Lab Manual

Mobile Computing

BSIT 5A

SQLite Database with Flutter

Getting Started with SQLite in Flutter

SQLite is a lightweight, file-based database widely used in mobile apps to store data locally without needing internet. In Flutter, the most popular plugin to work with SQLite is **SQFlite**. This package provides a Dart API for interacting with the SQLite database.
It offers functionalities for:

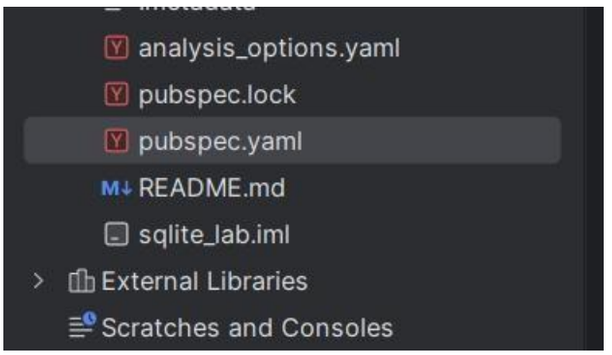
- Creating and opening databases
- Defining tables and columns
- Performing CRUD operations (Create, Read, Update, Delete) on data
- Querying the database to retrieve specific information

Lab Objective:

In this lab, students will learn how to integrate SQFlite in a Flutter application and perform **CRUD Operations** (Create, Read, Update, Delete) on a local SQLite database.

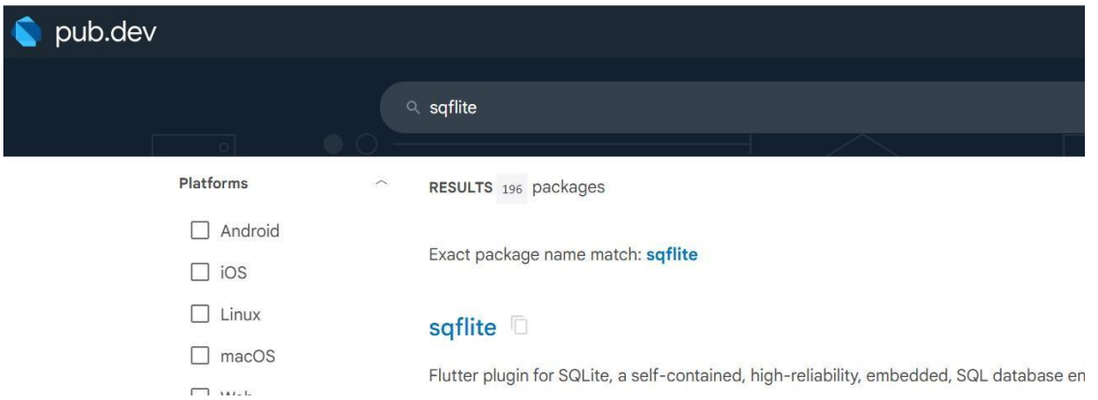
Step#1: Adding dependencies

In order to use SQLite in Flutter we need two dependencies i.e. sqflite and path. Add these dependencies in the pubspec.yaml file.




```yaml
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.4.1   # SQLite package for database operations
  path_provider: ^2.1.5  # Path provider for storing database
```

Where to get right versions from?

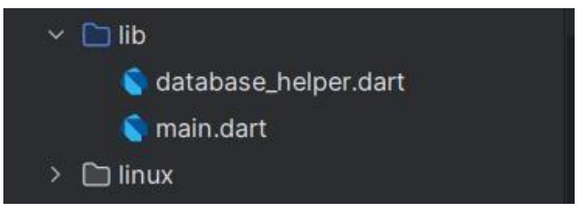


The sqflite package provides functions to insert, query, update, and delete data from the database, making it easy to manage structured data offline within your app.

path_provider: ^2.1.5

This package helps locate commonly used directories on the device file system, such as the app's documents directory or temporary directory.
It's useful when you need to store files locally, including databases like SQLite, so your app can access them later.
path_provider helps find appropriate file paths on different platforms (iOS, Android) to store files securely and consistently.

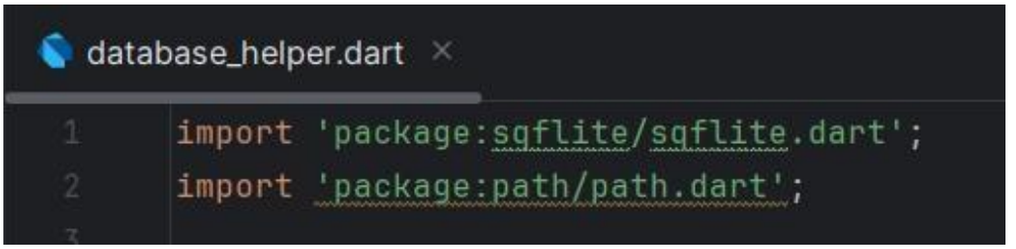
Step#2: Create database_helper.dart file inside lib directory



The database_helper.dart file is a utility file used to manage all the interactions with the database in a Flutter project, specifically when working with SQLite. It encapsulates all the database-related operations into one place, making the code more organized, reusable, and easier to maintain.

1. Database Initialization

- The DatabaseHelper class is typically a singleton, ensuring there's only one instance of the database connection throughout the app. Ensures that there is only one connection to the database at any given time.
- It contains a user defined method (_initDB) that opens or creates the database and initializes it with the necessary schema (e.g., creating tables).



1. **import 'package:sqflite/sqflite.dart'**

Key Classes and Functions from this Package:

- openDatabase(): Used to open or create an SQLite database.
- Database: The main class representing the SQLite database connection, providing methods like query(), insert(), update(), and delete().
- Database.execute(): Executes an SQL statement, such as creating tables or running custom queries.

2. import 'package:path/path.dart';

This import brings in the path package, which provides utilities to handle and manipulate file paths in a platform-independent way. In Flutter, it's used to construct paths for various files, such as the SQLite database file.

Key Functions from this Package:

join(): Combines multiple segments of a path into a single file path, ensuring the correct file separators are used for the platform (e.g., \ on Windows or / on macOS/Linux).

getDatabasesPath(): Returns the default directory where the app can store its database files.

```dart
class DatabaseHelper {
  static final DatabaseHelper instance = DatabaseHelper._init();
  static Database? _database;

  DatabaseHelper._init();
```

static final DatabaseHelper instance = DatabaseHelper._init();

This line declares a static variable instance that holds a single instance of the DatabaseHelper class.
DatabaseHelper._init() calls the private constructor to initialize the instance. By making this instance static, it ensures that the class is accessed through DatabaseHelper.instance rather than creating new instances each time. This provides a global access point to the database helper.

static Database? _database;

This declares a static variable _database of type Database?, which will hold the reference to the SQLite database connection.
The ? indicates that this variable can be null initially, as the database connection is established lazily (i.e., only when it's needed).

```dart
Future<Database> get database async {
  if (_database != null) return _database!;
  _database = await _initDB('app_database.db');
  return _database!;
}
```

```dart
Future<Database> _initDB(String filePath) async {
  final dbPath = await getDatabasesPath();
  final path = join(dbPath, filePath);
  return openDatabase(path, version: 1, onCreate: _onCreate);
}
```
a

String filePath: The function takes a filePath parameter, which specifies the name of the SQLite database file (for example, 'app_database.db'). This path will be used to locate or create the database file.

getDatabasesPath(): This is a method provided by the path_provider package that returns the default directory path where database files are stored on the device.

join(): This is a function provided by the path package. It combines the dbPath (the directory path) with the filePath (the database file name) to create a full path to the database file.

openDatabase(path): This function opens the SQLite database at the provided path. If the database doesn't already exist, it will be created.

version: 1: This sets the version number of the database schema. If you need to update the schema later (for example, adding new tables or columns), you can increment the version number, and the sqflite package will automatically handle schema migrations.

onCreate: _onCreate: This is a function that gets triggered when the database is created (i.e., when it doesn’t already exist). The _onCreate function is typically used to define the initial schema, such as creating tables.

The openDatabase() **method returns a** Future<Database>, which represents the eventual database instance once the database has been successfully opened or created.

3. On Create Function

```
Future _onCreate(Database db, int version) async {
  await db.execute('''
    CREATE TABLE users(
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      name TEXT,
      address TEXT
    )
  ''');
}
```

This function is called when the database is first created. It defines the structure of the users table with columns id, name, and address.

The ''' (triple single quotes) you are referring to are used in Dart for defining multi-line strings.

```
Future<int> insertUser(Map<String, dynamic> user) async {
  final db = await instance.database;
  return await db.insert('users', user);
}
```

Future<int>: The function returns a Future<int>. This means it is an asynchronous function that will eventually return an integer. Specifically, the integer represents the number of rows affected by the insert operation (usually 1 if the insert is successful).

Map<String, dynamic> user: This parameter is a map where the key is a String (representing the column names in the database), and the value is a dynamic (representing the data to be inserted into those columns).

await instance.database: This line retrieves the Database instance from the DatabaseHelper class, using the database getter

```dart
  Future<List<Map<String, dynamic>>> fetchUsers() async {
    final db = await instance.database;
    return await db.query('users');
  }

  Future<int> deleteUser(int id) async {
    final db = await instance.database;
    return await db.delete('users', where: 'id = ?', whereArgs: [id]);
  }
}
```

fetchUsers()

This function is used to fetch all the users from the users table in the database.

Future<List<Map<String, dynamic>>>: The function returns a Future that will eventually resolve to a List<Map<String, dynamic>>. Each element in the list is a map that contains column names as keys and their corresponding values from the database rows.

Map<String, dynamic> means the keys are the column names (e.g., id, name, address), and the values are the actual data for each user.

The db.query() function is used to query the database. In this case, it queries all rows from the users table.

The query() function returns a List<Map<String, dynamic>>, where each map represents a row from the table. The keys of the map are the column names (id, name, address), and the values are the corresponding data for that row.

deleteUser()

This function deletes a user from the users table by its id.

return await db.delete('users', where: 'id = ?', whereArgs: [id]);:

db.delete(): This is the method used to delete records from the database.

'users': The table from which records will be deleted.

where: 'id = ?': The where clause specifies the condition for which rows to delete. In this case, it deletes the rows where the id column matches the specified id argument.

whereArgs: [id]: This is a list of arguments that will replace the ? placeholder in the where clause. The id passed to the function is placed in this list to complete the condition ('id = ?').

Main.dart file:

```dart
import 'package:flutter/material.dart';
import 'database_helper.dart';

void main() {
  runApp(MyApp());
}
```

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter SQLite Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),  // ThemeData
      home: MyHomePage(),
      debugShowCheckedModeBanner: false,
    );  // MaterialApp
  }
}
```

```dart
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

```dart
class _MyHomePageState extends State<MyHomePage> {
  final nameController = TextEditingController();
  final addressController = TextEditingController();
  List<Map<String, dynamic>> _users = [];
```

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

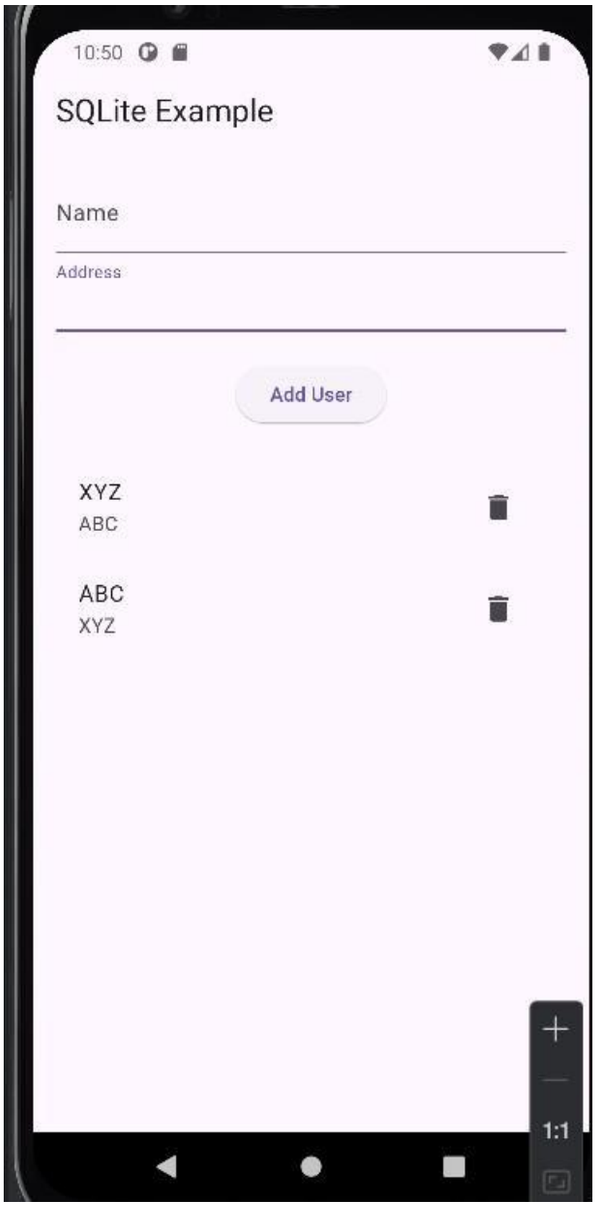
initState() is a lifecycle method called once when the widget is added to the widget tree, ideal for initializing tasks. The super.initState() call ensures that the parent class's initialization logic is executed. The _fetchUsers() method fetches data from the database when the widget is first created. This setup prepares the widget's data before it is displayed.

```dart
Future<void> _fetchUsers() async {
  final users = await DatabaseHelper.instance.fetchUsers();
  setState(() {
    _users = users;
  });
}
```

The _fetchUsers() function retrieves user records from the database and updates the state of the widget to display this data.

```dart
Future<void> _addUser() async {
  final name = nameController.text;
  final address = addressController.text;
  if (name.isNotEmpty && address.isNotEmpty) {
    await DatabaseHelper.instance.insertUser({'name': name, 'address': address});
    nameController.clear();
    addressController.clear();
    _fetchUsers();
  }
}

Future<void> _deleteUser(int id) async {
  await DatabaseHelper.instance.deleteUser(id);
  _fetchUsers();
}
```


10:50
SQLite Example
Name
Address
Add User
XYZ
ABC
ABC
XYZ
1:1

Lab Task

Imagine you are creating a note-taking application using Flutter. Each note has a title and a description.

Users should be able to:

- Create, read, and delete notes.
- Store notes locally using SQLite.

The DatabaseHelper class handles all database interactions. It initializes the database, creates the necessary table (notes), and provides methods to insert, retrieve, and delete notes.

UI Structure:

The main UI (NotesPage) displays a button to add new notes and a list of existing notes fetched from the SQLite database.

The TextEditingController objects are used for taking user input for title and description. When the "Add Note" button is pressed, a dialog appears allowing the user to enter a new note's title and description. For example:

The notes are displayed in a ListView, and each note has a delete button that removes it from the database. For example: