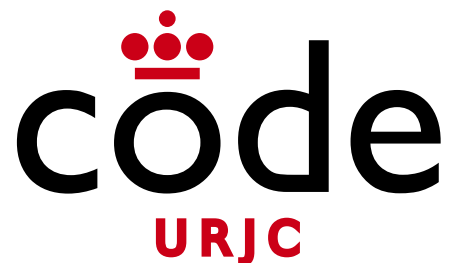


Desarrollo de Aplicaciones para Dispositivos  
Móviles

Tema 3: Desarrollo Híbrido

Tema 3.4: Persistencia en Flutter  
Bases de datos



©2023

Michel Maes

Algunos derechos reservados

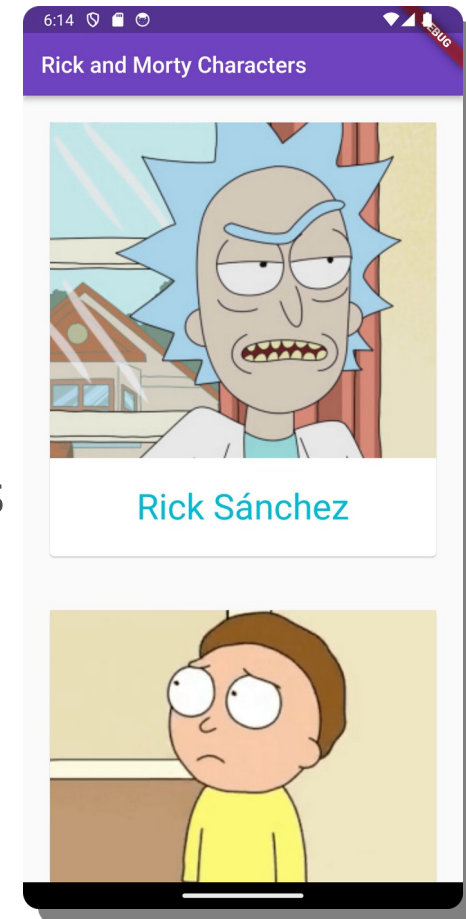
Este documento se distribuye bajo la licencia  
“Atribución-CompartirIgual 4.0 Internacional”  
de Creative Commons Disponible en  
<https://creativecommons.org/licenses/by-sa/4.0/deed.es>

# Bases de datos en Flutter

- Desde Flutter podemos realizar conexiones a bases de datos externas
- Si nuestra aplicación se va a compilar a **Android o iOS** (para web necesitaremos una externa) podemos hacer uso de un SQLite para tener persistencia en local
- Utilizaremos la librería sqflite de Dart

# Bases de datos en Flutter

- Veremos como trabajar con sqflite desde un ejemplo guiado
- Partiremos de una aplicación que muestra personajes (obtenidos de una lista)
- Veremos cómo:
  - Crear una tabla para almacenar los personajes
  - Agregar personajes a la BBDD
  - Recuperar personajes de la BBDD
  - Borrar personajes de la BBDD



## Crear una tabla para almacenar los personajes

- En primer lugar, añadiremos la librería **sqflite** al archivo pubspec.yaml.
- De manera adicional, añadiremos **path** para poder obtener el fichero de la base de datos local

```
dependencies:  
  flutter:  
    sdk: flutter  
  sqflite: ^2.3.0  
  path: ^1.8.2
```

- Recuerda realizar un **Pub get**

## Crear una tabla para almacenar los personajes

- Modificaremos la clase Character

```
class Character {  
  final int? id;  
  final String name;  
  final String image;  
  
  Character({  
    this.id,  
    required this.name,  
    required this.image,  
  });  
  
  Map<String, dynamic> toMap() {  
    return {  
      'name': name,  
      'image': image,  
    };  
  }  
  
  @override  
  String toString() {  
    return 'Character{id: $id, name: $name, image: $image}';  
  }  
}
```

## Crear una tabla para almacenar los personajes

- Modificaremos la clase Character

```
class Character {  
  final int? id;  
  final String name;  
  final String image;  
  
  Character({  
    this.id,  
    required this.name,  
    required this.image,  
  });  
  
  Map<String, dynamic> toMap() {  
    return {  
      'name': name,  
      'image': image,  
    };  
  }  
  
  @override  
  String toString() {  
    return 'Character{id: $id, name: $name, image: $image}';  
  }  
}
```

Añadimos el campo ID

## Crear una tabla para almacenar los personajes

- Modificaremos la clase **Character**

```
class Character {  
  final int? id;  
  final String name;  
  final String image;  
  
  Character({  
    this.id,  
    required this.name,  
    required this.image,  
  });  
  
  Map<String, dynamic> toMap() {  
    return {  
      'name': name,  
      'image': image,  
    };  
  }  
  
  @override  
  String toString() {  
    return 'Character{id: $id, name: $name, image: $image}';  
  }  
}
```

Creamos el método **toMap**,  
que transforma nuestro objeto  
en un mapa procesable por la  
base de datos



## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

Creemos un método `db()` que nos devuelve una `Future` que contiene nuestra BBDD

## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

Esta linea será necesaria para  
llamar al método **antes** de  
ejecutar el método `runApp()`

## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

Al crear la base de datos con **openDatabase()**, en primer lugar le pasamos la ruta al archivo local (lo crea si no existe)

## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

Podemos definir qué hacer al crear la base de datos. Este es un buen lugar para crear la tabla **Characters**

## Crear una tabla para almacenar los personajes

- Creamos la clase **CharacterService**

```
import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class CharacterService {

  static Future<Database> db() async {
    WidgetsFlutterBinding.ensureInitialized();
    return openDatabase(
      join(await getDatabasesPath(), 'characters_database.db'),
      onCreate: (db, version) {
        return db.execute(
          'CREATE TABLE Characters(id INTEGER PRIMARY KEY, name TEXT, image TEXT)',
        );
      },
      version: 1,
    );
  }
}
```

El parámetro **id** será la clave primaria y auto-incremental

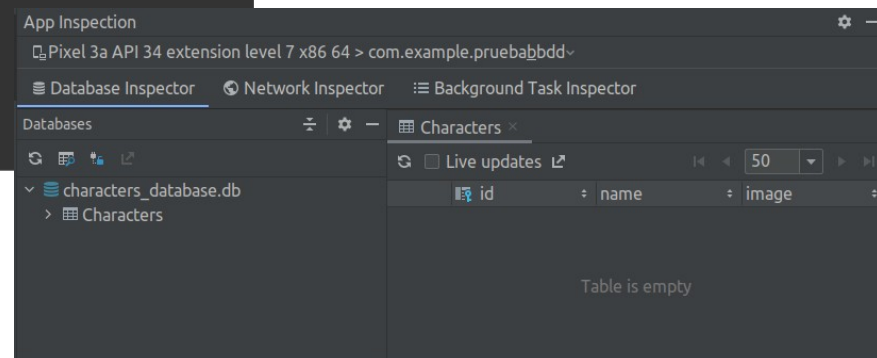
# Bases de datos en Flutter

## Crear una tabla para almacenar los personajes

- Llamamos a nuestro método db() en main.dart
- Podemos comentar runApp() por el momento
- Si arrancamos la aplicación, creará la tabla **Characters**
- Podemos verla en App Inspection > Database Inspector

```
import 'package:flutter/material.dart';
import 'package:prueba_bbdd/CharacterService.dart';

void main() async {
  CharacterService.db();
  //runApp(const HomeScreen());
}
```



## Agregar personajes a la base de datos

- Añadimos el método **insertCharacter** a CharacterService

```
static Future<void> insertCharacter(Character character) async {  
  final connection = await db();  
  
  await connection.insert(  
    'Characters',  
    character.toMap(),  
    conflictAlgorithm: ConflictAlgorithm.replace,  
  );  
}
```



## Agregar personajes a la base de datos

- Añadimos el método `insertCharacter` a `CharacterService`

```
static Future<void> insertCharacter(Character character) async {  
  final connection = await db();  
  
  await connection.insert(  
    'Characters',  
    character.toMap(),  
    conflictAlgorithm: ConflictAlgorithm.replace,  
  );  
}
```

Obtenemos la conexión a la base de datos con `db()` y realizamos la inserción

## Agregar personajes a la base de datos

- Añadimos el método `insertCharacter` a `CharacterService`

```
static Future<void> insertCharacter(Character character) async {  
  final connection = await db();  
  
  await connection.insert(  
    'Characters',  
    character.toMap(),  
    conflictAlgorithm: ConflictAlgorithm.replace,  
  );  
}
```

Le pasamos el mapa con las propiedades del objeto  
Character

## Agregar personajes a la base de datos

- Añadimos el método **insertCharacter** a CharacterService

```
static Future<void> insertCharacter(Character character) async {  
    final connection = await db();  
  
    await connection.insert(  
        'Characters',  
        character.toMap(),  
        conflictAlgorithm: ConflictAlgorithm.replace,  
    );  
}
```

Añadimos la estrategia a seguir ante conflictos (en este caso, simplemente reemplazar)

# Bases de datos en Flutter

## Agregar personajes a la base de datos

- Llamamos al método en main.dart

```
void main() async {  
  CharacterService.db();  
  CharacterService.insertCharacter(Character(name: "Rick Sánchez",  
image: "https://rickandmortyapi.com/api/character/avatar/1.jpeg"))  
  //runApp(const HomeScreen());  
}
```

App Inspection

Pixel 3a API 34 extension level 7 x86 64 > com.example.prueba\_bdd

Database Inspector   Network Inspector   Background Task Inspector

Databases

characters\_database.db

Characters

id : INTEGER

name : TEXT

image : TEXT

Characters

Live updates

Results are read-only

id	name	image
1	Rick Sánchez	https://rickandmortyapi.com/api/character/avat

Podemos observar que se  
añade en el **Database  
Inspector**

## Recuperar personajes de la base de datos

- Añadimos el método **loadCharacters** a CharacterService

```
static Future<List<Character>> loadCharacters() async {  
    final connection = await db();  
  
    final List<Map<String, dynamic>> maps = await connection.query('Characters');  
  
    return List.generate(maps.length, (i) {  
        return Character(  
            id: maps[i]['id'] as int,  
            name: maps[i]['name'] as String,  
            image: maps[i]['image'] as String,  
        );  
    });  
}
```

## Recuperar personajes de la base de datos

- Añadimos el método **loadCharacters** a CharacterService

```
static Future<List<Character>> loadCharacters() async {  
  final connection = await db();  
  
  final List<Map<String, dynamic>> maps = await connection.query('Characters');  
  
  return List.generate(maps.length, (i) {  
    return Character(  
      id: maps[i]['id'] as int,  
      name: maps[i]['name'] as String,  
      image: maps[i]['image'] as String,  
    );  
  });  
}
```

Recuperamos la lista de personajes a través de la conexión con **query**.  
Obtenemos el resultado como una lista de mapas

## Recuperar personajes de la base de datos

- Añadimos el método **loadCharacters** a CharacterService


```
static Future<List<Character>> loadCharacters() async {  
    final connection = await db();  
  
    final List<Map<String, dynamic>> maps = await connection.query('Characters');  
  
    return List.generate(maps.length, (i) {  
        return Character(  
            id: maps[i]['id'] as int,  
            name: maps[i]['name'] as String,  
            image: maps[i]['image'] as String,  
        );  
    });  
}
```

Creamos una lista de **Character** con los parámetros recuperados de la BBDD y la devolvemos

## Recuperar personajes de la base de datos

- Llamamos al método en main.dart

```
void main() async {  
  CharacterService.db();  
  var characters = await CharacterService.loadCharacters();  
  print(characters);  
  runApp(const HomeScreen());  
}
```



```
I/flutter ( 5326): [Character{id: 1, name: Rick Sánchez, image:  
https://rickandmortyapi.com/api/character/avatar/1.jpeg}]
```

El personaje estará cargado de la anterior ejecución



## Recuperar personajes de la base de datos

- Modificaremos la app para que utilice la base de datos en lugar de los datos en memoria
- Dejamos el método **main()** como estaba al principio

```
void main() async {  
  runApp(const HomeScreen());  
}
```

- Dejamos el atributo **characters** de **\_HomeScreenState** como una lista vacía

```
class _HomeScreenState extends State<HomeScreen> {  
  List<Character> characters = [];
```

## Recuperar personajes de la base de datos

- Añadimos un nuevo método a `_HomeScreenState` que dé valor a *characters*. Lo llamaremos en `initState()`

```
@override
void initState() {
  super.initState();
  _getRefreshCharacters();
}

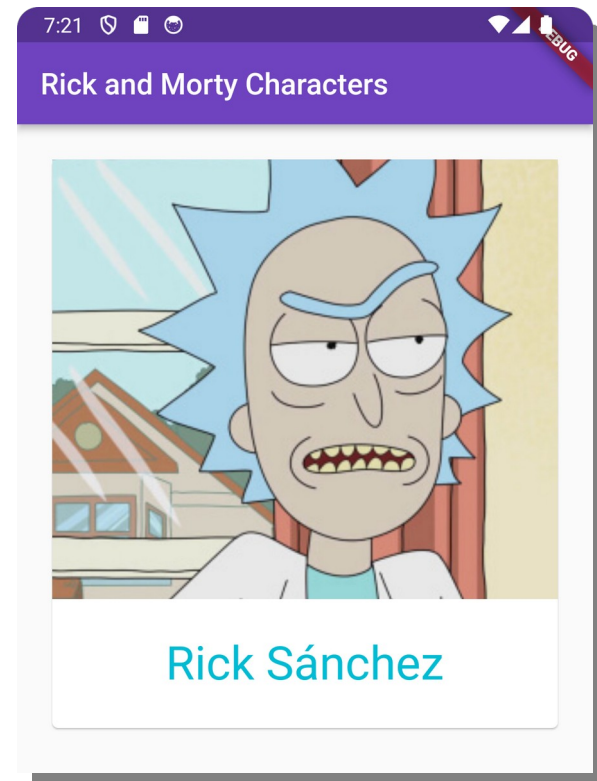
void _getRefreshCharacters() async {
  final data = await CharacterService.loadCharacters();
  setState(() {
    characters = data;
  });
}
```

Utilizamos `setState()` para forzar que se recargue la vista

## Recuperar personajes de la base de datos

- Nuestra vista utiliza **ListView** y **CharacterWidget** para mostrar la lista de personajes

```
// No copiar
ListView(
  children: [
    for (var character in characters)
      CharacterWidget(
        character: character,
      ),
  ],
),
```



## Agregar personajes a la base de datos (Cont.)

- A continuación añadiremos un formulario en la vista conectado a nuestro método **insertCharacter()**
- Añadimos el siguiente código a **\_HomeScreenState**

```
final TextEditingController _nameController = TextEditingController();
final TextEditingController _imageController = TextEditingController();

Future<void> addCharacter() async {
  await CharacterService.insertCharacter(
    Character(name: _nameController.text, image: _imageController.text));
  _nameController.clear();
  _imageController.clear();
  _getRefreshCharacters();
}
```

## Agregar personajes a la base de datos (Cont.)

- A continuación añadiremos un formulario en la vista conectado a nuestro método **insertCharacter()**
- Añadimos el siguiente código a **\_HomeScreenState**

```
final TextEditingController _nameController = TextEditingController();  
final TextEditingController _imageController = TextEditingController();  
  
Future<void> addCharacter() async {  
  await CharacterService.insertCharacter(  
    Character(name: _nameController.text, image: _imageController.text));  
  _nameController.clear();  
  _imageController.clear();  
  _getRefreshCharacters();  
}
```

Las variables de la clase **TextEditingController** nos permitirán almacenar el valor de los campos en componentes **TextField**

## Agregar personajes a la base de datos (Cont.)

- A continuación añadiremos un formulario en la vista conectado a nuestro método **insertCharacter()**
- Añadimos el siguiente código a **\_HomeScreenState**

```
final TextEditingController _nameController = TextEditingController();
final TextEditingController _imageController = TextEditingController();

Future<void> addCharacter() async {
  await CharacterService.insertCharacter(
    Character(name: _nameController.text, image: _imageController.text));
  _nameController.clear();
  _imageController.clear();
  _getRefreshCharacters();
}
```

En el método addCharacter  
llamaremos a **insertCharacter()**

## Agregar personajes a la base de datos (Cont.)

- A continuación añadiremos un formulario en la vista conectado a nuestro método **insertCharacter()**
- Añadimos el siguiente código a **\_HomeScreenState**

```
final TextEditingController _nameController = TextEditingController();
final TextEditingController _imageController = TextEditingController();

Future<void> addCharacter() async {
  await CharacterService.insertCharacter(
    Character(name: _nameController.text, image: _imageController.text));
  _nameController.clear();
  _imageController.clear();
  _getRefreshCharacters();
}
```

Limpiamos los campos de  
texto

## Agregar personajes a la base de datos (Cont.)

- A continuación añadiremos un formulario en la vista conectado a nuestro método **insertCharacter()**
- Añadimos el siguiente código a **\_HomeScreenState**

```
final TextEditingController _nameController = TextEditingController();
final TextEditingController _imageController = TextEditingController();

Future<void> addCharacter() async {
  await CharacterService.insertCharacter(
    Character(name: _nameController.text, image: _imageController.text));
  _nameController.clear();
  _imageController.clear();
  _getRefreshCharacters();
}
```

Volvemos a cargar los personajes



## Agregar personajes a la base de datos (Cont.)

- Para la vista, añadimos un widget propio llamado **CharacterForm** en la misma columna que **Expanded**, pasándole los controladores y un callback.

```
child: Column(  
  children: [  
    CharacterForm(  
      nameController: _nameController,  
      imageController: _imageController,  
      onCreateCharacter: () {  
        setState(() {  
          addCharacter();  
        });  
      },  
    ),  
    Expanded(...),  
  ],  
)
```

```

class CharacterForm extends StatelessWidget {
  const CharacterForm({
    super.key,
    required TextEditingController nameController,
    required TextEditingController imageController,
    required this.onCreateCharacter,
  }) : _nameController = nameController, _imageController = imageController;

  final TextEditingController _nameController;
  final TextEditingController _imageController;
  final Function onCreateCharacter;

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(15.0),
      child: Column(
        children: [
          TextField(
            controller: _nameController,
            decoration: const InputDecoration(hintText: 'Name'),
          ),
          const SizedBox( height: 10),
          TextField(
            controller: _imageController,
            decoration: const InputDecoration(hintText: 'Image'),
          ),
          const SizedBox( height: 20),
          ElevatedButton(
            onPressed: () { onCreateCharacter(); },
            child: const Text('Create new character'),
          ),
        ],
      ),
    );
  }
}

```

```

class CharacterForm extends StatelessWidget {
  const CharacterForm({
    super.key,
    required TextEditingController nameController,
    required TextEditingController imageController,
    required this.onCreateCharacter,
  }) : _nameController = nameController, _imageController = imageController;

  final TextEditingController _nameController;
  final TextEditingController _imageController;
  final Function onCreateCharacter;

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(15.0),
      child: Column(
        children: [
          TextField(
            controller: _nameController,
            decoration: const InputDecoration(hintText: 'Name'),
          ),
          const SizedBox( height: 10),
          TextField(
            controller: _imageController,
            decoration: const InputDecoration(hintText: 'Image'),
          ),
          const SizedBox( height: 20),
          ElevatedButton(
            onPressed: () { onCreateCharacter(); },
            child: const Text('Create new character'),
          ),
        ],
      ),
    );
  }
}

```

Campos de texto que guardarán el valor introducido en los controladores

```

class CharacterForm extends StatelessWidget {
  const CharacterForm({
    super.key,
    required TextEditingController nameController,
    required TextEditingController imageController,
    required this.onCreateCharacter,
  }) : _nameController = nameController, _imageController = imageController;

  final TextEditingController _nameController;
  final TextEditingController _imageController;
  final Function onCreateCharacter;

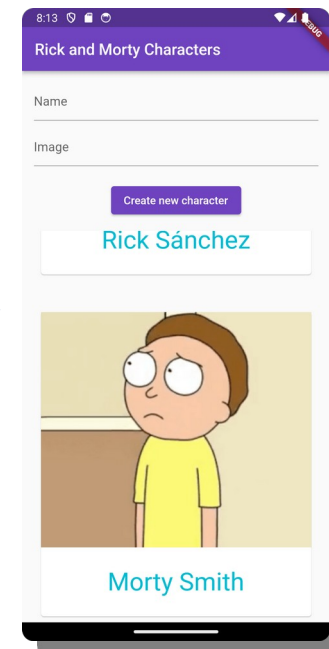
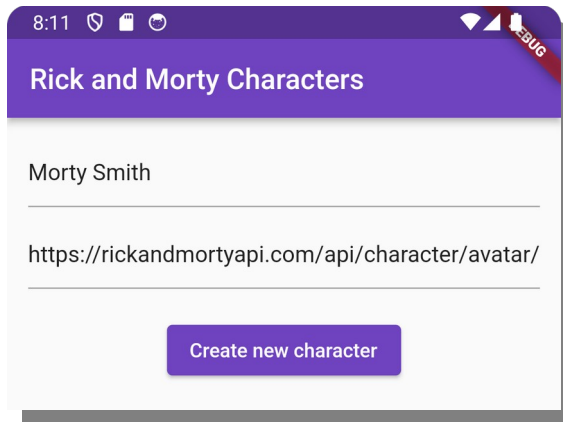
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(15.0),
      child: Column(
        children: [
          TextField(
            controller: _nameController,
            decoration: const InputDecoration(hintText: 'Name'),
          ),
          const SizedBox( height: 10),
          TextField(
            controller: _imageController,
            decoration: const InputDecoration(hintText: 'Image'),
          ),
          const SizedBox( height: 20),
          ElevatedButton(
            onPressed: () { onCreateCharacter(); },
            child: const Text('Create new character'),
          ),
        ],
      ),
    );
  }
}

```

Botón del formulario que  
llama a **onCreateCharacter**

## Agregar personajes a la base de datos (Cont.)

- Ya podemos agregar nuevos personajes a la base de datos
- Cuando se añadan, se actualizará la lista de personajes automáticamente



## Borrar personajes de la BBDD

- Por último, añadiremos la opción de poder borrar un personaje
- Crearemos el método **deleteCharacter** en **CharacterService**

```
static Future<void> deleteCharacter(int id) async {  
  final connection = await db();  
  
  await connection.delete(  
    'Characters',  
    where: 'id = ?',  
    whereArgs: [id],  
  );  
}
```

## Borrar personajes de la BBDD

- Por último, añadiremos la opción de poder borrar un personaje
- Crearemos el método **deleteCharacter** en **CharacterService**

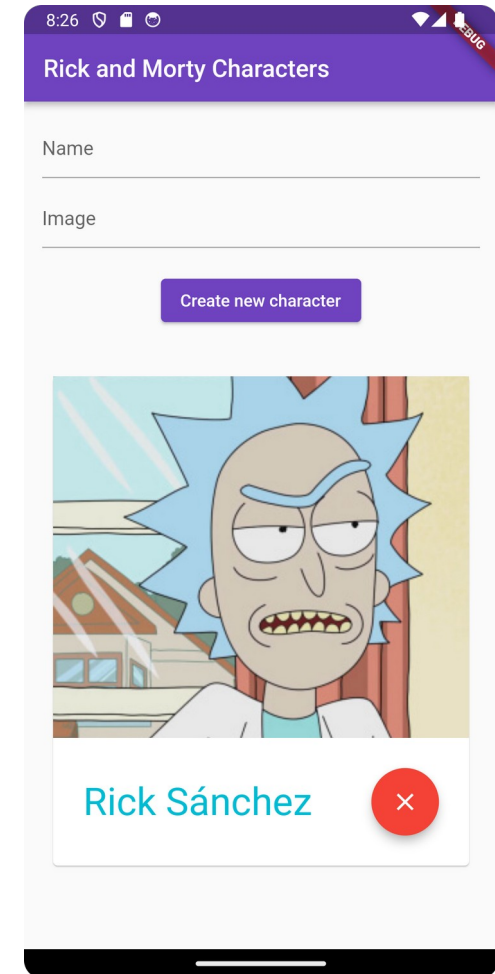
```
static Future<void> deleteCharacter(int id) async {  
    final connection = await db();  
  
    await connection.delete(  
        'Characters',  
        where: 'id = ?',  
        whereArgs: [id],  
    );  
}
```

Borraremos utilizando el parámetro **id** de Character

# Bases de datos en Flutter

## Ejercicio

- Añadiremos la funcionalidad anterior la vista (borrar personaje)
- Crearemos un botón por cada personaje
- Este botón deberá llamar a la función **deleteCharacter()** y actualizar la vista





## Más contenido

- En la asignatura no vemos en profundidad la conexión con bases de datos desde Flutter
- Existe una librería parecida a Room en Flutter (**Drift**)
  - Curva de aprendizaje más alta (generación de código)
  - Mejor manejo de entidades (no es necesario realizar parseos manuales)
  - <https://drift.simonbinder.eu/docs/getting-started/>

## Más contenido

- No vemos como trabajar con bases de datos externas
- Una aplicación **Flutter** de manera sencilla en **Firestore**, que cuenta con su propia base de datos externa (no SQLite)
- <https://docs.flutter.dev/data-and-backend/firebase>