Mobile Programming

Local Storage & Data Persistence

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

SharedPreferences | SQLite | Secure Storage | Hive



Table of Contents

1. Why Local Storage Matters
2. Storage Options Overview
3. SharedPreferences / UserDefaults
4. SQLite with sqflite
5. Drift — Type-Safe SQLite
6. Hive — NoSQL Key-Value Store
7. Flutter Secure Storage
8. File System Access
9. Offline-First Architecture
10. Mini Lab: Encrypted Notes App
11. Summary & Key Takeaways

Why Local Storage Matters

Mobile apps must work **offline** and persist data across sessions.

Reasons for Local Storage:

- **Offline support:** Users expect apps to work without internet.
- **Performance:** Local data is faster than network calls.
- **State persistence:** Remember user preferences, login state.
- **Data caching:** Reduce bandwidth and server load.
- **Security:** Sensitive data must be stored securely.

> **Golden Rule:** Never assume the network is available. Always have a local fallback.

Storage Options Overview

Storage	Type	Use Case	Encrypted
SharedPreferences	Key-Value	Settings, flags, simple cache	No
SQLite (sqflite)	Relational DB	Structured data, queries, relations	Manual
Drift	Type-safe SQLite	Complex DB with migrations	Optional
Hive	NoSQL Key-Value	Fast, simple object storage	AES-256
Secure Storage	Encrypted KV	Tokens, passwords, keys	Yes (Keychain/Keystore)
Files	Raw files	Images, PDFs, downloads	No

SharedPreferences

> Simple key-value storage for primitives. Wraps Android's `SharedPreferences` and iOS's `UserDefaults`.

SharedPreferences

```dart
import 'package:shared_preferences/shared_preferences.dart';
class SettingsService {
  late final SharedPreferences _prefs;
  Future<void> init() async {
    _prefs = await SharedPreferences.getInstance();
  }
  // Read
  String? get theme => _prefs.getString('theme');
  bool get isDarkMode => _prefs.getBool('dark_mode') ?? false;
  int get launchCount => _prefs.getInt('launch_count') ?? 0;
  // Write
  Future<void> setTheme(String theme) => _prefs.setString('theme', theme);
  Future<void> setDarkMode(bool value) => _prefs.setBool('dark_mode', value);
  // Increment
  Future<void> incrementLaunchCount() async {
    final count = _prefs.getInt('launch_count') ?? 0;
    await _prefs.setInt('launch_count', count + 1);
  }
  // Remove
  Future<void> clear() => _prefs.clear();
}
```

SQLite with sqflite

> Full relational database for structured data with queries, joins, and transactions.

```dart
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(), 'myapp.db');
    return openDatabase(
      path,
      version: 1,
      onCreate: (db, version) async {
        await db.execute('''
          CREATE TABLE notes (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,...)
        ''');
      },
    );
```

SQLite — CRUD Operations

```
class NoteRepository {
  final DatabaseHelper _dbHelper;
  NoteRepository(this._dbHelper);
  ...
}
```

SQLite — CRUD Operations (create)

```dart
Future<int> insert(Note note) async {
  final db = await _dbHelper.database;
  return db.insert('notes', note.toMap());
}
```

SQLite — CRUD Operations (update)

```dart
Future<int> update(Note note) async {
  final db = await _dbHelper.database;
  return db.update('notes', note.toMap(),
    where: 'id = ?', whereArgs: [note.id]);
}
```

SQLite — CRUD Operations (delete)

```dart
Future<int> delete(int id) async {
  final db = await _dbHelper.database;
  return db.delete('notes', where: 'id = ?', whereArgs: [id]);
}
```

SQLite — CRUD Operations (search - getAll)

```dart
Future<List<Note>> search(String query) async {
  final db = await _dbHelper.database;
  final maps = await db.query('notes',
    where: 'title LIKE ? OR body LIKE ?',
    whereArgs: ['%$query%', '%$query%']);
  return maps.map((m) => Note.fromMap(m)).toList();
}

Future<List<Note>> getAll() async {
  final db = await _dbHelper.database;
  final maps = await db.query('notes', orderBy: 'created_at DESC');
  return maps.map((m) => Note.fromMap(m)).toList();
}
```

SQLite — Migrations

Database schemas evolve. Handle version upgrades properly.

```
Future<void> _v2Migration(Database db) async {
  await db.execute('ALTER TABLE notes ADD COLUMN is_encrypted INTEGER DEFAULT 0');
}

Future<void> _v3Migration(Database db) async {
  await db.execute('ALTER TABLE notes ADD COLUMN category TEXT DEFAULT "general"');
}
```

SQLite — Migrations

```dart
openDatabase(
  path,
  version: 3,
  onCreate: (db, version) async {
    await db.execute('''
      CREATE TABLE notes (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        body TEXT NOT NULL,
        created_at INTEGER NOT NULL
      )
    ''');
    // Run all migrations for fresh install
    await _v2Migration(db);
    await _v3Migration(db);
  },
  onUpgrade: (db, oldVersion, newVersion) async {
    if (oldVersion < 2) await _v2Migration(db);
    if (oldVersion < 3) await _v3Migration(db);
  },
);
```

Hive — NoSQL Key-Value Store

Fast, lightweight NoSQL database written in pure Dart. Great for simple objects.

```
dependencies:
  hive: ^2.2.3
  hive_flutter: ^1.1.0
```

Hive — NoSQL Key-Value Store

```dart
// Initialize
await Hive.initFlutter();
// Register adapter (for custom objects)
Hive.registerAdapter(NoteAdapter());
// Open a box
final notesBox = await Hive.openBox<Note>('notes');
// CRUD
await notesBox.put('key1', note);          // Create/Update
final note = notesBox.get('key1');          // Read
await notesBox.delete('key1');              // Delete
final allNotes = notesBox.values.toList();  // Read all
// Listen to changes
notesBox.watch().listen((event) {
  print('Box changed: ${event.key}');
});
```

Flutter Secure Storage

For sensitive data (tokens, passwords, API keys). Uses platform-specific secure storage.

- **Android:** EncryptedSharedPreferences (MasterKey + AES-256) or Keystore.
- **iOS:** Keychain Services.

```dart
import 'package:flutter_secure_storage/flutter_secure_storage.dart';
class SecureStorageService {
  static const _storage = FlutterSecureStorage(
    aOptions: AndroidOptions(encryptedSharedPreferences: true),
    iOptions: IOSOptions(accessibility: KeychainAccessibility.first_unlock),
  );
  static Future<void> saveToken(String token) => _storage.write(key: 'auth_token', value: token);
  static Future<String?> getToken() => _storage.read(key: 'auth_token');
  static Future<void> deleteToken() => _storage.delete(key: 'auth_token');
  static Future<void> saveCredentials({
    required String email, required String password,
  }) async {
    await _storage.write(key: 'user_email', value: email);
    await _storage.write(key: 'user_password', value: password);
  }
  static Future<void> clearAll() => _storage.deleteAll();
}
```

File System Access

Store files (images, PDFs, downloads) in the app's document or cache directories.

```dart
import 'dart:io';
import 'package:path_provider/path_provider.dart';
class FileService {
  Future<String> get _localPath async {
    final directory = await getApplicationDocumentsDirectory();
    return directory.path;
  }
  Future<File> get _localFile async {
    final path = await _localPath;
    return File('$path/counter.txt');
  }
  Future<String> readCounter() async {
    try {
      final file = await _localFile;
      return await file.readAsString();
    } catch (e) {
      return '0';
    }
  }
  Future<File> writeCounter(int counter) async {
    final file = await _localFile;
    return file.writeAsString('$counter');
```

Offline-First Architecture

Design your app to work offline first, sync when online.

Core Principles

1. **Local = Source of Truth:** UI reads from local DB.
2. **Sync Engine:** Push local changes to server when online.
3. **Conflict Resolution:** Last-write-wins, merge, or user choice.
4. **Queue:** Store pending operations locally.

Data Flow

```
UI → Local DB → Sync Queue → Server
                    ↑              ↓
                Merge ← Remote Data
```

```dart
class SyncService {
  final Queue<Operation> _pendingOps = [];

  Future<void> sync() async {
    if (!await isOnline()) return;
    for (final op in _pendingOps) {
      await _pushToServer(op);
    }
    _pendingOps.clear();
    await _pullFromServer();
  }
}
```

Exercise

Encrypted Notes App

Objective: Build a notes app with encrypted storage.

Requirements:

1. SQLite database with `sqflite`.
2. Notes model with title, body, timestamp.
3. CRUD operations via `NoteRepository`.
4. Encryption per note using `flutter_secure_storage`.
5. Search functionality.
6. Category column.

Summary: Key Takeaways

Storage Options

- `SharedPreferences` — Simple key-value.
- `SQLite/sqflite` — Relational, structured data.
- `Hive` — Fast NoSQL, pure Dart.
- `Secure Storage` — Tokens, passwords.

Patterns

- Repository pattern for data access.
- Migrations for schema evolution.
- Offline-first for reliability.

Security

- Never store sensitive data in plain text.
- Use `flutter_secure_storage` for credentials.
- Encrypt database if it contains PII.

Best Practices

- Initialize storage early (main).
- Handle migration carefully.
- Test with fresh and upgraded DBs.
- Clear cache on logout.

Thank You! 🙌

Questions?

"Why did the developer store his secrets in SharedPreferences?
Because he thought they were 'shared' with his friends only.
Spoiler: They weren't."