Sqlcool Documentation

Release 0.1

synw

Feb 18, 2020

Crud operations

The objectives of this lib is to provide a simple api while staying close to sql.

CHAPTER 1

Schema definition

1.1 Columns

```
DbTable category = DbTable("category")..varchar("name", unique: true);
DbTable product = DbTable("product")
   ..varchar("name", unique: true)
   ..integer("price")
  ..real("number")
  ..boolean("bool", defaultValue: true)
  ..text("description")
  ..blob("blob")
  ..timestamp()
   ..foreignKey("category", onDelete: OnDelete.cascade);
```

Parameters for the column constructors:

name *String* the name of the column

Optional parameters:

unique *bool* if the column must be unique

nullable *bool* if the column can be null

defaultValue *dynamic* (depending on the row type: integer if the row is integer for example) the default value of a column

check *String* a check constraint: ex:

```
DbTable("table")..integer("intname", check="intname>0");
```

Note: the foreignKey must be placed after the other fields definitions

Create an index on a column:

```
DbTable("table")
    ..varchar("name")
    ..index("name");
```

Unique together constraint:

```
DbTable("table")
    ..varchar("name")
    ..integer("number")
    ..uniqueTogether("name", "number");
```

1.2 Methods

Initialize the database with a schema:

```
db.init(path: "mydb.sqlite", schema: <DbTable>[category, product]);
```

Check if the database has a schema:

```
final bool hasSchema = db.hasSchema()  // true or false;
```

Get a table schema:

```
final DbTable productSchema = db.schema.table("product");
```

Check if a table is in the schema:

```
final bool tableExists = db.schema.hasTable("product");
```

Check if a table has a column:

```
final bool columnExists = db.schema.table("product").hasColumn("name");
```

CHAPTER 2

Initialize database

2.1 Initialize an empty database

```
import 'package:sqlcool/sqlcool.dart';

Db db = Db();

// either use the schema definition constructor
// or define the tables by hand
void myInit() {
   String q1 = """CREATE TABLE category (
      id INTEGER PRIMARY KEY,
      name TEXT NOT NULL
   )""";
   String q2 = """CREATE TABLE product (
      id INTEGER PRIMARY KEY,
      name TEXT NOT NULL,
      price REAL NOT NULL,
      category_id INTEGER,
      CONSTRAINT category
         FOREIGN KEY (category_id)
         REFERENCES category(id)
         ON DELETE CASCADE
   )""";
   // the path is relative to the documents directory
   String dbpath = "data.sqlite";
   List<String> queries = [q1, q2];
   db.init(path: dbpath, queries: queries, verbose: true).catchError((e) {
       throw("Error initializing the database: $e");
   });
}

void main() {
   /// initialize the database async. Use the [onReady]
```

(continues on next page)

(continued from previous page)

```
   /// callback later to react to the initialization completed event
  myInit();
  runApp(MyApp());
}

// then later check if the database is ready

@override
void initState() {
   db.onReady.then((_) {
      setState(() {
         print("STATE: THE DATABASE IS READY");
      });
   });
super.initState();
}
```

Required parameters for `init`:

path *String* path where the database file will be stored: relative to the documents directory path

Optional parameter:

sqfliteDatabase *Database* an optional existing Sqflite database

queries *List<String>* queries to run at database creation

fromAsset *String* path to the Sqlite asset file, relative to the documents directory

absolutePath *bool* if *true* the provided path will not be relative to the

documents directory and taken as absolute :verbose: *bool* `true` or `false`

The database is created in the documents directory. The create table queries will run once on database file creation.

2.2 Initialize a database from an Sqlite asset file

```
void main() {
   String dbpath = "data.sqlite";
   db.init(path: dbpath, fromAsset: "assets/data.sqlite", verbose: true).
↪catchError((e) {
       print("Error initializing the database; $e");
   });
}
```

2.3 Multiple databases

```
import 'package:sqlcool/sqlcool.dart';

void main() {
   db1 = Db();
   db2 = Db();
   // ...
}
```

2.4 Verbosity

The `Db` methods have a `verbose` option that will print the query. To get more detailled information and queries results you can activate the Sqflite debug mode:

```
db.init(path: dbpath, queries: [q], debug: true);
```

CHAPTER 3

Database operations

3.1 Insert

```dart
import 'package:sqlcool/sqlcool.dart';

Map<String, String> row = {
    slug: "my-item",
    name: "My item",
}
await db.insert(table: "category", row: row, verbose: true);
```

Required parameters:

table *String* name of the table, required

row *Map<String, String>* data, required

Optional parameter:

verbose *bool* `true` or `false`

3.2 Select

```dart
import 'package:sqlcool/sqlcool.dart';

List<Map<String, dynamic>> rows =
   await db.select(table: "product", limit: 20, where: "name LIKE '%something%'",
      orderBy: "price ASC");
```

Required parameter:

table *String* name of the table, required

Optional parameters:

columns *String* the columns to select: default is "*"

where *String* the where sql clause

orderBy *String* the sql order by clause

groupBy *String* the sql group by clause

limit *int* the sql limit clause

offset *int* the sql offset clause

verbose *bool* `true` or `false`

3.3 Update

```
import 'package:sqlcool/sqlcool.dart';

Map<String, String> row = {
    slug: "my-item-new",
    name: "My item new",
}
int updated = await db.update(table: "category", row: row, where: "id=1", verbose:␣
↪true);
```

Required parameters:

table *String* name of the table, required

row *Map<String, String>* data, required

Optional parameters:

where *String* the where sql clause

verbose *bool* `true` or `false`

3.4 Delete

```
import 'package:sqlcool/sqlcool.dart';

await db.delete(table: "category", where: "id=1");
```

Required parameters:

table *String* name of the table, required

where *String* the where sql clause

Optional parameter:

verbose *bool* `true` or `false`

3.5 Upsert

```
import 'package:sqlcool/sqlcool.dart';

Map<String, String> row = {
    slug: "my-item",
    name: "My item",
}
await db.upsert(
              table: "product",
              row: row,
              preserveRow: "category",
              indexColumn: "id"
     );
```

Required parameters:

table *String* name of the table, required

row *Map<String, String>* data, required

Optionnal parameters:

preserveColumns *List<String>* a list of columns to preserve,

the data in these columns will not be updated. Note: the *indexColumn* parameter is required when using this method (used to retrieve the existing data). :indexColumn: *String* the reference index column use to retrieve existing data in case of preserve :verbose: *bool* `true` or `false`

3.6 Join

```
import 'package:sqlcool/sqlcool.dart';

List<Map<String, dynamic>> rows = await db.join(
                  table: "product", offset: 10, limit: 20,
                  columns: "id, name, price, category.name as category_name",
                  joinTable: "category",
                  joinOn: "product.category=category.id");
```

Required parameter:

table *String* name of the table, required

Optional parameters:

columns *String* the select sql clause

where *String* the where sql clause

joinTable *String* join table name

joinOn *String* join on sql clause

orderBy *String* the sql order by clause

groupBy *String* the sql group by clause

limit *int* the sql limit clause

offset *int* the sql offset clause

verbose *bool* `true` or `false`

3.7 Exists

```
import 'package:sqlcool/sqlcool.dart';

bool exists = await db.exists(table: "category", "id=3");
```

Required parameters:

table *String* name of the table, required

where *String* the where sql clause

3.8 Raw query

```
import 'package:sqlcool/sqlcool.dart';

List<Map<String, dynamic>> result = await db.query("SELECT * FROM mytable");
```

Required parameters:

query *String* the sql query, required

verbose *bool* `true` or `false`

CHAPTER 4

Batch insert

```dart
import 'package:sqflite/sqlflite.dart';
import 'package:sqlcool/sqlcool.dart';

var rows = <Map<String, String>>[{"name": "one"}, {"name": "two"}];

await db.batchInsert(
         table: "item",
         rows: rows,
         configAlgoritm: ConflictAlgorithm.replace)
```

CHAPTER 5

Using the bloc pattern for select

A *SelectBloc* is available to use the bloc pattern.

5.1 Select bloc

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

class _PageSelectBlocState extends State<PageSelectBloc> {
  SelectBloc bloc;

  @override
  void initState() {
    super.initState();
    this.bloc = SelectBloc(
        table: "items", orderBy: "name", verbose: true);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("My app"),
      ),
      body: StreamBuilder<List<Map>>(
          stream: bloc.items,
          builder: (BuildContext context, AsyncSnapshot snapshot) {
            if (snapshot.hasData) {
              // the select query has not found anything
              if (snapshot.data.length == 0) {
                return Center(
                  child: Text(
                      "No data. Use the + in the appbar to insert an item"),
```

(continues on next page)

(continued from previous page)

```
                );
              }
              // the select query has results
              return ListView.builder(
                  itemCount: snapshot.data.length,
                  itemBuilder: (BuildContext context, int index) {
                    var item = snapshot.data[index];
                    return ListTile(
                      title: GestureDetector(
                        child: Text(item["name"]),
                        onTap: () => print("Action"),
                      ),
                    );
                  });
            } else {
              // the select query is still running
              return CircularProgressIndicator();
            }
          }),
    );
  }
}

class PageSelectBloc extends StatefulWidget {
  @override
  _PageSelectBlocState createState() => _PageSelectBlocState();
}
```

`SelectBloc` class:

Required parameter:

table *String* name of the table, required

Optional parameters:

select *String* the select sql clause

where *String* the where sql clause

joinTable *String* join table name

joinOn *String* join on sql clause

orderBy *String* the sql order_by clause

limit *int* the sql limit clause

offset *int* the sql offset clause

reactive *bool* if *true* the select bloc will react to database changes. Defaults to *false*

verbose *bool* `true` or `false`

database *Db* the database to use: default is the default database

5.2 Join queries

```
@override
void initState() {
   super.initState();
   this.bloc = SelectBloc(table: "product", offset: 10, limit: 20,
                          select: "id, name, price, category.name as category_name",
                          joinTable: "category",
                          joinOn: "product.category=category.id");
}
```

CHAPTER 6

Declare the model

It is possible to use a mixin to extend a custom model and give it database interaction methods. This way when querying the database no deserializing and type casts are needed: only model objects are used

6.1 Extend with DbModel

```
class Car with DbModel {
    String name;
    double price;
}
```

6.2 Override getters

```
class Car with DbModel {
    @override
    int id;

    @override
    Db get db => conf.db;

    @override
    DbTable get table => carTable;
}
```

`conf.db` is the `Db` object used. `carTable` is the car table schema

6.3 Declare a schema

```
final carTable = DbTable("car")
   ..varchar("name")
   ..integer("max_speed")
   ..real("price")
   ..integer("year")
   ..boolean("is_4wd", defaultValue: false);
```

Include this schema in your database initialization call:

```
db.init(path: "db.sqlite", schema: <DbTable>[carTable]);
```

6.4 Define serializers

The `toDb` serializer and `fromDb` deserializer must be defined

```
class Car with DbModel {
  @override
  Map<String, dynamic> toDb() {
    final row = <String, dynamic>{
      "name": name,
      "max_speed": maxSpeed,
      "price": price,
      "year": year.millisecondsSinceEpoch,
      "is_4wd": is4wd,
      "manufacturer": manufacturer.id
    };
    return row;
  }

  @override
  Car fromDb(Map<String, dynamic> map) {
    final car = Car(
    id: map["id"] as int,
    name: map["name"].toString(),
    maxSpeed: map["max_speed"] as int,
    price: map["price"] as double,
    year: DateTime.fromMillisecondsSinceEpoch(map["year"] as int),
    is4wd: (map["is_4wd"].toString() == "true"),
  );
  return car;
  }
}
```

CHAPTER 7

Data mutations

Once properly declared the model can be modified in the database

7.1 Insert

```
final car = Car(name: "My car", price: 25000.0);
car.sqlInsert();
```

7.2 Update

```
car.price = 23000.0;
car.sqlUpdate();
```

7.3 Upsert

```
car.name = "My new car name";
car.sqlUpsert();
```

7.4 Delete

```
car.sqlDelete();
```

The query parameters are the same than for regular queries: check the database operations section for details

CHAPTER 8

Select operations

The select calls are done via an instance of the model. The recommended method is to define some select static methods in your model:

```
class Car with DbModel {
    static Future<List<Car>> select({String where, int limit}) async {
        final cars = List<Car>.from(
            await Car().sqlSelect(where: where, limit: limit));
        return cars;
    }
}
```

And then use it:

```
List<Car> cars = await Car.select(where: "price<50000");
```

CHAPTER 9

Foreign keys support

The database models support foreign keys. Example: create a foreign key model:

```
class Manufacturer with DbModel {
    Manufacturer({this.name});

    final String name;

    @override
    int id;

    @override
    Db get db => conf.db;

    @override
    DbTable get table => manufacturerTable;

    @override
    Map<String, dynamic> toDb() => <String, dynamic>{"name": name};

    @override
    Manufacturer fromDb(Map<String, dynamic> map) =>
        Manufacturer(name: map["name"].toString());
}
```

To set a foreign key mention it in your table schema:

```
final carTable = DbTable("car")
   ..varchar("name")
   ..real("price")
   ..foreign_key("manufacturer");
```

Update the serializers in the main model to use the foreign key:

```
class Car with DbModel {
    @override
    Map<String, dynamic> toDb() {
        final row = <String, dynamic>{
            // ...
            "manufacturer": manufacturer.id
        };
    return row;
    }

    @override
    Car fromDb(Map<String, dynamic> map) {
        final car = Car(
            // ...
        );
        // the key will be present only with join queries
        // in a simple select this data is not present
        if (map.containsKey("manufacturer")) {
            car.manufacturer =
                Manufacturer().fromDb(map["manufacturer"] as Map<String, dynamic>);
        }
        return car;
    }
}
```

To perform a join query:

```
class Car with DbModel {
    static Future<List<Car>> selectRelated({String where, int limit}) async {
        final cars = List<Car>.from(
            await Car().sqlJoin(where: where, limit: limit));
        return cars;
    }
}
```

And then use it:

```
List<Car> cars = await Car.selectRelated(where: "price<50000");
print(cars[0].manufacturer.name);
```

CHAPTER 10

Reactivity

10.1 Changefeed

A changefeed is available (inspired by Rethinkdb). It's a stream that will notify about any change in the database.

```
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:sqlcool/sqlcool.dart';
import 'dialogs.dart';

class _PageState extends State<Page> {
  StreamSubscription _changefeed;

  @override
  void initState() {
     _changefeed = db.changefeed.listen((change) {
      print("CHANGE IN THE DATABASE:");
      print("Change type: ${change.type}");
      print("Number of items impacted: ${change.value}");
      print("Query: ${change.query}");
      if (change.type == DatabaseChange.update) {
        print("${change.value} items updated");
      }
    });
    super.initState();
  }

  @override
  void dispose() {
    _changefeed.cancel();
    super.dispose();
  }

  // ...
```

(continues on next page)

(continued from previous page)

```
}

class Page extends StatefulWidget {
  @override
  _PageState createState() => _PageState();
}
```

10.2 Reactive select bloc

A `SelectBloc` can take a `reactive` parameter. If it is `true` the bloc will automatically rebuild itself on any database change

Check the example for usage demo.

CHAPTER 11

Synchronized map

A map that will auto save it's values to the database.

```
import 'package:sqlcool/sqlcool.dart';

// Define the map with initial data
var myMap = SynchronizedMap(
   db: db, // an Sqlcool database
   table: "a_table",
   where: "id=1",
   columns = "col1,col2,col3",
   verbose: true
);

// Wait until the map is initialized
await myMap.onReady;

// Changing the map will auto update the data in the database:
myMap.data["col1"] = "value";

// Dispose the map when finished using
myMap.dispose();
```

CHAPTER 12

Indices and tables

- genindex
- modindex
- search