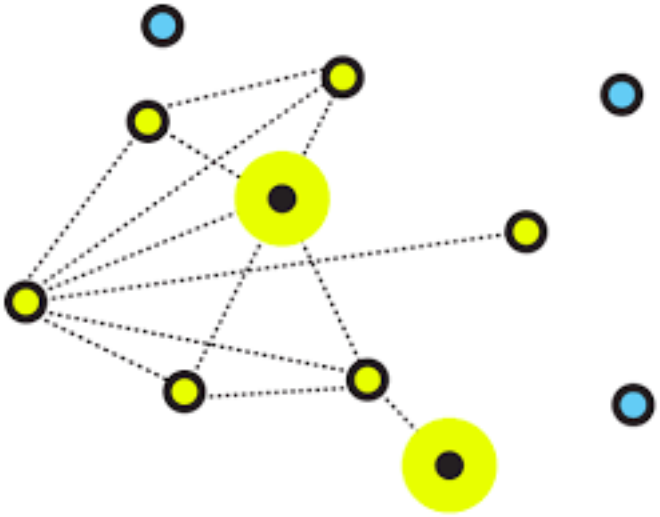
Chapter VI

PERSISTENCE IN FLUTTER



1.File Operation in Flutter

In some cases, you need to read and write files to disk. For example, you may need to persist data across app launches, or download data from the internet and save it for later offline use.

We will user path_provider plugin to get read and writable file path along with dart:io library

1. Read and write files

The path_provider plugin provides a platform-agnostic way to access commonly used locations on the device's file system. The plugin currently supports access to two file system locations:

Temporary directory

A temporary directory (cache) that the system can clear at any time. On iOS, this corresponds to the NSCachesDirectory. On Android, this is the value that getCacheDir()) returns.

Documents directory

A directory for the app to store files that only it can access. The system clears the directory only when the app is deleted. On iOS, this corresponds to the NSDocumentDirectory. On Android, this is the AppData directory.

Eg.

```
Future<String> get _localPath async {
  final directory = await getApplicationDocumentsDirectory();

  return directory.path;
}
```

2. Create a reference to the file location

```
Future<File> get _localFile async {

          final path = await _localPath;
          return File('$path/counter.txt');
    }
```

3. Write data to the file

```
Future<File> writeCounter(int counter) async {
  final file = await _localFile;

  // Write the file.
  return file.writeAsString('$counter');
}
```

4. Read data from the file

```
Future<int> readCounter() async {
  try {
    final file = await _localFile;

    // Read the file.
    String contents = await file.readAsString();

    return int.parse(contents);
  } catch (e) {
    // If encountering an error, return 0.
    return 0;
  }
}
```

Eg. https://flutter.dev/docs/cookbook/persistence/reading-writing-files

2. Database Operation with SQLite in Flutter

If writing an app that needs to persist and query larger amounts of data on the local device, consider using a database instead of a local file or key-value store. In general, databases provide faster inserts, updates, and queries, compared to other local persistence solutions.
Flutter apps can make use of the SQLite databases via the sqflite plugin available on pub. This recipe demonstrates the basics of using sqflite to insert, read, update, and remove data about various Dogs.

For Understanding of SQLite refer to http://www.sqlitetutorial.net/

1. **SQFlite**

1.Add In pubspec.yaml
dependencies:

```
  flutter:
    sdk: flutter
  sqflite:
  path:
```

2.Import in required file

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

■ **Available Methods**

-openDatabase (

```
String path, {int version,
FutureOr<void> Function(Database) onConfigure,
FutureOr<void> Function(Database, int) onCreate,
FutureOr<void> Function(Database, int, int) onUpgrade,
FutureOr<void> Function(Database, int, int) onDowngrade,
FutureOr<void> Function(Database) onOpen,
bool readOnly = false,
bool singleInstance = true}.  )
```

```
eg.final Future<Database> database = openDatabase(

  join(await getDatabasesPath(), 'mysqlite.db'),

  onCreate: (db, version) {
    var sql = "CREATE TABLE users(id INTEGER PRIMARY KEY,
         username TEXT, password TEXT)"
    return db.execute( sql )
     ,
    );
  },

  version: 1,
);
```

Database Class Methods

db.execute(sql);
db.insert(
String table,
Map<String, dynamic> values,
{ String nullColumnHack,
ConflictAlgorithm conflictAlgorithm.
}

Future<List<Map<String, dynamic>>> **db.query**(
String table,
{bool distinct,
List<String> columns,
String where,
List<dynamic> whereArgs,
String groupBy,
String having,
String orderBy,
int limit,
int offset})
db.batch( );
db.close( );
db.delete(tableName,{String where, List<dynamic> whereArgs}));

```
eg.final Database db = await database;

  var newUser = User(1,”admin”,”admin”);
  var user1 = User(2,”user”,”user”)
  var user2 = User(3,”user”,”user”)

   db.insert(
     'users',
     newUser.toMap(),
     conflictAlgorithm: ConflictAlgorithm.replace,
   );
```

```
eg.final Database db = await database;

 final List<Map<String, dynamic>> usermaps = await db.query('users');
 print(usermaps)

var users = List.generate(maps.length, (i) {
     return Dog(
       id: maps[i]['id'],
       name: maps[i]['user'],
       age: maps[i]['password'],
     );
   });
```

```
eg.final Database db = await database;

 final List<Map<String, dynamic>> usermaps = await db.query(‘users’,
    where: “ id = ? ”,
    whereArgs: [ 1  ]
);
 print(usermaps)
```

```
eg.final Database db = await database;

   await db.update('users',
    user2.toMap( ),
    where: " id = ? ",
    whereArgs: [ 2  ]
);
```

```
eg.final Database db = await database;

   await db.delete('users',
    user5.toMap( ),
    where: " id = ? ",
    whereArgs: [ 5  ]
);
```

■ Copying Existing Database file in flutter

```
// Construct the path to the app's writable database file:
var dbDir = await getDatabasesPath();
var dbPath = join(dbDir, "app.db");

// Delete any existing database if necessary:
//await deleteDatabase(dbPath);

// Create the writable database file from the bundled demo
database file:
ByteData data = await rootBundle.load("assets/demo.db");
List<int> bytes =
data.buffer.asUint8List(data.offsetInBytes,
data.lengthInBytes);
await File(dbPath).writeAsBytes(bytes);
```

Finally, you can open the created database file on app startup:

```
var db = await openDatabase(dbPath);
```