**Local Data Storage**

Local data storage in context of mobile applications refer to storing data inside users’ device. Since the data exist on their device, it provides better accessibility where no network operation is required to access or send data to local storage compared to server-side database.

However, it is important to note that when comparing local data storage vs centralized server-side data storage, none of these two methods is above each other’s in all scenarios. Both has their own specific uses which developers themselves need to decide on which to use.

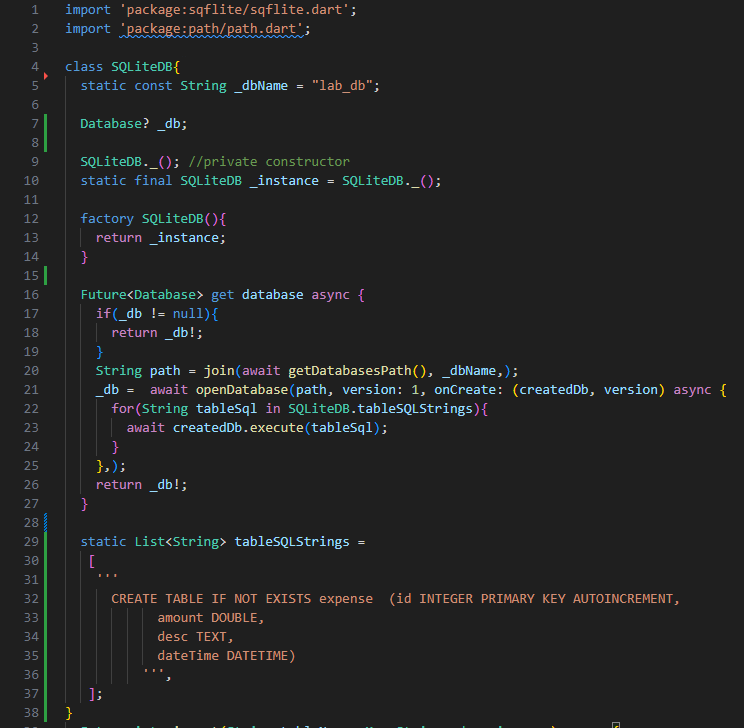
Although, as mentioned previously, local data storage is far better in terms of accessibility of data, it is not shared with other users since it is decentralized data storage. Each of users’ device has their own local data storage which stores different data or different copies of data. Hence, it is not suitable to be used for data which is shared for all user, for example products in shopping apps or posts in social media apps. However, sometimes it is still used to store these kinds of centralized data as local backup to enable applications to work offline to certain extent.

Few of common ways to store data locally on user device that has their own different uses which is:

* **Shared Preferences**
  + A simple key value pair data storage which is specific to an application. It is useful for simple data types such as string, int, double etc.
  + Shared preferences are unable to handle complex data types such as map by default. Even though it is key value pair like almost similar to Json, but it doesn’t have built in support to handle nested key value pairs without using Json strings.
  + Also lacks searching features which actually is not needed in its use due to its simple structure.
  + Mostly, used to store small and unstructured data such as users’ authentication token or application configurations.
* **SQLite**
  + A structured database storage like MySQL, MariaDB etc.
  + It is specific to each application where each application has their own SQLite directory which is isolated from other apps. Even if multiple applications use same database name it won’t mix up.
  + Provide relational database feature similar to most RDBMS.
  + Allow querying data for easier access and even complex queries involving joins.
  + It can be used for most cases but sometime might be an overkill to use SQLite just to store simple data like token.

SQLite implementation in flutter

1. Install dependencies. sqflite: ^2.3.0
2. Create and define a new controller “SQLiteDB”



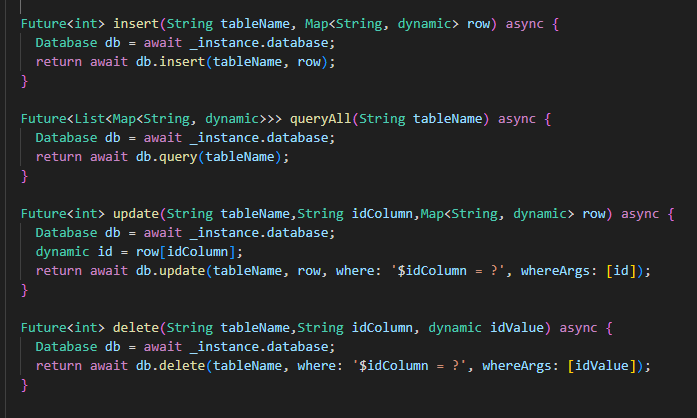
The structure of this class might appear weird since it is different than normal class. It is a **singleton** **class** which is a pattern in defining a class to ensure that **only 1 instance** of the class is instantiated and used throughout the application.

* Firstly, since ideally, we only need 1 database for an application, the name is declared as static attribute.
* Then we have the nullable private instance attribute \_db.
* P**rivate constructor** is declared to limit instantiation so that this class can’t be instantiated normally like normal classes.
* S**tatic object** of this class is declared using the private constructor which will be the only instance of this class in the application.
* **Factory constructor** is used to return the static instance of this class. Factory constructor is one of the alternatives ways to instantiate a class. It can be used to implement **additional logic in your class instantiation** which might be outside of the class scope. Analogically, let say the class is cake recipe. The baking process is instantiation which produce a cake (object). But the cake is not delivered to buyer directly, it goes through packaging process. Can the cake package itself? Of course, not since it is outside the scope of the object itself. This packaging is example of additional logic which can be added inside a factory method. Factory constructor has many other uses depends on what logic is added in the instantiation.

In this context, factory constructor is used to return the existing object of the database instead of instantiating new one to ensure one single database instance is maintained during runtime.

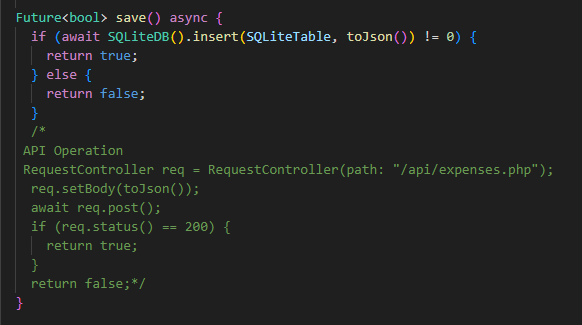
* A getter method is used to handle the access to the \_db attribute which is the actual SQLite database object. Additional logic is included in the getter method where we initialize the database if it is still null by opening the database in the specified path and database name. in the onCreate callback of the openDatabase() method, we execute the DDL queries of the related tables which is stored in a static attribute tableSQLStrings by looping through the list.
* Adding more table can simply be achieved by adding more entries to the tableSQLStrings list.

1. Define simple CRUD methods in the SQLiteDB class.



* In each of these methods, the actual SQLite database instance is accessed via the getter method/properties “database” which will automatically open the database first if it is not yet open.
* Note that, these are very simple CRUD method, modification will definitely be needed for more complex database transactions.

1. Change the implementation of save and loadAll method in “Expense” class to use SQLite instead of API calls. Notice that class name with bracket SQLiteDB() is called to use the SQLite feature. While this statement does return an instance of SQLiteDB, it does not create a new instance. This is an example of how factory constructor can be called which is same as normal constructor. Only the underlying logic is different.



A computer screen shot of code

Description automatically generated

1. Run and test the application.

Notice that, even though changes occur in the application, the view/interface part can remain untouched. This is the benefit of decoupling codes to decrease the dependency chain.

It is also the reason why classes such as SQLiteDB and RequestController is necessary since both SQflite and http, the package used are not internally part of our application. Using such dependencies on multiple files impose risk of tedious change management.

Technically, we can directly use http and SQflite inside our model class but imagine 20 or hundreds model class which each directly uses the dependency. Then, suddenly the dependencies are deprecated, or you have to change it to others or maybe required to add additional logic. Without proper wrapper/controller class you will have to go into each class that use that dependency one by one to implement changes.