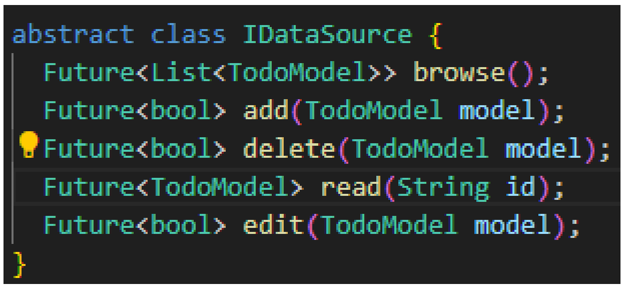
SQLite Tutorial – Todo's

Introduction

This tutorial will take you through importing the packages and setting up for SQLite in flutter. Note that SQLite is not available on the web deployment platform. It will add the mapping to SQL from the model so we can easily convert and create the objects in flutter and SQL. It will also cover the option of extending and dynamically selecting implementations using a generic class hierarchy and the getIt package.

Datasource Interface

Create a **new folder** for services and inside create an **interface** (abstract class in dart) for your todo datasource. Make sure the functions do not declare a body (no curly brackets). Create the basic CRUD/BREAD style functions for retrieving the data without writing any code to implement it yet. Each data source (offline SQL or API will use this as a basis). We are going to be interacting with a database which will require **asynchronous functionality** to execute. The return types should be a **Future<type>**. This is what Dart uses for asynchronous functions like Task<type> in C#. An example of what this might look like is this:

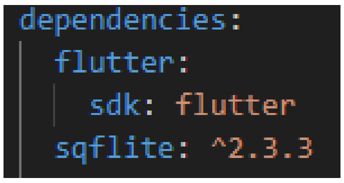


```
abstract class IDataSource {
  Future<List<TodoModel>> browse();
  Future<bool> add(TodoModel model);
  Future<bool> delete(TodoModel model);
  Future<TodoModel> read(String id);
  Future<bool> edit(TodoModel model);
}
```

Note: for this tutorial I will follow the BREAD convention for data access.

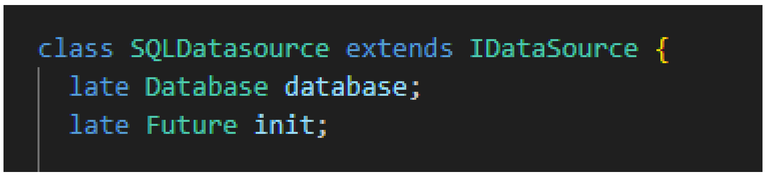
SQLite Datasource

Now that we have the generic interface that defines all our executions, we can create an SQLite version that uses this signature to query a local database. This will be useful for offline data when an internet connection (and our API) is not available. To add this functionality, **we need** two things **a new class** that implements our interface and **the package for SQLite**. In the **pubspec.yaml** where we add packages add the **sqflite: ^2.3.3** package.



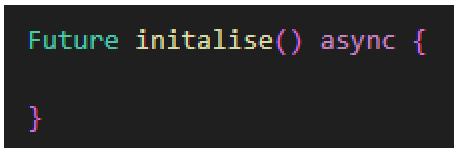
```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.3.3
```

Create the **new file** for the SQLiteDatasource that **implements the datasource interface** you created before. Give it two fields a "**late Database database;**". This will be the registered database that is either created or opened by SQLite. Then we will create a future for the initialisation of the class and SQLite, as this is created asynchronously. Finally, **a constructor** that calls the **async function** to set up the database correctly.



```
class SQLDatasource extends IDataSource {
  late Database database;
  late Future init;
```

Now we can create the initialise function. This is an async function that does not return anything. Like shown before we can use the Future type to return from or manage async functionality. Like C# the function also declares that it is async, however it does so slightly differently in syntax. The Async keyword comes after the function parentheses but before the body curly brackets.



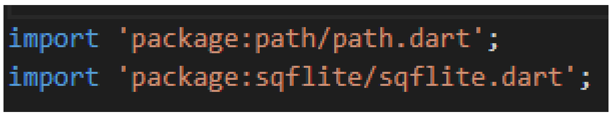
```
Future initalise() async {

}
```

Initialise()

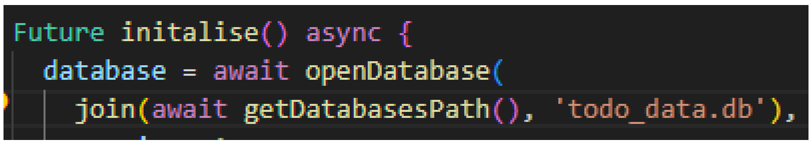
Now we have the function we can set up the database and complete by setting the initialised. Assign the database to be equal to the result of the openDatabase();

The openDatabase() function will take two parameters, one is necessary, the path to the file that holds the SQLite database. To create this, we also need to import another core package. At the top of the page import these files:



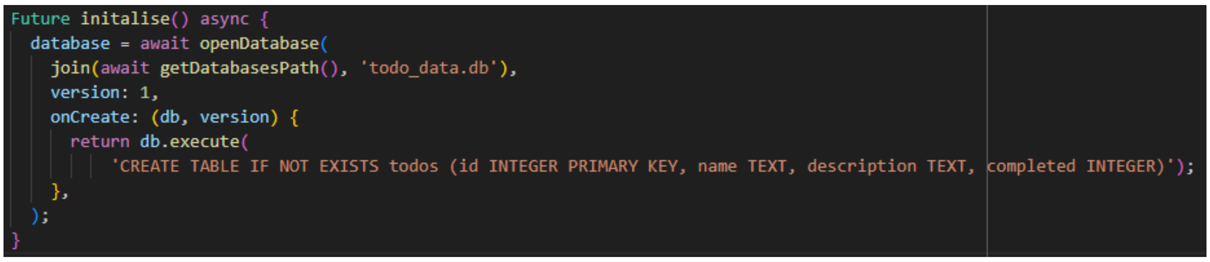
```
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
```

The path dart file helps us concat the two strings that represent the path while maintaining platform specific syntax. Use join() to pass in two parts: await getDatabasePath() and the string that represents the database itself. For this example, use 'todo_database.db'.



```
Future initalise() async {
  database = await openDatabase(
    join(await getDatabasesPath(), 'todo_data.db'),
```

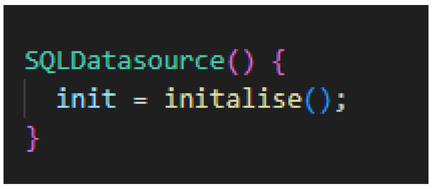
Finally, we need the onCreate: function that defines what happens when the database is created. This takes two parameters the DB and the version. Inside this we need to execute what happens when the creation of the database is finished. I.E set up any tables that are necessary for the app to work. This will not return anything so we can call the db.execute and create the tables necessary. In this case we only really need one now. Finally, set initialised to true to let the rest of the object know its finished the async set up processes. Follow SQLite syntax for creating your database (check your types). Your completed function should look like this:



```
Future initalise() async {
  database = await openDatabase(
    join(await getDatabasesPath(), 'todo_data.db'),
    version: 1,
    onCreate: (db, version) {
      return db.execute(
        'CREATE TABLE IF NOT EXISTS todos (id INTEGER PRIMARY KEY, name TEXT, description TEXT, completed INTEGER)');
    },
  );
}
```

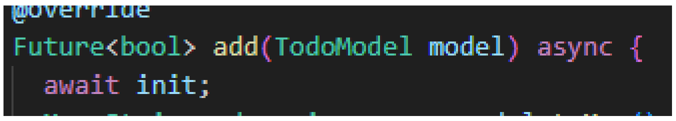
Now that we have created the initialise function that creates and sets up the database we can call this within our constructor to ensure the Future is captured and can be called upon later to wait before any other execution:

<u>Constructor</u>



```
SQLDatasource() {
  init = initalise();
}
```

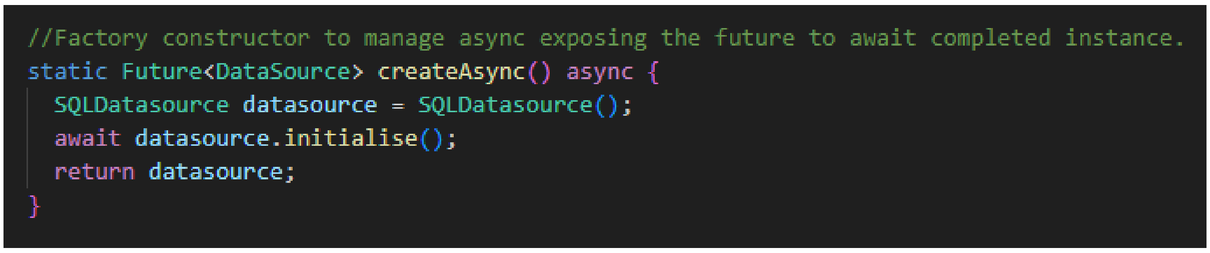
<u>Example Function</u>



```
@override
Future<bool> add(TodoModel model) async {
  await init;
```

Factory Await Constructor (Refactor)

An alternative and improved approach would be to use the factory method to generate the class. Not to be confused with the factory software development pattern but inspired by the idea:

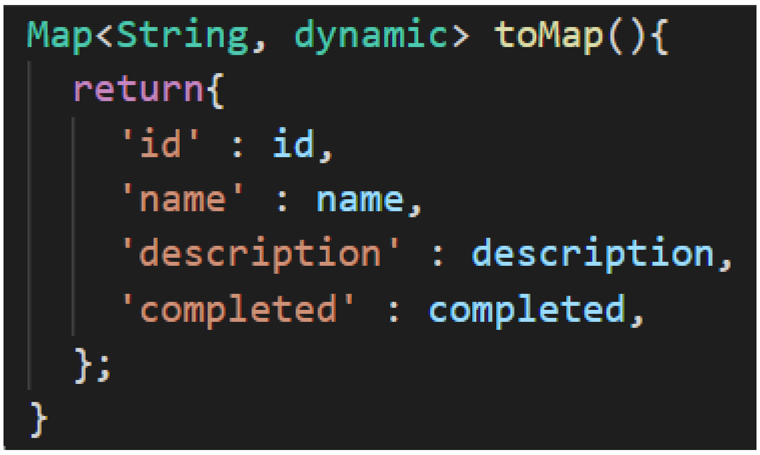


```
//Factory constructor to manage async exposing the future to await completed instance.
static Future<DataSource> createAsync() async {
  SQLDatasource datasource = SQLDatasource();
  await datasource.initialise();
  return datasource;
}
```

What this achieves instead, is enables the await initialise method to resolve before returning the instance. This method itself can be awaited to ensure that the class constructor does not complete until the initialise method has finished. While we still call the initialise, instead this time we are exposing the future of that to the caller. Effectively now we do not need the use of await init in every method call in this class. Cleaning up that approach.

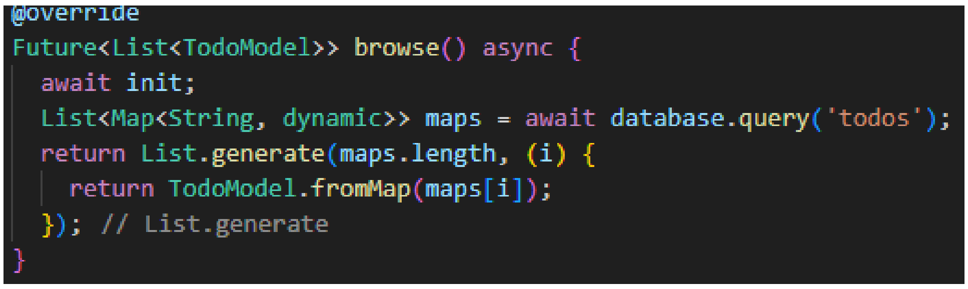
ToMap()

Updating the **model** to work smoothly with SQLite, add the toMap() functionality. This allows the code to map back to the SQL key value pair it needs. When we add a model, we can use this to write straight to SQL. Create a function toMap() that returns a Map<string, dynamic>. Return an object in curly brackets with named parameters like this:



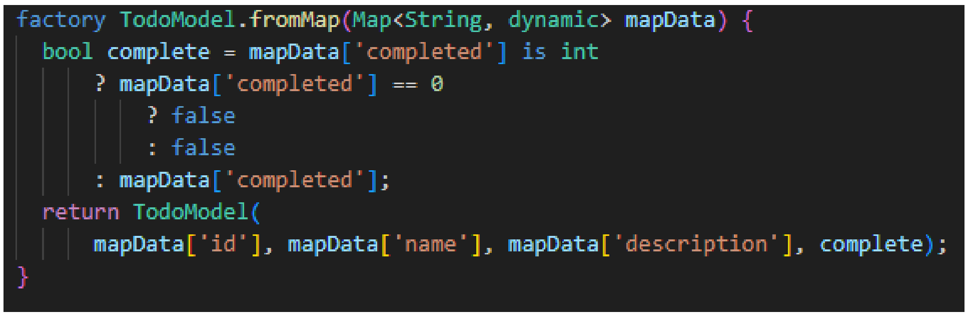
```
Map<String, dynamic> toMap(){
  return{
    'id' : id,
    'name' : name,
    'description' : description,
    'completed' : completed,
  };
}
```

Implement the functions for each BREAD aspect of the SQL database. To start here is an example of the browse() function:



```
@override
Future<List<TodoModel>> browse() async {
  await init;
  List<Map<String, dynamic>> maps = await database.query('todos');
  return List.generate(maps.length, (i) {
    return TodoModel.fromMap(maps[i]);
  }); // List.generate
}
```

A few things to note here. The use of List.generate from a length and item that returns an object. In this example I have collapsed the accessing of the map into a factory function inside my todo model. It returns an instance of the todo from the map data. This uses the map by accessing the maps properties in square brackets.



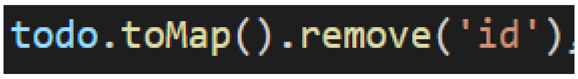
```
factory TodoModel.fromMap(Map<String, dynamic> mapData) {
  bool complete = mapData['completed'] is int
      ? mapData['completed'] == 0
          ? false
          : false
      : mapData['completed'];
  return TodoModel(
      mapData['id'], mapData['name'], mapData['description'], complete);
}
```

Note that when we fetch the Todo from the database, we fetch the Id but when we store a new Todo, we don't need to provide the ID. This is due to SQLite's natural increment for the primary key.

Exercise 1 – BREAD

- Complete the rest of the Todo SQL functions for:
  - Add()

Note: For add, you need to remove the id for inserting so the id is created automatically. As our map includes the id you can remove the key like this:



```
todo.toMap().remove('id')
```

Note: Remove function returns the value it removed

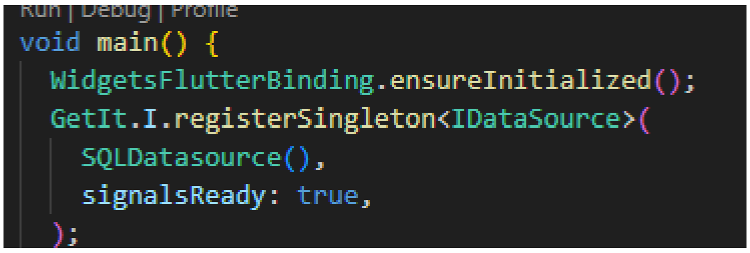
  - Delete()
  - Edit()

Get it

Next, we're going to set our generics up. This allows us to register an implementation of a source of our data using an interface that will potentially update based on location, Os or internet connectivity. Our datasource implements our tododatasource which is our generic class. We can register a type of this in the get it package. The get it package is a service locator that helps register and find services from anywhere in our code. There are ways we could use this to overcome some obstacles Provider/consumer faces however, it also can be used alongside to complement it.

Setup

Add the get_it: ^7.6.0 package to the pubspec.yaml file. Then, inside the main function in our main.dart add the following lines:

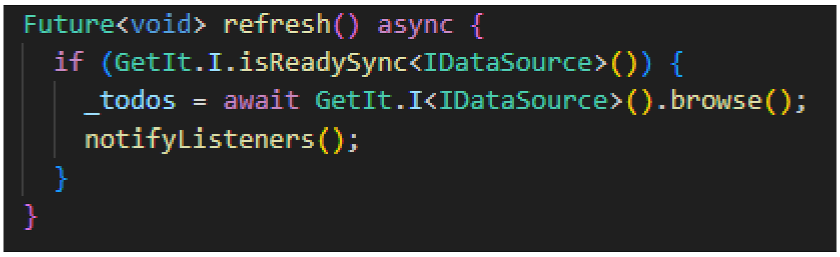


```
Run | Debug | Profile
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  GetIt.I.registerSingleton<IDataSource>(
    SQLDatasource(),
    signalsReady: true,
  );
```

This waits for flutter to finishing binding the widgets then loads our services. Now we can add our registration of the singleton of our datasource. Signals ready means we can use a function later to check if the class is initialised. In a synchronous declaration this isn't necessary but good to validate in case the registration takes a while.

Todo_list.dart

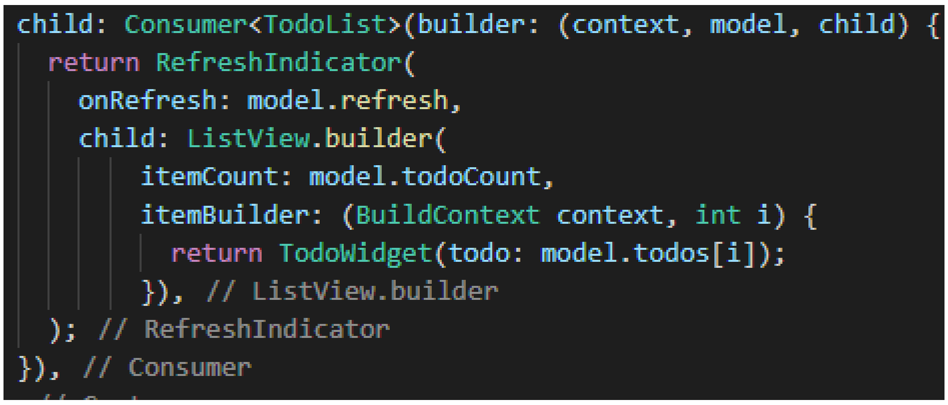
Now we must update our model to pull from the database. In the todo_list.dart model that we used in our provider/consumer. Add a refresh function. It needs to be async and update the todos by awaiting the result of the browse() call. This returns our list regardless of the source. It must be async so returns a future with the list, if awaited it just returns the resulting list. It looks something like this:



```
Future<void> refresh() async {
  if (GetIt.I.isReadySync<IDataSource>()) {
    _todos = await GetIt.I<IDataSource>().browse();
    notifyListeners();
  }
}
```

To allow us to update the _todos from inside the model we need to remove the final as will not be fetched from the database. To reflect the data from the database when the Todo list model loads, add a constructor that calls the refresh function. (Calling this should update the ui).

Now we can update out view (main.dart) to be able to refresh the ui by adding a Refresh indicator widget. It takes two parameters: onRefresh and child. Wrap our listview.builder with this widget (don't forget to use Ctrl + . To automatically wrap a widget. You can replace the type later if you need and add extra parameters. Your final consumer Todo list should look like this:



```
child: Consumer<TodoList>(builder: (context, model, child) {
  return RefreshIndicator(
    onRefresh: model.refresh,
    child: ListView.builder(
        itemCount: model.todoCount,
        itemBuilder: (BuildContext context, int i) {
          return TodoWidget(todo: model.todos[i]);
        }), // ListView.builder
  ); // RefreshIndicator
}), // Consumer
```

Todo ID

The Todo id is final and reflected from the database, when we add a Todo, we do not map the Todo id because it is used to create the ID in the database, and by default that will always be 0. which means every time we add a new Todo it will overwrite the first entry in the table. This is why we remove the id in the add method after we toMap it. Its kept however as other DataSource's may utilise the ID.

Exercise 2 – Polish and complete

Complete the remaining functions and add the add function into your todo_list by calling the getIt instance to get the DataSource and calling add.

Note: There are some additional tweaks to be completed to make sure that the application runs, such as updating the Todo to use and take in the ID that the DB needs (this can also be resolved another way). Do research to resolve any issues found and work through in class to complete the rest of the components using what knowledge has been touched on in this tutorial.