Lab session 4: CRUD operations with SQLite using Flutter

Task-1 Implement CRUD operations based on the base code and tutorial given

In this documentation, we will demonstrates the basics of using sqflite to insert, read, update, and remove data about various Dogs.

To do so, we will apply the following steps in a new flutter project:

1. Add the dependencies.
2. Define the Dog data model.
3. Open the database.
4. Create the dogs table.
5. Insert a Dog into the database.
6. Retrieve the list of dogs.
7. Update a Dog in the database.
8. Delete a Dog from the database.

Add the dependencies

To work with SQLite databases, we need to import the sqflite and path packages.

- The sqflite package provides classes and functions to interact with a SQLite database.
- The path package provides functions to define the location for storing the database on disk.

To add the packages as a dependency, run flutter pub add:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Dev\db_test_app> flutter pub add sqflite path
Resolving dependencies...
  flutter_lints 2.0.3 (3.0.1 available)
  lints 2.1.1 (3.0.0 available)
  matcher 0.12.16 (0.12.16+1 available)
  material_color_utilities 0.5.0 (0.8.0 available)
  meta 1.10.0 (1.11.0 available)
  path 1.8.3 (from transitive dependency to direct dependency) (1.9.0 available)
+ sqflite 2.3.1
+ sqflite_common 2.5.0+2
+ synchronized 3.1.0+1
  test_api 0.6.1 (0.7.0 available)
```

Define the Dog data model.

Before creating the table to store information on Dogs, we will take a few moments to define the data that needs to be stored.

For this example, we will define a Dog class that contains three pieces of data: A unique id, the name, and the age of each dog.

```dart
lib > models.dart > Dog
class Dog {
  final int id;
  final String name;
  final int age;

  const Dog({
    required this.id,
    required this.name,
    required this.age,
  });
}
```

Open the database.

Before reading and writing data to the database, we need to open a connection to the database. This involves two steps:

- Define the path to the database file using **getDatabasesPath()** from the sqflite package, combined with the join function from the path package.
- Open the database with the **openDatabase()** function from sqflite.

```dart
Run | Debug | Profile
void main() async {
  // Avoid errors caused by flutter upgrade.
  // Importing 'package:flutter/widgets.dart' is required.
  WidgetsFlutterBinding.ensureInitialized();
  // Open the database and store the reference.
  final database = openDatabase(
    // Set the path to the database. Note: Using the `join` function from the
    // `path` package is best practice to ensure the path is correctly
    // constructed for each platform.
    join(await getDatabasesPath(), 'doggie_database.db'),
  );
}
```

Create the dogs table.

Next, we will create a table to store information about various Dogs.

For this example, we will create a table called dogs that defines the data that can be stored. Each Dog contains an id, name, and age. Therefore, these are represented as three columns in the dogs table.

- The id is a Dart int, and is stored as an INTEGER SQLite Datatype. It is also good practice to use an id as the primary key for the table to improve query and update times.
- The name is a Dart String, and is stored as a TEXT SQLite Datatype.
- The age is also a Dart int, and is stored as an INTEGER Datatype.

```dart
  // When the database is first created, create a table to store dogs.
  onCreate: (db, version) {
    // Run the CREATE TABLE statement on the database.
    return db.execute(
      'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
    );
  },
  // Set the version. This executes the onCreate function and provides a
  // path to perform database upgrades and downgrades.
  version: 1,
);
```

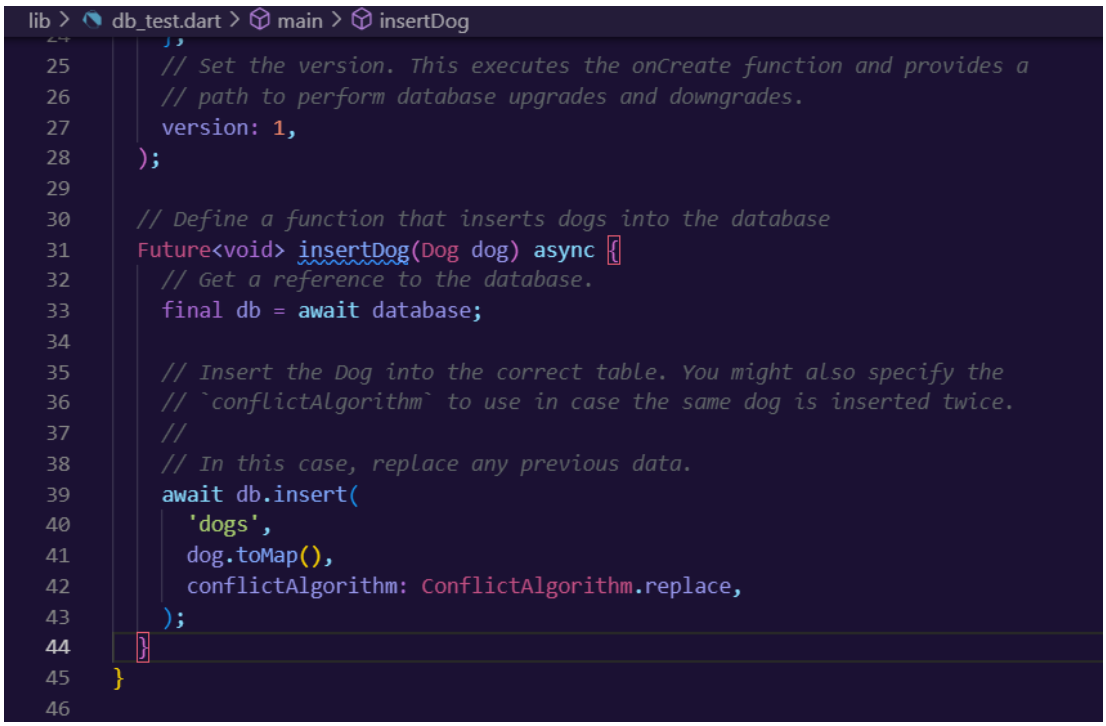
Insert a Dog into the database.

Now that we have a database with a table suitable for storing information about various dogs, it's time to read and write data.

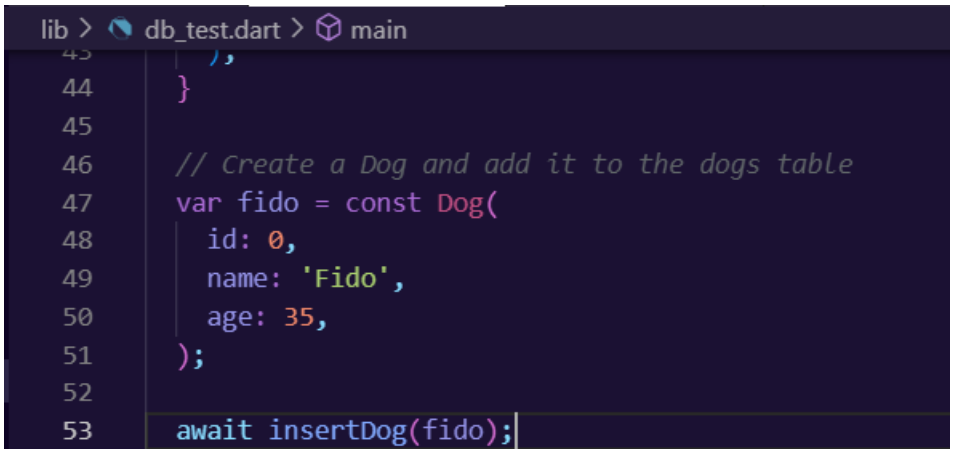
First,we will insert a Dog into the dogs table. This involves two steps:

- Convert the Dog into a Map

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



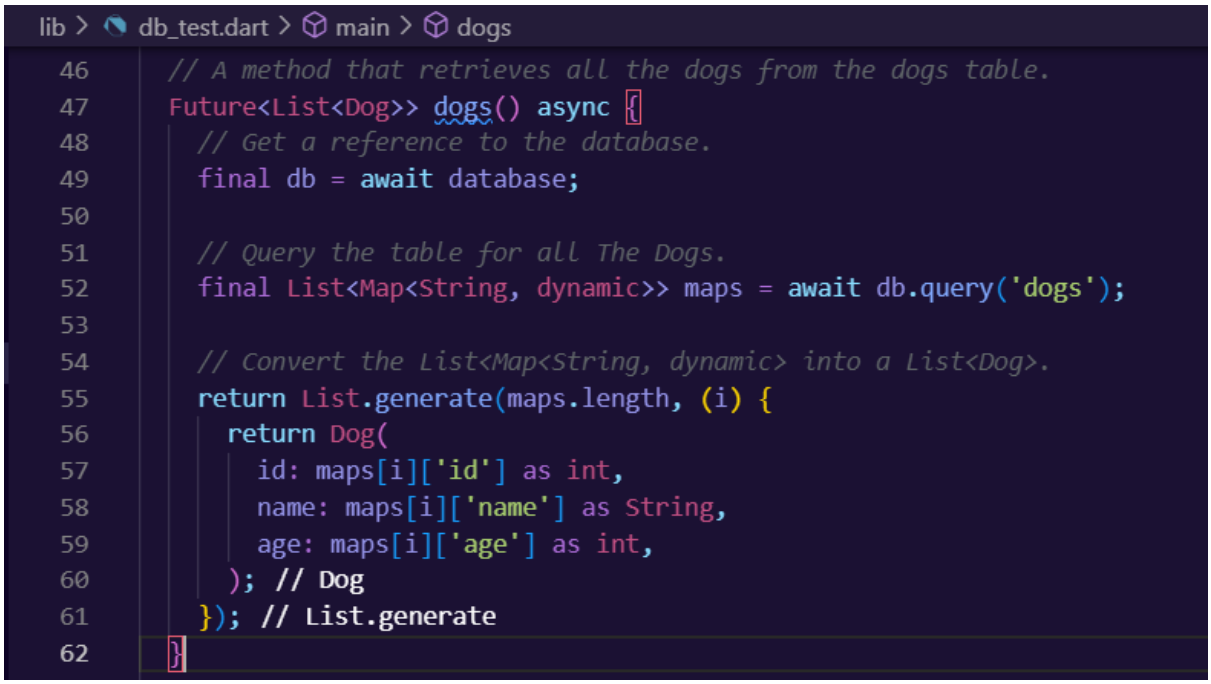
We will insert the first entry in the database with the following lines:



Retrieve the list of dogs.

Now that a Dog is stored in the database, we will query the database for a specific dog or a list of all dogs. This involves two steps:

- Run a query against the dogs table. This returns a List<Map>.
- Convert the List<Map> into a List<Dog>.



Update a Dog in the database.

After inserting information into the database, we might want to update that information at a later time. We can do this by using the **update()** method from the sqflite library.

This involves two steps:

- Convert the Dog into a Map.
- Use a where clause to ensure to update the correct Dog.

```dart
lib > db_test.dart > main > updateDog
61      }); // List.generate
62    }
63
64    Future<void> updateDog(Dog dog) async {
65      // Get a reference to the database.
66      final db = await database;
67
68      // Update the given Dog.
69      await db.update(
70        'dogs',
71        dog.toMap(),
72        // Ensure that the Dog has a matching id.
73        where: 'id = ?',
74        // Pass the Dog's id as a whereArg to prevent SQL injection.
75        whereArgs: [dog.id],
76      );
77    }
```

Using whereArgs to pass arguments to a where statement instead of using string interpolation, such as where: "**id = ${dog.id}**". This helps safeguard against SQL injection attacks.

The following lines of code will update the age of the dog and print the updated dog info in the console:

```dart
lib > db_test.dart > main
91    // Update Fido's age and save it to the database.
92    fido = Dog(
93      id: fido.id,
94      name: fido.name,
95      age: fido.age + 7,
96    );
97    await updateDog(fido);
98
99    // Print the updated results.
100   print(await dogs()); // Prints Fido with age 42.
101 }
```

Delete a Dog from the database.

In addition to inserting and updating information about Dogs, we can also remove dogs from the database. To delete data, we use the delete() method from the sqflite library.

In this section, we will create a function that takes an id and deletes the dog with a matching id from the database. To make this work, we must provide a where clause to limit the records being deleted.

```dart
lib > db_test.dart > main > deleteDog
77    }
78
79    Future<void> deleteDog(int id) async {
80      // Get a reference to the database.
81      final db = await database;
82
83      // Remove the Dog from the database.
84      await db.delete(
85        'dogs',
86        // Use a `where` clause to delete a specific dog.
87        where: 'id = ?',
88        // Pass the Dog's id as a whereArg to prevent SQL injection.
89        whereArgs: [id],
90      );
91    }
```

This part of the code will call the delete function with the id of the entry we want to delete and print the content of the table in the console.

Testings

The final code should be like this:

```dart
import 'dart:async';

import 'package:flutter/widgets.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
import 'models.dart';

void main() async {
  // Avoid errors caused by flutter upgrade.
  // Importing 'package:flutter/widgets.dart' is required.
  WidgetsFlutterBinding.ensureInitialized();
  // Open the database and store the reference.
  final database = openDatabase(
    // Set the path to the database. Note: Using the `join` function from the
    // `path` package is best practice to ensure the path is correctly
    // constructed for each platform.
    join(await getDatabasesPath(), 'doggie_database.db'),
    // When the database is first created, create a table to store dogs.
    onCreate: (db, version) {
      // Run the CREATE TABLE statement on the database.
      return db.execute(
        'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
      );
    },
    // Set the version. This executes the onCreate function and provides a
    // path to perform database upgrades and downgrades.
    version: 1,
  );

  // Define a function that inserts dogs into the database
  Future<void> insertDog(Dog dog) async {
    // Get a reference to the database.
    final db = await database;
```

```
  // Insert the Dog into the correct table. You might also specify the
  // `conflictAlgorithm` to use in case the same dog is inserted twice.
  //
  // In this case, replace any previous data.
  await db.insert(
    'dogs',
    dog.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

// A method that retrieves all the dogs from the dogs table.
Future<List<Dog>> dogs() async {
  // Get a reference to the database.
  final db = await database;

  // Query the table for all The Dogs.
  final List<Map<String, dynamic>> maps = await db.query('dogs');

  // Convert the List<Map<String, dynamic> into a List<Dog>.
  return List.generate(maps.length, (i) {
    return Dog(
      id: maps[i]['id'] as int,
      name: maps[i]['name'] as String,
      age: maps[i]['age'] as int,
    );
  });
}

Future<void> updateDog(Dog dog) async {
  // Get a reference to the database.
  final db = await database;

  // Update the given Dog.
  await db.update(
    'dogs',
    dog.toMap(),
    // Ensure that the Dog has a matching id.
    where: 'id = ?',
    // Pass the Dog's id as a whereArg to prevent SQL injection.
    whereArgs: [dog.id],
  );
}

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

```
    // Remove the Dog from the database.
    await db.delete(
      'dogs',
      // Use a `where` clause to delete a specific dog.
      where: 'id = ?',
      // Pass the Dog's id as a whereArg to prevent SQL injection.
      whereArgs: [id],
    );
  }

  // Create a Dog and add it to the dogs table
  var fido = const Dog(
    id: 0,
    name: 'Fido',
    age: 35,
  );

  await insertDog(fido);

  // Now, use the method above to retrieve all the dogs.
  print(await dogs()); // Prints a list that include Fido.

  // Update Fido's age and save it to the database.
  fido = Dog(
    id: fido.id,
    name: fido.name,
    age: fido.age + 7,
  );
  await updateDog(fido);

  // Print the updated results.
  print(await dogs()); // Prints Fido with age 42.

  // Delete Fido from the database.
  await deleteDog(fido.id);

  // Print the list of dogs (empty).
  print(await dogs());
}
```

We have created the model in the models.dart file for better cpde organization:

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

```dart
  const Dog({
    required this.id,
    required this.name,
    required this.age,
  });

  // Convert a Dog into a Map. The keys must correspond to the names of the
  // columns in the database.
  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'name': name,
      'age': age,
    };
  }

  // Implement toString to make it easier to see information about
  // each dog when using the print statement.
  @override
  String toString() {
    return 'Dog{id: $id, name: $name, age: $age}';
  }
}
```

As we did not create any UI interface, we will observe the result in the console upon running the file:

```
PS D:\Dev\db_test_app> flutter run lib/db_test.dart
Launching lib/db_test.dart on sdk gphone64 x86 64 in debug mode...
Running Gradle task 'assembleDebug'...                             5.4s
√  Built build\app\outputs\flutter-apk\app-debug.apk.
Installing build\app\outputs\flutter-apk\app-debug.apk...       1,785ms
I/flutter (12252): [Dog{id: 0, name: Fido, age: 35}]
I/flutter (12252): [Dog{id: 0, name: Fido, age: 42}]
I/flutter (12252): []
Syncing files to device sdk gphone64 x86 64...                     32ms

Flutter run key commands.
r Hot reload.
```

We can distinguish the 3 prints we have established after inserting, updating and deleting the data.

Conclusion

Implementing CRUD operations in a Flutter app was streamlined through the utilization of fundamental Sqflite commands. The Sqflite package provided a convenient interface for creating, reading, updating, and deleting records in the app's SQLite database. This simplicity allowed for efficient management of data, enhancing the overall development experience.