Module 8: Local Storage and Persistence

Theory Assignments

Name: Chandvaniya Abhay

1. Explain the difference between local storage options (shared_preferences, SQLite, Hive).

1. shared_preferences

Purpose: Store small amounts of simple data
Type: Key–value storage
Persistence: Yes (data remains after app restart)

What it's good for

- App settings and user preferences
- Flags (e.g., isLoggedIn, darkModeEnabled)
- Small strings, numbers, and booleans

Supported data types

- int, double, bool, String
- List<String>

Pros

- Very easy to use
- Lightweight
- Built into Flutter ecosystem

Cons

- Not suitable for large data
- No complex objects
- No querying or relationships

Example use cases

- Theme selection
- Language preference
- First-time app launch flag

2. SQLite (via sqflite)

Purpose: Store structured, relational data
Type: Relational database
Persistence: Yes

What it's good for

- Complex data models
- Large datasets
- Data with relationships (tables, foreign keys)

Supported data types

- Integers, text, blobs, etc. (via SQL schema)
- Complex data through normalization

Pros

- Powerful querying with SQL
- Supports indexing and joins
- Reliable and widely used

Cons

- More boilerplate code
- Manual schema design and migrations
- Slower to develop compared to NoSQL options

Example use cases

- Offline-first apps
- Financial records
- Chat history or logs

3. Hive

Purpose: Fast local NoSQL database
Type: Key–value / object storage
Persistence: Yes

What it's good for

- Storing Dart objects
- Medium to large datasets
- High-performance local storage

Supported data types

- Primitive types
- Lists, maps
- Custom objects (with TypeAdapters)

Pros

- Very fast (no SQL parsing)
- Easy to store objects
- No native dependencies
- Simple API

Cons

- No complex queries like SQL joins
- Requires adapters for custom objects
- Less flexible for relational data

Example use cases

- Cached API responses
- User profiles
- Offline app state

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

1. CRUD in SQLite (using sqflite)

SQLite is a **relational database**, so CRUD operations are performed using **SQL queries**.

Example Table

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

Create (INSERT)

```
await db.insert(
  'users',
  {'name': 'Alice', 'age': 25},
);
```

SQL equivalent

```
INSERT INTO users (name, age) VALUES ('Alice', 25);
```

Read (SELECT)

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

With condition

```
await db.query(
  'users',
  where: 'age > ?',
  whereArgs: [18],
);
```

Update (UPDATE)

```
await db.update(
  'users',
  {'age': 26},
  where: 'id = ?',
  whereArgs: [1],
);
```

Delete (DELETE)

```
await db.delete(
  'users',
  where: 'id = ?',
  whereArgs: [1],
);
```

Key Points for SQLite

- Uses **SQL syntax**
- Supports **complex queries**, joins, and indexes
- Best for **structured relational data**
- Requires schema & migrations

2. CRUD in Hive

Hive is a **NoSQL key–value database**, so CRUD operations are simpler and object-focused.

Example Model

```
@HiveType(typeId: 0)
class User {
  @HiveField(0)
  String name;

  @HiveField(1)
  int age;

  User(this.name, this.age);
}
```

Create (ADD / PUT)

```
var box = Hive.box<User>('users');
await box.add(User('Alice', 25));
```

With custom key

```
await box.put('user_1', User('Alice', 25));
```

Read (GET)

```
User? user = box.get('user_1');
```

Read all

```
List<User> users = box.values.toList();
```

Update

```
await box.put('user_1', User('Alice', 26));
```

Update by index

```
User user = box.getAt(0)!;
user.age = 26;
await user.save();
```

Delete

```
await box.delete('user_1');
```

Delete by index

```
await box.deleteAt(0);
```

Key Points for Hive

- No SQL required
- Stores **Dart objects directly**
- Extremely fast
- Limited querying (no joins)

3. Explain the advantages and use cases for shared_preferences.

Advantages of shared_preferences

1. Simple and Easy to Use

- Minimal setup
- No database schema or models
- Ideal for beginners and quick implementations

```
await prefs.setBool('isLoggedIn', true);
```

2. Persistent Storage

- Data remains available after:
  - App restarts
  - App closures
- Useful for maintaining app state

3. Lightweight & Low Overhead

- Uses native storage:
  - **Android:** SharedPreferences
  - **iOS:** NSUserDefaults
- Very low memory and storage usage

4. Cross-Platform Consistency

- Same API for Android, iOS, Web, and Desktop
- Platform differences handled internally

5. No External Dependencies

- No need for databases, migrations, or adapters
- Fast access for small data

Supported Data Types

- bool
- int
- double
- String
- List<String>

Not suitable for complex objects or large datasets

Common Use Cases

1. User Preferences

- Theme mode (dark/light)
- Language selection
- Notification settings

```
prefs.setString('theme', 'dark');
```

2. App State Flags

- First-time launch detection
- Onboarding completion
- Feature toggles

```
prefs.setBool('isFirstLaunch', false);
```

3. Authentication State

- Login status
- Remember-me option (not for sensitive tokens)

```
prefs.setBool('loggedIn', true);
```

4. UI & UX Settings

- Font size
- Layout preferences
- Last selected tab

5. Simple Caching

- Last search query
- Recently viewed item IDs (small list)

```
prefs.setStringList('recentSearches', searches);
```

Limitations (When NOT to Use)

- Large datasets
- Complex data structures
- Frequent write operations
- Sensitive data (use secure storage instead)