**Module 8: Local Storage and Persistence**

**Q. 1) Explain the difference between local storage options (shared\_preferences, SQLite, Hive).**

**A. 1. shared\_preferences**

* **What it is:** A simple key-value storage backed by native platform storage (NSUserDefaults on iOS, SharedPreferences on Android).
* **Best For:** Storing small amounts of simple data like user settings, flags, or preferences.
* **Data type support:** int, double, bool, String, List<String>.
* **Pros:**
  + Easy to use.
  + Great for lightweight, persistent data.
* **Cons:**
  + Not suitable for large or complex data.
  + No advanced querying (you only get/set values by key).
* **Example use cases:**
  + Remembering dark mode setting.
  + Saving login state (isLoggedIn = true).

**2. SQLite**

* **What it is:** A full relational database engine stored on the device.
* **Best For:** Structured, relational data with relationships (tables, rows, queries).
* **Package:** sqflite.
* **Pros:**
  + Powerful querying (SQL).
  + Well-suited for apps with complex data models.
  + Widely used and mature.
* **Cons:**
  + More boilerplate (you write SQL queries).
  + Slower to set up compared to Hive.
* **Example use cases:**
  + Offline-first apps (chat history, notes).
  + Expense tracking (tables with categories, amounts, users).

**3. Hive**

* **What it is:** A fast, lightweight, NoSQL key-value database written in Dart.
* **Best For:** Storing large amounts of data in a non-relational format.
* **Pros:**
  + Extremely fast (optimized for Flutter).
  + Strongly typed (with adapters).
  + No native dependencies → works on all platforms.
  + Easy to use compared to SQLite.
* **Cons:**
  + Not relational (no joins).
  + Not as mature for very complex queries.
* **Example use cases:**
  + Caching API responses.
  + Storing user profiles, settings, or offline data.
  + Apps needing high-speed local storage (e.g., games).

**Comparison Table**

| **Feature** | **shared\_preferences** | **SQLite (sqflite)** | **Hive** |
| --- | --- | --- | --- |
| **Type** | Key-Value Store | Relational DB | NoSQL (Key-Value / Boxes) |
| **Data Size** | Small (prefs, flags) | Large, structured data | Large, unstructured or semi-structured |
| **Complexity** | Very easy | Medium (requires SQL) | Easy (custom objects via adapters) |
| **Performance** | Fast for small data | Slower than Hive for large ops | Very fast (optimized for Flutter) |
| **Best Use Case** | App settings | Offline apps with tables/relations | Caching, fast storage, gaming apps |

**Real-Life Examples**

* **shared\_preferences →** Save "theme = dark", "token = abc123".
* **SQLite → Notes app:** notes table with id, title, content, date.
* **Hive →** Store offline music playlists, cached products, or game scores.

**In summary:**

* Use shared\_preferences for simple key-value settings.
* Use SQLite when you need structured relational data with queries.
* Use Hive for fast, large, NoSQL-style storage (especially if you don’t want SQL overhead).

**Q. 2) Describe CRUD operations and how they are implemented in SQLite or Hive.**

**A. What is CRUD?**

**CRUD stands for the four basic operations you can perform on data:**

1. **C – Create →** Add new data.
2. **R – Read →** Retrieve data.
3. **U – Update →** Modify existing data.
4. **D – Delete →** Remove data.

**🗄 CRUD in SQLite (with sqflite package)**

**SQLite is a relational database. You define tables, then use SQL queries.**

**Example: User Model**

class User {

final int? id;

final String name;

final int age;

User({this.id, required this.name, required this.age});

Map<String, dynamic> toMap() {

return {'id': id, 'name': name, 'age': age};

}

}

Database Helper

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

class DatabaseHelper {

static Database? \_db;

Future<Database> get database async {

if (\_db != null) return \_db!;

\_db = await \_initDB();

return \_db!;

}

Future<Database> \_initDB() async {

final path = join(await getDatabasesPath(), 'users.db');

return await openDatabase(

path,

version: 1,

onCreate: (db, version) {

return db.execute(

'CREATE TABLE users(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, age INTEGER)',

);

},

);

}

**// C – Create**

Future<int> insertUser(User user) async {

final db = await database;

return await db.insert('users', user.toMap());

}

**// R – Read**

Future<List<User>> getUsers() async {

final db = await database;

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

return List.generate(maps.length, (i) {

return User(id: maps[i]['id'], name: maps[i]['name'], age: maps[i]['age']);

});

}

**// U – Update**

Future<int> updateUser(User user) async {

final db = await database;

return await db.update(

'users',

user.toMap(),

where: 'id = ?',

whereArgs: [user.id],

);

}

**// D – Delete**

Future<int> deleteUser(int id) async {

final db = await database;

return await db.delete('users', where: 'id = ?', whereArgs: [id]);

}

}

**⚡ CRUD in Hive**

Hive is a NoSQL key-value database (much simpler, faster, no SQL required).

**Step 1: Create User Model + Hive Adapter**

import 'package:hive/hive.dart';

part 'user.g.dart';

@HiveType(typeId: 0)

class User extends HiveObject {

@HiveField(0)

String name;

@HiveField(1)

int age;

User({required this.name, required this.age});

}

**Run build command to generate adapter:**

flutter packages pub run build\_runner build

**Step 2: Open Hive Box**

void main() async {

WidgetsFlutterBinding.ensureInitialized();

Hive.initFlutter();

Hive.registerAdapter(UserAdapter());

await Hive.openBox<User>('users');

runApp(MyApp());

}

**Step 3: CRUD Operations**

var box = Hive.box<User>('users');

**// C – Create**

await box.add(User(name: "Alice", age: 25));

**// R – Read**

User? user = box.getAt(0);

print(user?.name); // Alice

**// U – Update**

user?.name = "Alice Smith";

user?.save();

**// D – Delete**

await box.deleteAt(0);

**📊 Comparison:** SQLite vs Hive for CRUD

| **Feature** | **SQLite (sqflite)** | **Hive** |
| --- | --- | --- |
| **Data Model** | Relational (tables, rows) | NoSQL (key-value, boxes) |
| **Data Definition** | SQL schema required | Schema-less (adapters for objects) |
| **Best For** | Complex structured data | Fast storage, caching, simple objects |
| **Performance** | Slower for large data | Very fast (pure Dart) |
| **Learning Curve** | Higher (SQL required) | Easier (native Dart objects) |

**In summary:**

* **CRUD =** Create, Read, Update, Delete.
* **In SQLite, you use SQL queries and tables →** good for structured, relational data.
* **In Hive, you work with boxes and Dart objects →** great for speed and simplicity.

**Q. 3) Explain the advantages and use cases for shared\_preferences.**

**A. What is shared\_preferences?**

* **A Flutter plugin that provides persistent storage for simple data as key–value pairs.**
* **It uses:**
  + **Android →** SharedPreferences (native)
  + **iOS →** NSUserDefaults (native)
* **Works like a tiny local dictionary that survives app restarts.**

**✅ Advantages of shared\_preferences**

1. **Simplicity & Ease of Use**
   * Very easy to implement (get/set values).
   * No need for database setup.
2. **Persistence**
   * Data remains even after the app is closed or restarted.
3. **Cross-Platform Support**
   * Works seamlessly on Android, iOS, Web, macOS, Linux, Windows.
4. **Lightweight**
   * Perfect for storing small data (booleans, strings, numbers, lists).
5. **No Extra Dependencies**
   * Uses platform-native storage → reliable and efficient.
6. **Performance**
   * Fast for reading/writing small amounts of data (like settings or tokens).

**Use Cases for shared\_preferences**

1. **App Settings & Preferences**
   * Dark mode toggle (theme = dark).
   * Language/locale choice (lang = en).
   * Notification settings.
2. **Authentication State**
   * Save login flag (isLoggedIn = true).
   * Store small tokens or user IDs.
3. **Onboarding / Tutorial Flow**
   * Show onboarding screens only once (hasSeenIntro = true).
4. **User Personalization**
   * Save preferred font size or color scheme.
   * Store last visited tab or page.
5. **Tiny Caching**
   * Save last search query.
   * Store last-used filter or sort order.

**Example Code**

import 'package:flutter/material.dart';

import 'package:shared\_preferences/shared\_preferences.dart';

class PreferencesExample extends StatefulWidget {

@override

\_PreferencesExampleState createState() => \_PreferencesExampleState();

}

class \_PreferencesExampleState extends State<PreferencesExample> {

bool isDarkMode = false;

@override

void initState() {

super.initState();

\_loadTheme();

}

**// Read value**

\_loadTheme() async {

final prefs = await SharedPreferences.getInstance();

setState(() {

isDarkMode = prefs.getBool('isDarkMode') ?? false;

});

}

**// Save value**

\_toggleTheme(bool value) async {

final prefs = await SharedPreferences.getInstance();

prefs.setBool('isDarkMode', value);

setState(() {

isDarkMode = value;

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text("Shared Preferences Example")),

body: Center(

child: SwitchListTile(

title: Text("Dark Mode"),

value: isDarkMode,

onChanged: \_toggleTheme,

),

),

);

}

}

**Here, the app remembers the user’s theme choice even after restart.**

**When *not* to use shared\_preferences**

* Large or complex data (use SQLite or Hive instead).
* Storing sensitive data like passwords (use flutter\_secure\_storage).

**In summary:**shared\_preferences is ideal for small, simple, persistent key-value data like settings, flags, and preferences. It’s lightweight, cross-platform, and super easy to use.