva lista en la que cada elemento de la original es sustituido por el resultado de aplicarle el lambda que recibe.*

En nuestro caso, el lambda deja todos los elementos iguales, excepto el actor cuyo id es el modificado, que es reemplazado por el objeto que devuelve el api.

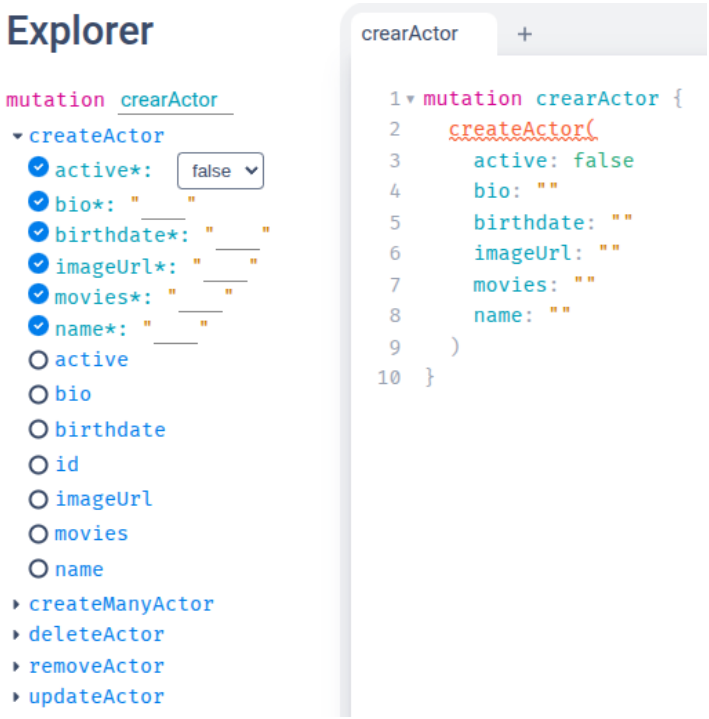
15. Ejecuta la app y comprueba cómo puedes modificar los datos del actor y los cambios se reflejan inmediatamente (por ejemplo, se nota bien al modificar el nombre del actor)

15.- Creación de un nuevo actor

Ahora v*amos a hacer que al pulsar en el botón **+** de la **Toolbar,** se abra el **EditorActor** vacío para rellenar los datos de un nuevo actor, y programaremos su botón de crear para solicitar al api el registro de ese actor. Para eso, programaremos la función **crearActor** que usará el api, y luego la llamaremos en la función **accionCrearActor** que hemos puesto en el componente **App**.*

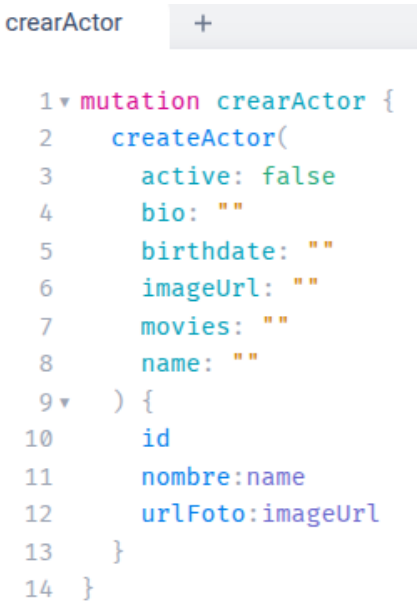
*En primer lugar, usaremos **GraphiQL** para construir gráficamente la sentencia de creación.*

1. Abre **GraphiQL** y pulsa el botón para acceder al explorador.
2. Añade una nueva sentencia de tipo **Mutation** llamada **crearActor**
3. Elige la mutación predefinida **createActor** y observa que por defecto, esa mutación ya incluye parámetros para recibir todos los campos del actor



*La función predefinida **createActor** crea un actor (le pone un id automático) y devuelve un json con los datos del actor creado.*

4. En la zona azul, selecciona los campos **id, name** e **imageUrl** de forma que el json que devuelve la operación contenga dichos campos y renómbralos para que coincidan con los del tipo **Actor**



*Hacemos esto porque queremos que al crear un actor nuevo, se obtenga un objeto **Actor** que añadiremos a **listaActores***

5. En la mutación **crearActor**, añade entre paréntesis variables obligatorias para todos los campos del actor y asígnalas a los parámetros que usa la función **createActor**

crearActor +

```
mutation crearActor (
  $nombre:String!,
  $activo:Boolean!,
  $biografia:String!,
  $fechaNacimiento:String!,
  $urlFoto:String!,
  $peliculas:[String]!
){
  createActor(
    active: $activo
    bio: $biografia
    birthdate: $fechaNacimiento
    imageUrl: $urlFoto
    movies: $peliculas
    name: $nombre
  ) {
    id
    nombre:name
    urlFoto:imageUrl
  }
}
```

6. Selecciona y copia la sentencia que has creado en el portapapeles
7. Abre **CrudActores.ts** y crea una función asíncrona llamada **crearActor** que reciba como parámetro un objeto **DatosFormulario** (contendrá los datos del actor que se va a crear), y devuelva un objeto **Actor** con el actor creado

```
export async function crearActor(datos:DatosFormulario):Promise<Actor>{
  const sentencia = gql`
      mutation crearActor (
        $nombre:String!,
        $activo:Boolean!,
        $biografia:String!,
        $fechaNacimiento:String!,
        $urlFoto:String!,
        $peliculas:[String]!
    ){
      createActor(
        active: $activo
        bio: $biografia
        birthdate: $fechaNacimiento
        imageUrl: $urlFoto
        movies: $peliculas
        name: $nombre
      ) {
        id
        nombre:name
        urlFoto:imageUrl
      }
    }
  `
  const respuesta = await request(URL,sentencia,datos)
  return respuesta.createActor
}
```

*En este caso el objeto **datos** ya nos sirve como objeto de variables, puesto que contiene exactamente los mismos campos que necesita el api*

8. Abre **App** y programa la función **abrirFormularioNuevoActor** de forma que **actorSeleccionado** se ponga a **undefined** y a continuación se muestre el modal.

```
function abrirFormularioNuevoActor(){
  setActorSeleccionado(undefined)
  setModalVisible(true)
}
```

*Recuerda que esta función es la que se lanza cuando pulsamos el botón **+** de la **Toolbar**. De esta forma, al pulsarlo, se deseleccionará el posible actor que pudiera estar seleccionado y después se abrirá el modal que mostrará **EditorActor** en blanco*

9. En **App** programa la función **accionCrearActor** de forma que se llame a la función **crearActor** y si todo va bien, cree una nueva lista añadiendo el objeto **Actor** que devuelve dicha función a **listaActores**, acutalice con ella **listaActores** y por último, cierre el modal.

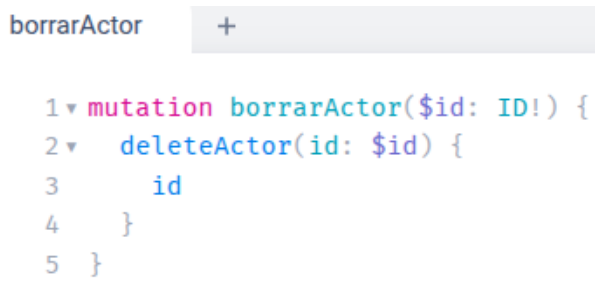
```
function accionCrearActor(datos:DatosFormulario){
  crearActor(datos)
    .then( nuevoActor => {
      const nuevaLista = [nuevoActor,...listaActores] // ponemos el actor nuevo al principio
      setListaActores(nuevaLista)
      setModalVisible(false)
    })
    .catch( error => mostrarError(error.toString()))
}
```

10. Ejecuta la app y comprueba que puedes crear nuevos actores

15.- Borrar el actor seleccionado

*Para terminar, vamos a programar el botón de borrar de forma que se borre el actor guardado en **actorSeleccionado.** Como es habitual, programaremos una función auxiliar llamada **borrarActor** que hará que el api borre el actor, y la llamaremos en **App** dentro de la función **accionBorrarAccion***

1. Abre **GraphiQL** y crea una mutación llamada **borrarActor**, que utilice la función predefinida **deleteActor**.
   - Como variable, recibirá el **id** del actor que se va a borrar
   - Dicha variable se pasará a la función **deleteActor**
   - La función **deleteActor** devolverá un objeto con el **id** del actor borrado



2. Abre **CrudActores.ts** y crea una función llamada **borrarActor** que reciba el **id** del actor que se va a borrar y que lance al api la sentencia creada anteriormente. La función devolverá el **id** del actor borrado.

```
export async function borrarActor(idActor:string):Promise<string>{
  const sentencia = gql`
    mutation borrarActor($id: ID!) {
      deleteActor(id: $id) {
        id
      }
    }
  `
  const variables = {
    id: idActor
  }
  const respuesta = await request(URL, sentencia, variables)
  return respuesta.deleteActor.id
}
```

*Como la respuesta del api siempre es un objeto, **respuesta.deleteActor** sería un objeto con un campo **id**. Nuestra función devuelve un **string** con el **id**, y por tanto, devolvemos el campo **id** de la respuesta.*

3. Abre **App** y programa la función **accionBorrarActor** de forma que compruebe que hay un actor seleccionado (en caso contrario, no se podría borrar nada) y en dicho caso, muestre una ventana de confirmación para borrar el actor. En caso de confirmarse, se llamará a una función (inexistente de momento) que reciba el **id** del actor, llamada **realizarBorrado**

```
function accionBorrarActor(){
  if(actorSeleccionado!==undefined){
    Alert.alert(
      `¿Deseas borrar a ${actorSeleccionado.nombre}`,
      "Un actor eliminado no podrá ser recuperado",
      [
        {text:"Si, eliminar", onPress: () => realizarBorrado(actorSeleccionado.id) },
        {text:"No, cancelar"}
      ]
    )
  }
}
```

4. Programa la función **realizarBorrado** de manera que llame a **borrarActor** y si todo va bien:
   - cree una nueva lista eliminando el actor que devuelve la operación de la lista de actores. Se usará **filter** para obtener una lista que contenga todos los actores de **listaActores**, excepto el que tiene el **id** del actor borrado
   - asigne dicha lista a **listaActores**
   - cierre el modal

```
function realizarBorrado(idActor:string){
   borrarActor(idActor)
     .then( idBorrado => {
       const nuevaLista = listaActores.filter( actor => actor.id!== idBorrado)
       setListaActores(nuevaLista)
       setModalVisible(false)
     })
     .catch(error => mostrarError(error.toString()))
 }
```

5. Haz un commit titulado “finalizado el tutorial 20”
6. Sitúate en la rama **main** y mezcla en ella la rama **tutorial20**
7. Haz un **push**