Flutter Offline & Caching Strategies

Learn to build offline-first Flutter apps with local storage, caching, and data syncing for seamless user experiences.

Website reference: haas.dev | https://dev-roast-app.vercel.app

---

Introduction

Users expect apps to work even when **internet connectivity is poor or unavailable**. Flutter provides multiple strategies for **offline storage, caching, and syncing local and remote data**, allowing apps to remain functional and reliable at all times.

Step 1: Understanding Offline Storage

Key options for storing data locally in Flutter:

1. **SharedPreferences** – Simple key-value storage (settings, small data).
2. **SQLite** – Structured relational database for more complex offline data.
3. **Hive** – Lightweight, fast, NoSQL database for Flutter apps.
4. **Caching Network Data** – Store API responses locally to reduce repeated network calls.

Step 2: Using SharedPreferences

- Ideal for **storing small, simple data** like user preferences.

```
import 'package:shared_preferences/shared_preferences.dart';

void saveUsername(String username) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.setString('username', username);
}

Future<String?> getUsername() async {
  final prefs = await SharedPreferences.getInstance();
  return prefs.getString('username');
}
```

- Pros: Easy, quick setup.
- Cons: Not suitable for large or relational data

- Cons: Not suitable for large or relational data.

Step 3: Using SQLite

- Store structured offline data with SQL queries.

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

Future<Database> initDb() async {
  return openDatabase(
    join(await getDatabasesPath(), 'app.db'),
    onCreate: (db, version) {
      return db.execute(
        'CREATE TABLE notes(id INTEGER PRIMARY KEY, title TEXT, content TEXT)',
      );
    },
    version: 1,
  );
}
```

- Use `insert`, `query`, `update`, `delete` for data operations.
- Pros: Strong, relational data support.
- Cons: Slightly more setup complexity.

Step 4: Using Hive

- Fast, NoSQL database with **minimal boilerplate**:

```
import 'package:hive/hive.dart';
import 'package:hive_flutter/hive_flutter.dart';

void main() async {
  await Hive.initFlutter();
  var box = await Hive.openBox('myBox');
  await box.put('name', 'Hafsa');
  print(box.get('name'));
}
```

- Pros: Fast, Flutter-friendly, supports offline caching for objects.

- Cons: Non-relational, may need adapters for complex data models.

Step 5: Caching Network Data

- Use cached data when offline to improve user experience.
- Example with **Dio & Hive**:

```
import 'package:dio/dio.dart';
import 'package:hive/hive.dart';

Future<void> fetchData() async {
  var box = await Hive.openBox('cacheBox');
  try {
    var response = await Dio().get('https://api.example.com/data');
    await box.put('cachedData', response.data);
  } catch (e) {
    print('Offline, using cached data: ${box.get('cachedData')}');
  }
}
```

- Ensures app works even without connectivity.

Step 6: Data Syncing Strategies

1. **Local-first approach**: Read/write to local DB first, sync with server when online.
2. **Conflict resolution**: Merge changes from server and local storage.
3. **Background sync**: Use background services to update data without user interaction.

Step 7: Practical Exercise

Objective: Build a Flutter notes app with offline support:

1. Use **Hive or SQLite** to store notes locally.
2. Fetch updates from a remote API and **cache them**.
3. Implement background sync to update local notes periodically.
4. Handle offline scenarios gracefully (show cached notes when offline).

Step 8: Key Takeaways

Step 5: Key Takeaways

- Offline-first apps **improve reliability and UX**.
- Choose the right storage: **SharedPreferences**, **SQLite**, or **Hive** depending on data complexity.
- Always cache network data for offline access.
- Implement **background syncing** for seamless updates.

Visit **haas.dev** for more resources and guides.
https://dev-roast-app.vercel.app
