**1. Difference Between Local Storage Options (shared\_preferences, SQLite, Hive)**

Flutter provides multiple local storage solutions, each suited for different use cases. The most commonly used are **shared\_preferences, SQLite, and Hive**.

| **Storage Option** | **Type** | **Best For** | **Pros** | **Cons** |
| --- | --- | --- | --- | --- |
| **shared\_preferences** | Key-Value Storage | Storing small data (settings, user preferences) | ✅ Easy to use, ✅ Fast, ✅ Persistent storage | ❌ Not for complex data, ❌ No relational data |
| **SQLite** | Relational Database | Storing structured data with relationships | ✅ Supports SQL queries, ✅ ACID-compliant, ✅ Good for large datasets | ❌ Slower than Hive for simple data, ❌ Requires setup |
| **Hive** | NoSQL Database | High-performance local storage | ✅ Very fast, ✅ Stores complex objects, ✅ Works offline | ❌ No SQL support, ❌ Larger app size |

**When to Use Each Storage Option**

* Use **shared\_preferences** for **small settings or user preferences** (e.g., theme mode, login state).
* Use **SQLite** for **structured, relational data** (e.g., storing contacts, orders, user details).
* Use **Hive** for **storing large, unstructured data** quickly (e.g., offline cache, chat messages).

**2. CRUD Operations in SQLite and Hive**

**What is CRUD?**

CRUD stands for **Create, Read, Update, and Delete** – the four essential database operations.

**Using SQLite for CRUD Operations**

SQLite is a structured, SQL-based database used in Flutter via the sqflite package.

**Step 1: Add Dependencies in pubspec.yaml**

dependencies:

sqflite: ^2.0.0

path\_provider: ^2.0.0

**Step 2: Initialize SQLite Database**

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

class DatabaseHelper {

static Future<Database> initDB() async {

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

return openDatabase(

path,

version: 1,

onCreate: (db, version) {

return db.execute(

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

);

},

);

}

}

**Step 3: Implement CRUD Operations in SQLite**

class User {

int id;

String name;

int age;

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

Map<String, dynamic> toMap() {

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

}

}

// CREATE Operation

Future<void> insertUser(User user) async {

final db = await DatabaseHelper.initDB();

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

}

// READ Operation

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

final db = await DatabaseHelper.initDB();

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']);

});

}

// UPDATE Operation

Future<void> updateUser(User user) async {

final db = await DatabaseHelper.initDB();

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

}

// DELETE Operation

Future<void> deleteUser(int id) async {

final db = await DatabaseHelper.initDB();

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

}

**Using Hive for CRUD Operations**

Hive is a NoSQL database optimized for performance and works with key-value pairs.

**Step 1: Add Dependencies in pubspec.yaml**

dependencies:

hive: ^2.2.3

hive\_flutter: ^1.1.0

dev\_dependencies:

hive\_generator: ^2.0.0

build\_runner: ^2.1.7

**Step 2: Initialize Hive**

import 'package:hive/hive.dart';

import 'package:hive\_flutter/hive\_flutter.dart';

void main() async {

await Hive.initFlutter();

runApp(MyApp());

}

**Step 3: Create and Use Hive Box**

class User {

final String name;

final int age;

User(this.name, this.age);

}

// CREATE Operation

Future<void> addUser(User user) async {

final box = await Hive.openBox('userBox');

box.put('user', user);

}

// READ Operation

Future<User?> getUser() async {

final box = await Hive.openBox('userBox');

return box.get('user');

}

// UPDATE Operation

Future<void> updateUser(User user) async {

final box = await Hive.openBox('userBox');

box.put('user', user);

}

// DELETE Operation

Future<void> deleteUser() async {

final box = await Hive.openBox('userBox');

box.delete('user');

}

**3. Advantages and Use Cases for shared\_preferences**

**What is shared\_preferences?**

**shared\_preferences** is a simple key-value storage system for saving small amounts of persistent data like user settings.

**Advantages of shared\_preferences**

✅ **Simple to Use** – No database setup required.  
✅ **Fast** – Reads and writes data quickly.  
✅ **Persistent Storage** – Data remains saved even after restarting the app.  
✅ **Lightweight** – Uses minimal storage.

**Use Cases for shared\_preferences**

* **Storing User Preferences** (e.g., dark mode, language settings).
* **Saving Login States** (e.g., remember me functionality).
* **Caching Small Data** (e.g., last entered search query).

**How to Use shared\_preferences in Flutter?**

**Step 1: Add Dependency in pubspec.yaml**

dependencies:

shared\_preferences: ^2.0.0

**Step 2: Save and Retrieve Data**

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

// Save Data

Future<void> saveData() async {

SharedPreferences prefs = await SharedPreferences.getInstance();

prefs.setString('username', 'JohnDoe');

}

// Retrieve Data

Future<String?> getData() async {

SharedPreferences prefs = await SharedPreferences.getInstance();

return prefs.getString('username');

}

**Step 3: Remove Data**

Future<void> removeData() async {

SharedPreferences prefs = await SharedPreferences.getInstance();

prefs.remove('username');

}