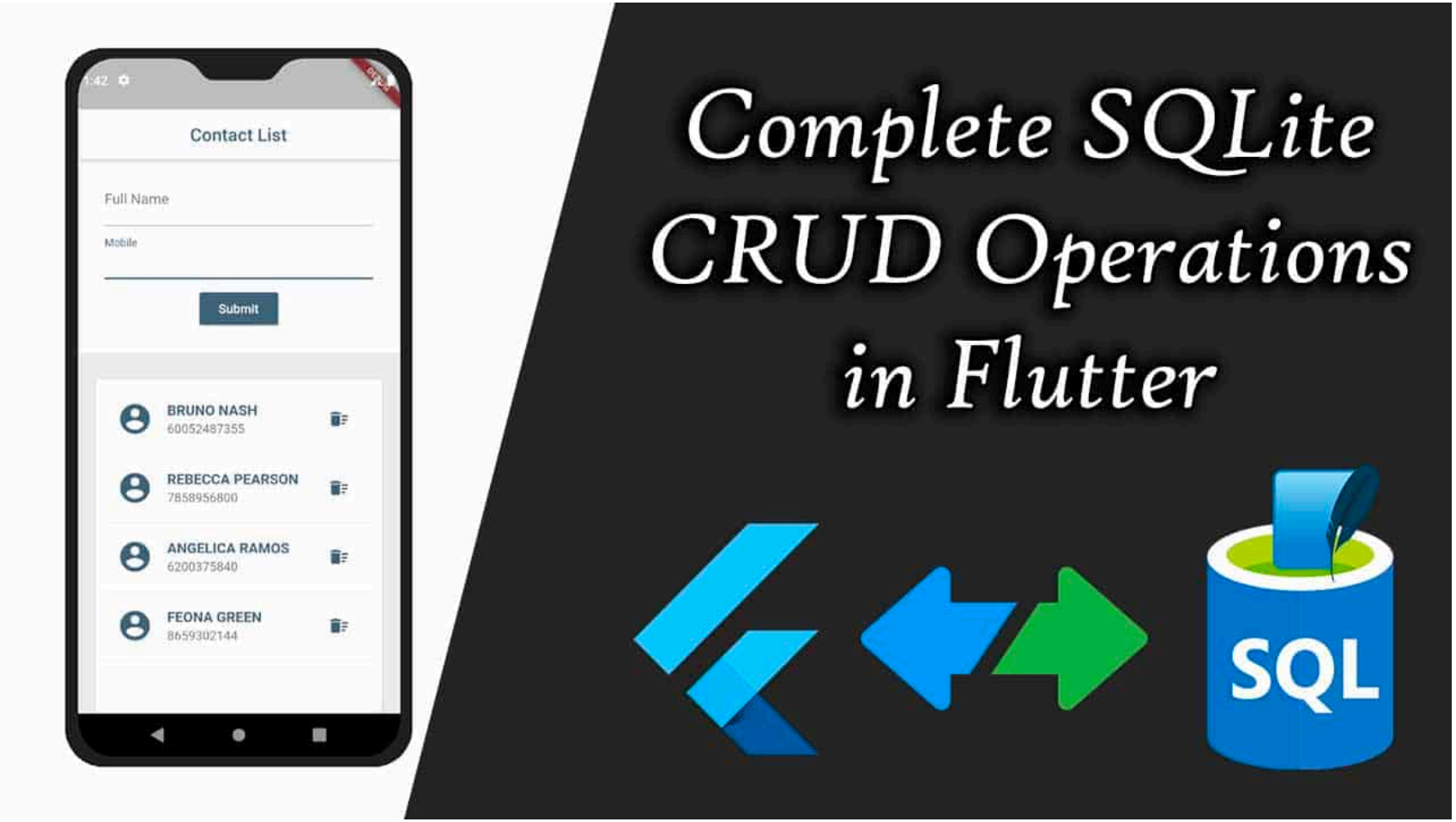
Resource Software Solution

Persist Data with SQLite

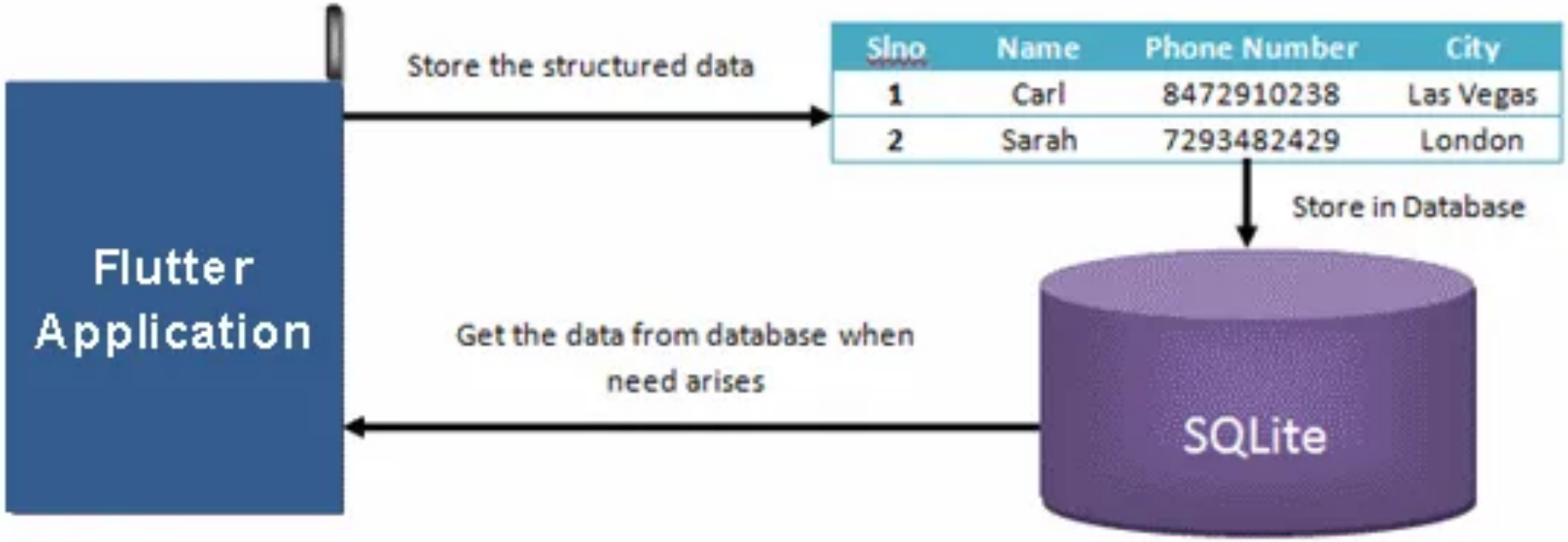
Contents

- Introduction
- CRUD statements



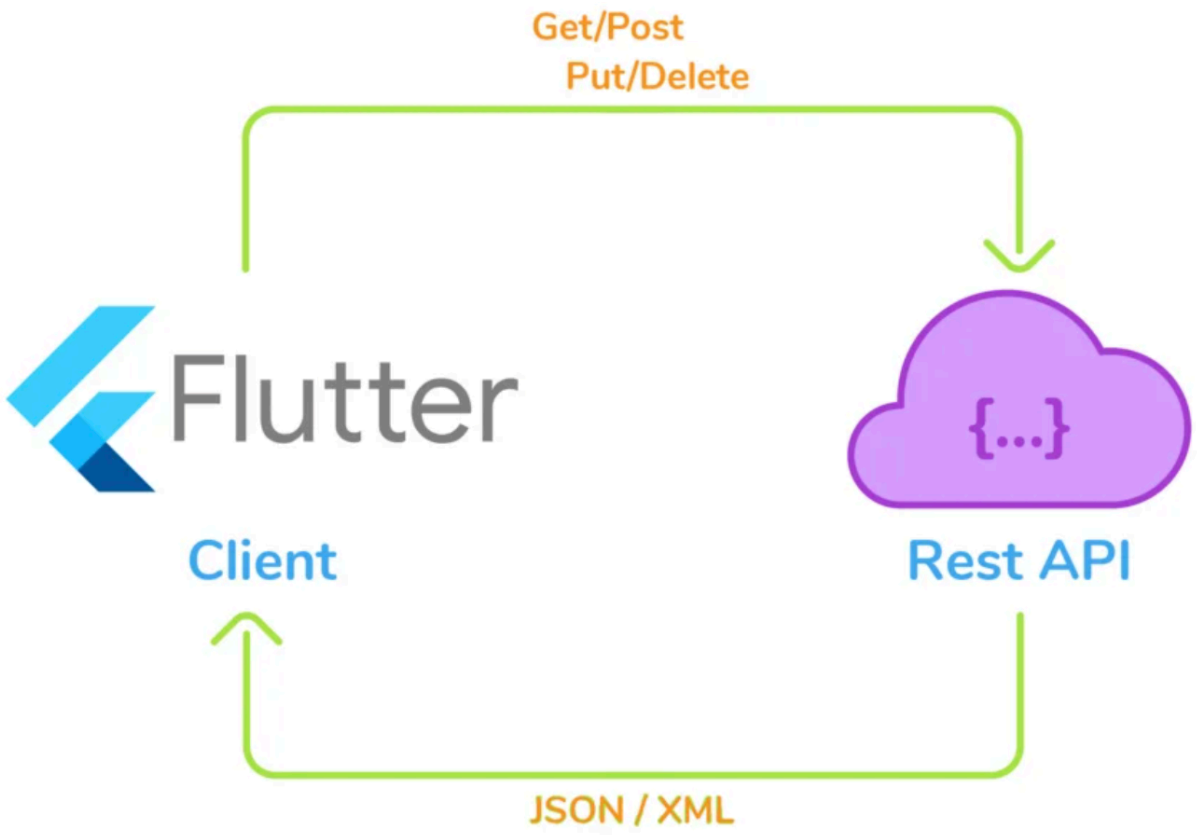
Introduction (1)

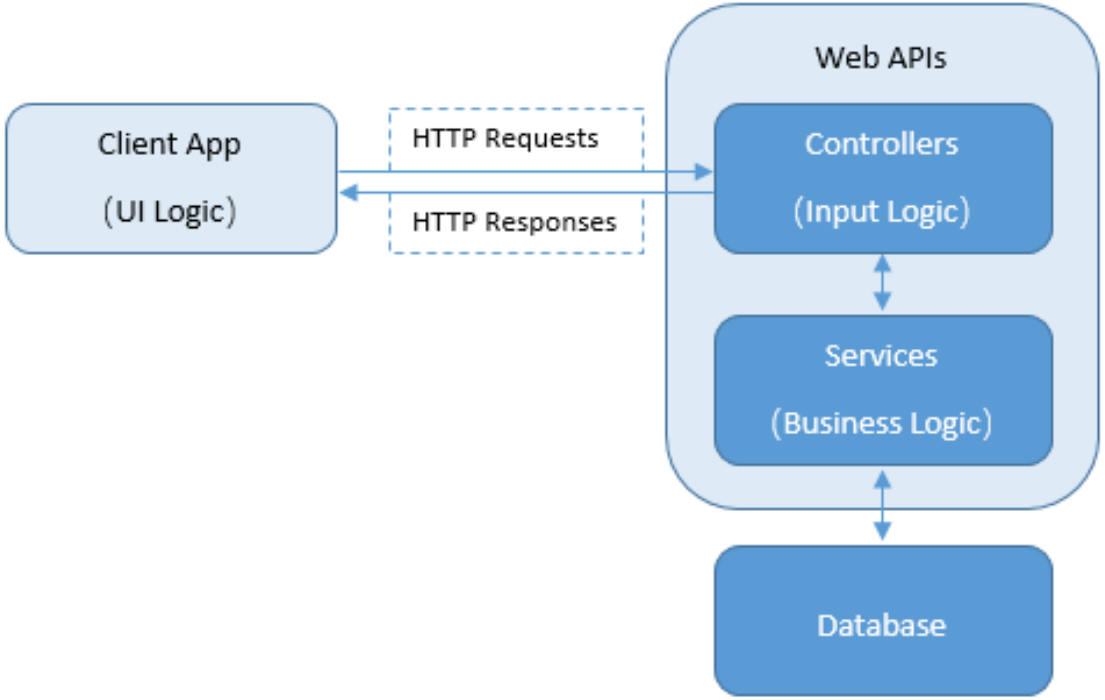
- Data on the **local** device.



Introduction (2)

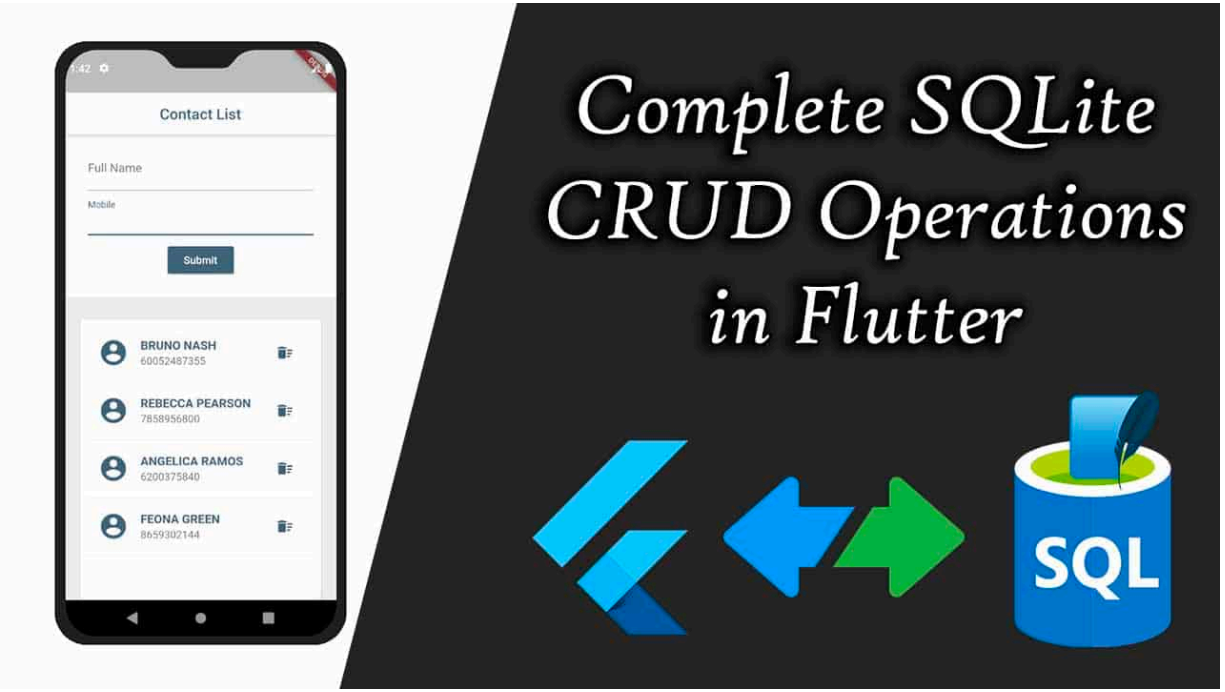
- Data on the **server**.





Introduction (3)

- Data on the **local** device.
- Steps to persist data with SQLite:
  1. Add the dependencies.
  2. Define the data model.
  3. Create the database.
  4. CRUD(Insert, Select, Update, Delete



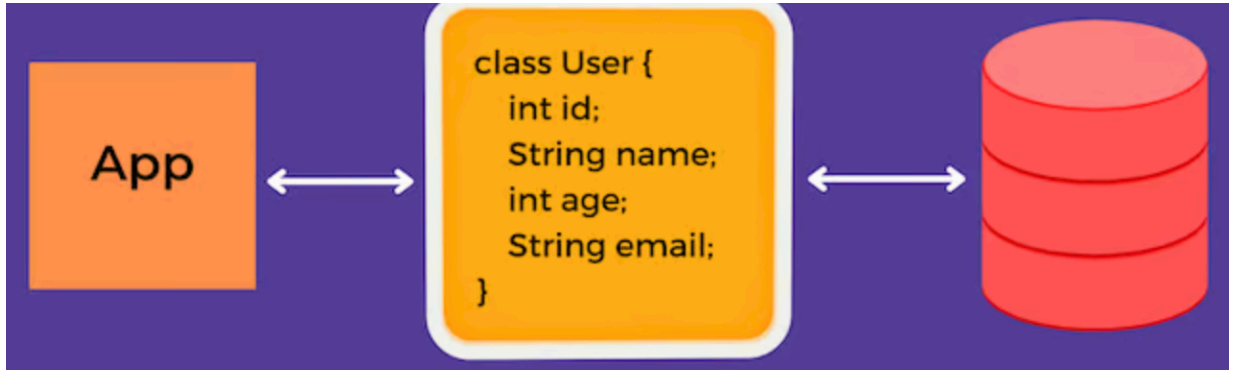
Add the Dependencies

- To work with SQLite databases, import the **sqflite** package.
- The **sqflite** package provides classes and functions to **interact with a SQLite database**.
- To add the packages as a dependency, run **flutter pub add**

```
sqflite: ^2.4.1
```

Define the Data Model

- To define the **data** that needs to be stored.
- For this example, define a **Student** class that contains three pieces of data: A unique **id**, the **name**, and the **age** of each student.




```
class Student {
  final int id;
  final String name;
  final int age;

  const Student(this.id, this.name, this.age);

  Map<String, dynamic> toMap() {
    return {
      'id' : id,
      'name' : name,
      'age' : age,
    };
  }
}
```

Create the Database

- Before reading and writing data to the database, open a connection to the database with the **openDatabase()** function.

```
final database = openDatabase(
  'demo.db',
   version: 1,
  // When the database is first created, create a table to store student.
  onCreate: (Database database, int version) async {
    await createItemTable(database);
    await createAccountTable(database);
  },
   [onUpgrade: (Database database, int oldVersion, int newVersion) async {

   }]
);
```

CREATE TABLE Statement (1)

- To create a new table in SQLite, you use **CREATE TABLE** statement using the following syntax:

```
CREATE TABLE table_name (
    column_1 data_type PRIMARY KEY,
    column_2 data_type NOT NULL,
    column_3 data_type DEFAULT VALUE,
  ...
);
```

- Datatypes in SQLite:
  1. **REAL** is a **floating point value**.
  2. **INTEGER**: Use this to define columns that store integer values
  3. **TEXT** is a **string**.
  4. **BLOB** (BLOB stands for a binary large object) to store any binary data into the SQLite table. Binary can be a **file, image, video, or a media**.
  5. **TIMESTAMP** is a temporal data type that holds the combination of **date and time**. The format of a TIMESTAMP is **YYYY-MM-DD HH:MM:SS** which is fixed at 19 characters.

CREATE TABLE Statement (2)

- Example

```
CREATE TABLE student (
   id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   name TEXT,
   age INTEGER,
   create_at timestamp default current_timestamp
 )
```

Insert Statement (1)

- This method helps insert a map of values into the specified table and returns the id of the last inserted row.

```
// Define a function that inserts student into the database
Future<void> insertStudent(Student student) async {
  // Get a reference to the database.
  final db = await database;

  await db.insert(
    'student',
    student.toMap(),
  );
}
```

```
class Student {
  final int id;
  final String name;
  final int age;

  const Student(this.id, this.name, this.age);

  Map<String, dynamic> toMap() {
    return {
      'id' : id,
      'name' : name,
      'age' : age,
    };
  }
}
```

```
Future<int> insert(
  String table,
  Map<String, Object?> values, {
  String? nullColumnHack,
  ConflictAlgorithm? conflictAlgorithm,
})
```

nullColumnHack, you want to insert an empty row into a table student(id, name), which id is auto generated and name is null.
-> insert('student', student.toMap(), 'name')

The **ConflictAlgorithm** allows you to define how to handle these conflicts.

Insert Statement (2)

- Use the **insert()** method to store the Map in the student table.

```
Future<void> insertUsersWithFail(List<Map<String, Object?>> users) async {
 for (var user in users) {
  try {
   await db.insert(
    'User',
    user,
    conflictAlgorithm: ConflictAlgorithm.
   );
  } catch (e) {
   // Handle the conflict error
   print('Insert failed due to conflict: $e')
  }
 }
}
```

[
User1,
User2,
User3,
]

conflicts

ConflictAlgorithm.rollback: This algorithm will undo any changes made during the transaction up to the point of the conflict.
ConflictAlgorithm.abort: This stops the transaction immediately, but any changes made before the conflict within the transaction are preserved. (**default**)
ConflictAlgorithm.fail: that specific user is not inserted, and an error is raised.
ConflictAlgorithm.ignore: No error is returned, but the conflicting row is not inserted.
ConflictAlgorithm.replace: The new row data replaces the existing row data that caused the conflict.

Select Statement

- Use the **query()** method. This returns a **List<Map>**

```
// A method that retrieves all the students from the student table.
Future<List<Map<String, dynamic>> getStudents() async {
  // Get a reference to the database.
  final db = await database;

  // Query the table for all the Students.
  return db.query('student');
}
```

```
Future<List<Map<String, Object?>>> query(
  String table, {
  bool? distinct,
  List<String>? columns,
  String? where,
  List<Object?>? whereArgs,
  String? groupBy,
  String? having,
  String? orderBy,
  int? limit,
  int? offset,
})
```

Update Statement

- This method for updating rows in the database. Returns the number of changes made

```
Future<void> updateStudent(Student student) async {
  // Get a reference to the database.
  final db = await database;

  await db.update(
    'student',
    student.toMap(),
    where: 'id = ?',
    whereArgs: [student.id],
  );
}
```

```
Future<int> update(
  String table,
  Map<String, Object?> values, {
  String? where,
  List<Object?>? whereArgs,
  ConflictAlgorithm? conflictAlgorithm,
})
```

Delete Statement

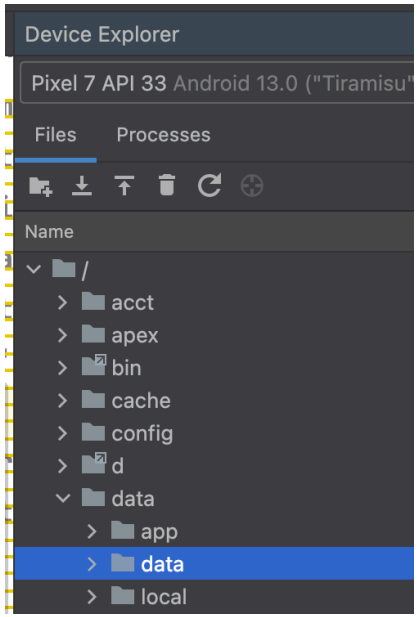
- This method for deleting rows in the database. Returns the number of rows affected.

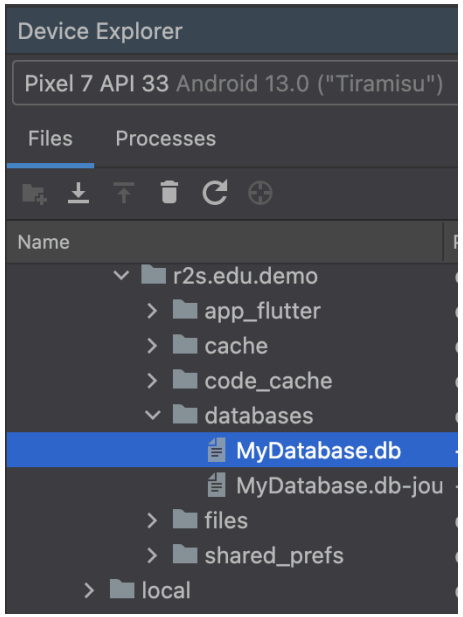
```
Future<void> deleteStudent(int id) async {
  // Get a reference to the database.
  final db = await database;

  // Remove the Student from the database.
  await db.delete(
    'student',
    where: 'id = ?',
    // Pass the Student's id as a whereArg.
    whereArgs: [id],
  );
}
```

```
Future<int> delete(
  String table, {
  String? where,
  List<Object?>? whereArgs,
})
```

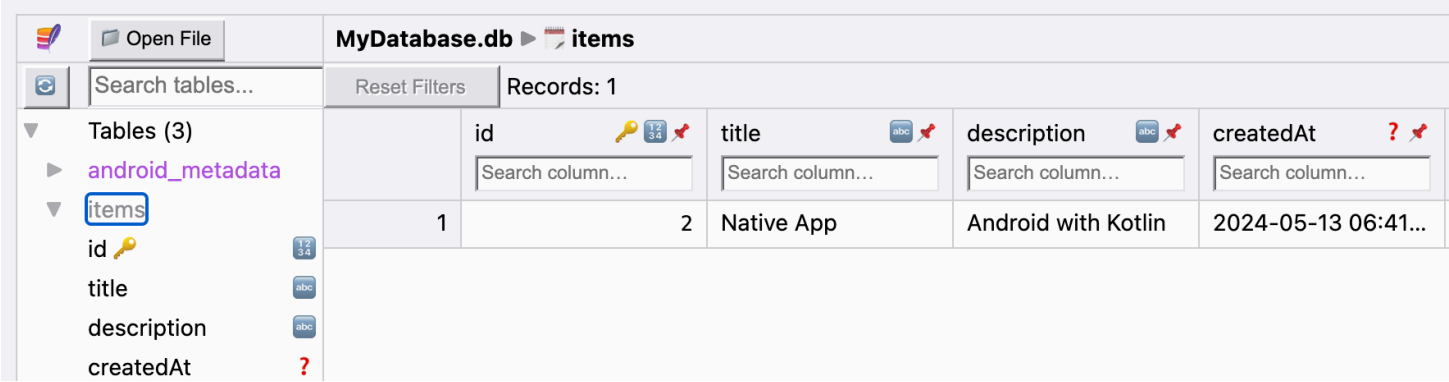
Check Database

- Using Device Explorer (View -> Tool Windows -> Device Explorer)





- https://sqliteviewer.app/



*Keeping up those **inspiration** and the **enthusiasm** in the **learning path**. Let confidence to bring it into **your career path** for getting gain the **success** as your expectation.*

Thank you

Contact

- **Name: R2S Academy**
- **Email: daotao@r2s.edu.vn**
- **Phone/Zalo: 0919 365 363**
- **FB: https://www.facebook.com/r2s.tuyendung**
- **Website: www.r2s.edu.vn**

Questions and Answers