10 лабораторная работа

Тема: Flutter: Работа с базами данных в Android Studio с использованием исходных команд SQL SELECT, INSERT, UPDATE, DELETE

Цель работы: Работа с базой данных и использование исходных команд.

SQLite это наиболее популярный способ для хранения данных на мобильных устройствах. В этой статье мы будем использовать пакет sqflite для использования SQLite. Sqflite — одна из наиболее часто используемых и актуальных библиотек для подключения SQLite базы данных в Flutter.

1. Добавление зависимостей

В нашем проекте открываем файл **pubspec.yaml**. Под зависимостями добавляем последнюю версию sqflite и path_provider.

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: any
  path_provider: any
```

2. Создадим DB Client

Теперь создадим новый файл Database.dart. В нем создадим синглтон.

Почему нам нужен синглтон: мы используем этот патерн для уверенности в том что у нас есть только одна сущность класса и для обеспечения глобальной точки входа в него.

1. Создадим приватный конструктор, который может использоваться только внутри этого класса

```
class DBProvider {
  DBProvider._();
```

```
  static final DBProvider db = DBProvider._();
}
```

2. Настроим базу данных

Следующим шагом будет создания объекта базы данных и предоставим геттер, где мы создадим объект базы данных, если он еще не был создан (ленивая инициализация)

```
static Database _database;
  Future<Database> get database async {
    if (_database != null)
    return _database;

    // if _database is null we instantiate it
    _database = await initDB();
    return _database;
  }
```

Если нет объекта, присвоенного базе данных, то мы вызовем функцию initDB для создания базы данных. В этой функции мы получим путь для сохранения базы данных и создания желаемых таблиц

```
initDB() async {
    Directory documentsDirectory = await getApplicationDocumentsDirectory();
    String path = join(documentsDirectory.path, "TestDB.db");
    return await openDatabase(path, version: 1, onOpen: (db) {
    }, onCreate: (Database db, int version) async {
      await db.execute("CREATE TABLE Client ("
          "id INTEGER PRIMARY KEY,"
          "first_name TEXT,"
          "last_name TEXT,"
          "blocked BIT"
          ")");
```

```
    });
  }
```

3. Создадим класс модели

Данные внутри базы данных будут конвертироваться в Dart Maps. Нам необходимо создать классы моделей с toMap и fromMap методами.

Для создания классов моделей, я собираюсь использовать этот сайт

Наша модель:

```
/// ClientModel.dart
import 'dart:convert';

Client clientFromJson(String str) {
    final jsonData = json.decode(str);
    return Client.fromJson(jsonData);
}

String clientToJson(Client data) {
    final dyn = data.toJson();
    return json.encode(dyn);
}

class Client {
    int id;
    String firstName;
    String lastName;
    bool blocked;

    Client({
        this.id,
        this.firstName,
        this.lastName,
```

```
        this.blocked,
    });

    factory Client.fromJson(Map<String, dynamic> json) => new Client(
        id: json["id"],
        firstName: json["first_name"],
        lastName: json["last_name"],
        blocked: json["blocked"],
    );

    Map<String, dynamic> toJson() => {
        "id": id,
        "first_name": firstName,
        "last_name": lastName,
        "blocked": blocked,
    };
}
```

4. CRUD operations

Create

Используя rawInsert:

```
newClient(Client newClient) async {
    final db = await database;
    var res = await db.rawInsert(
      "INSERT Into Client (id,first_name)"
      " VALUES (${newClient.id},${newClient.firstName})");
    return res;
  }
```

Используя insert:

```
newClient(Client newClient) async {
    final db = await database;
    var res = await db.insert("Client", newClient.toMap());
    return res;
  }
```

Другой пример с использованием большого ID в качестве нового ID

```
newClient(Client newClient) async {
    final db = await database;
    //get the biggest id in the table
    var table = await db.rawQuery("SELECT MAX(id)+1 as id FROM Client");
    int id = table.first["id"];
    //insert to the table using the new id
    var raw = await db.rawInsert(
        "INSERT Into Client (id,first_name,last_name,blocked)"
        " VALUES (?,?,?,?)",
        [id, newClient.firstName, newClient.lastName, newClient.blocked]);
    return raw;
  }
```

Read

Get Client by id

```
getClient(int id) async {
    final db = await database;
    var res =await  db.query("Client", where: "id = ?", whereArgs: [id]);
    return res.isNotEmpty ? Client.fromMap(res.first) : Null ;
  }
```

Get all Clients with a condition

```
getAllClients() async {
    final db = await database;
    var res = await db.query("Client");
    List<Client> list =
        res.isNotEmpty ? res.map((c) => Client.fromMap(c)).toList() : [];
    return list;
  }
```

Получить только заблокированных клиентов

```
getBlockedClients() async {
    final db = await database;
    var res = await db.rawQuery("SELECT * FROM Client WHERE blocked=1");
    List<Client> list =
        res.isNotEmpty ? res.toList().map((c) => Client.fromMap(c)) : null;
    return list;
  }
```

Update

Update an existing Client

```
updateClient(Client newClient) async {
    final db = await database;
    var res = await db.update("Client", newClient.toMap(),
        where: "id = ?", whereArgs: [newClient.id]);
    return res;
  }
```

Блокировка/разблокировка клиента

```
blockOrUnblock(Client client) async {
    final db = await database;
    Client blocked = Client(
        id: client.id,
        firstName: client.firstName,
        lastName: client.lastName,
        blocked: !client.blocked);
    var res = await db.update("Client", blocked.toMap(),
        where: "id = ?", whereArgs: [client.id]);
    return res;
  }
```

Delete

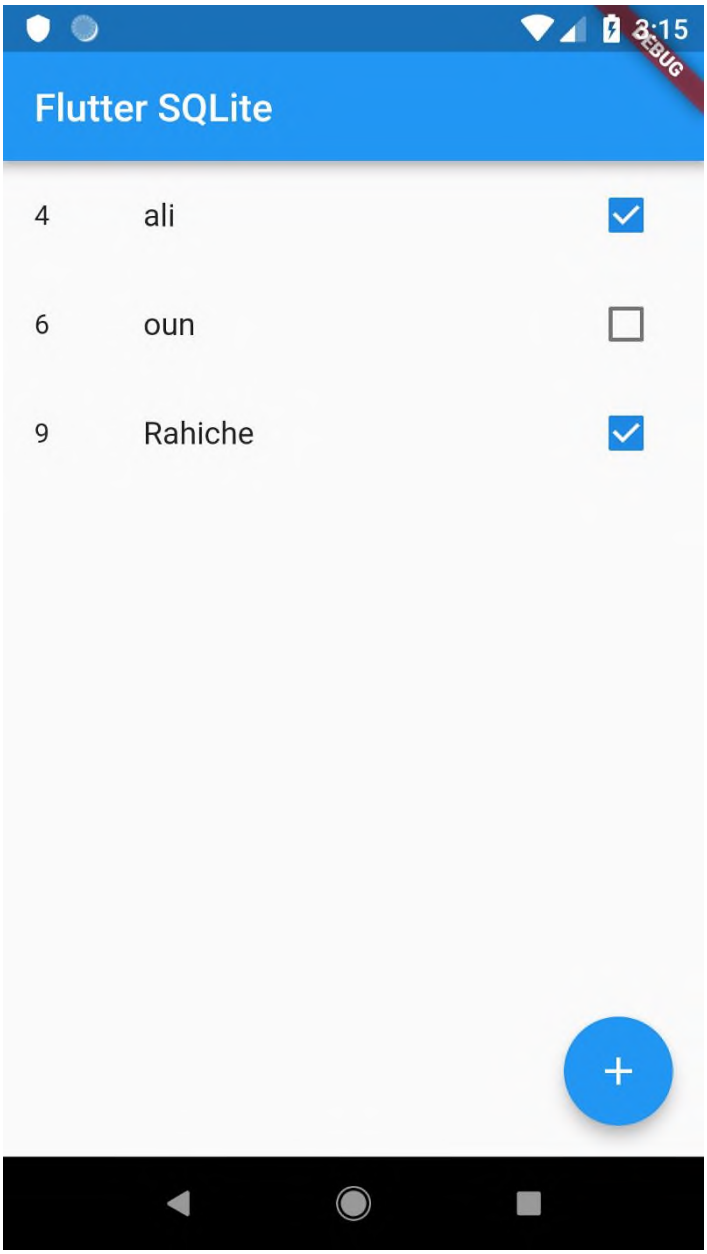
Delete one Client

```
deleteClient(int id) async {
    final db = await database;
    db.delete("Client", where: "id = ?", whereArgs: [id]);
  }
```

Delete All Clients

```
deleteAll() async {
    final db = await database;
    db.rawDelete("Delete * from Client");
  }
```

Demo



Для нашего демо мы создадим простое приложение, отображающее нашу базу данных.

Для начала сверстаем экран

```
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Flutter SQLite")),
      body: FutureBuilder<List<Client>>(
        future: DBProvider.db.getAllClients(),
```

```
      builder: (BuildContext context, AsyncSnapshot<List<Client>> snapshot) {
        if (snapshot.hasData) {
          return ListView.builder(
            itemCount: snapshot.data.length,
            itemBuilder: (BuildContext context, int index) {
              Client item = snapshot.data[index];
              return ListTile(
                title: Text(item.lastName),
                leading: Text(item.id.toString()),
                trailing: Checkbox(
                  onChanged: (bool value) {
                    DBProvider.db.blockClient(item);
                    setState(() {});
                  },
                  value: item.blocked,
                ),
              );
            },
          );
        } else {
          return Center(child: CircularProgressIndicator());
        }
      },
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.add),
      onPressed: () async {
        Client rnd = testClients[math.Random().nextInt(testClients.length)];
        await DBProvider.db.newClient(rnd);
        setState(() {});
      },
    ),
  );
}
```

Заметки:

1. FutureBuilder используется для получения данных из бд

2. FAB для инициализации тестовых клинетов

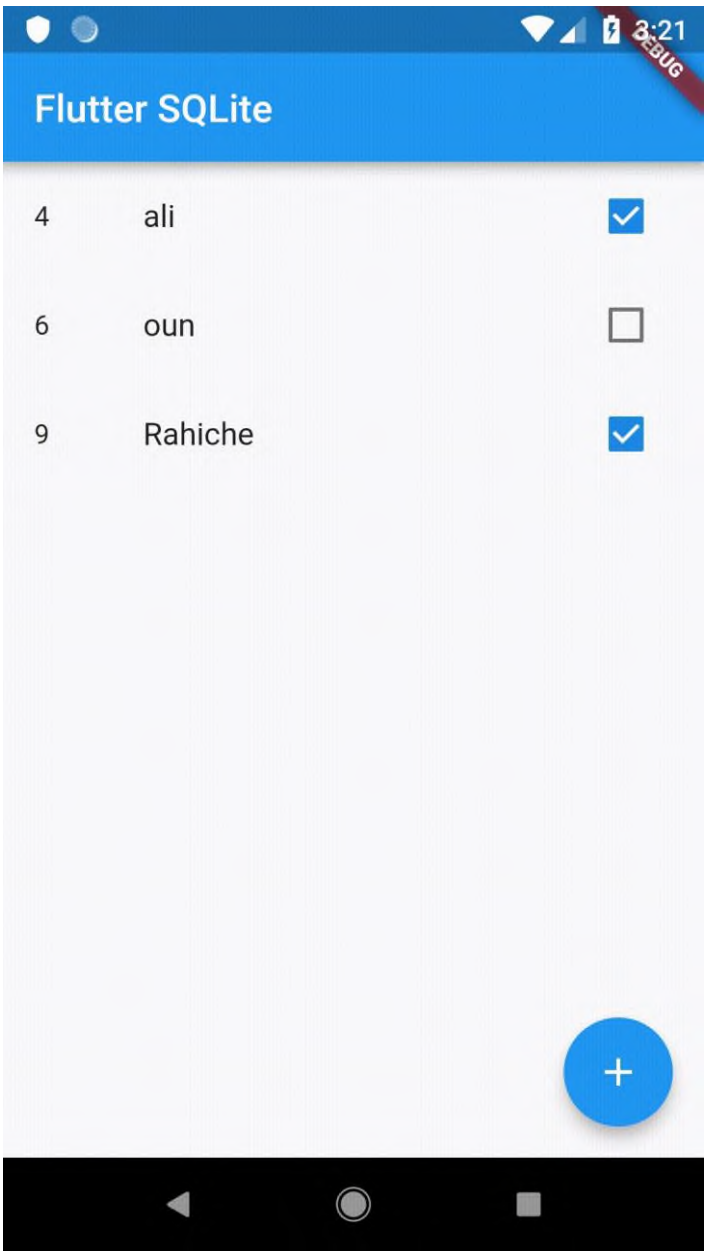
```
List<Client> testClients = [
    Client(firstName: "Raouf", lastName: "Rahiche", blocked: false),
    Client(firstName: "Zaki", lastName: "oun", blocked: true),
    Client(firstName: "oussama", lastName: "ali", blocked: false),
  ];
```

3. CircularProgressIndicator показывается, когда нет данных

4. Когда пользователь кликает по чекбоксам клиент блокируется/разблокируется

Теперь очень просто добавлять новые фичи, например если мы хотим удалить клиента, в момент когда он свайпнут, просто оберните ListTile в Dismissible widget вот так:

```
return Dismissible(
   key: UniqueKey(),
   background: Container(color: Colors.red),
   onDismissed: (direction) {
    DBProvider.db.deleteClient(item.id);
   },
    child: ListTile(...),
  );
```



Рефакторинг для использования BLoC паттерна

Мы сделали многое в этой статье, но в приложения в реальном мире, инициализация состояний в UI слое не очень хорошая идея. Отделим логику от UI.

Существует множество паттернов в Flutter, но мы будем использовать BLoC так как он наиболее гибкий для настройки.

```
class ClientsBloc {
  ClientsBloc() {
    getClients();
  }
  final _clientController =     StreamController<List<Client>>.broadcast();
  get clients => _clientController.stream;

  dispose() {
    _clientController.close();
  }

  getClients() async {
    _clientController.sink.add(await DBProvider.db.getAllClients());
  }
}
```

Заметки:
Notes:

1. getClients получает данные из БД (Client table) асинхронно. Мы будем использовать этот метод всегда, когда нам будет необходимо обновить таблицу, следовательно стоит поместить его в тело конструктора.

2. Мы создали StreamController.broadcast, для того чтобы слушать широковещательные события более одного раза. В нашем примере это не имеет особо значения, поскольку мы слушаем их только один раз, но неплохо было бы реализовать это на будущее.

3. Не забываем закрывать потоки. Таким образом мы предотвратим мемори лики. В нашем примере мы закрываем их используя dispose method в StatefulWidget

Теперь посмотрим на код

```
blockUnblock(Client client) {
  DBProvider.db.blockOrUnblock(client);
```

```
  getClients();
}

delete(int id) {
  DBProvider.db.deleteClient(id);
  getClients();
}

add(Client client) {
  DBProvider.db.newClient(client);
  getClients();
}
```

И наконец финальный результат

Исходники можно посмотреть здесь — Github

Создадим BLoC