Flutter: The 4 Ways to Store Data Locally (That You're Going to Need)

Anthony Oleinik Follow
Apr 14, 2021 · 8 min read ★

Credits for this beautiful graphic

Not all data is created equally — and not all data should be stored the same way. Similar to learning about HashMaps, Linked Lists, Heaps, etc., it's important to know the different ways to store data in your Flutter App to create the best user experience possible.

For instance, what happens when you don't need to deserialize a whole object — just a single field or value? It doesn't make sense grabbing a whole `config.json` from memory when all you need is a single boolean field, like `isDarkMode=false`. On the contrary, it doesn't make sense storing large, single objects in table — If you're only ever going to have 1 row, what's the point of designing a whole schema?

For each of the 4 ways I'll be talking about local data storage, I'll go over:

- What niche the storage fills

- Realistic use cases
- Examples w/ package recommendations

If you're about to write a Flutter app, I recommend a bookmark to this page so that you can always refer to it when trying to figure out how to store data.

Simple Key Value — Shared Preferences

Say a user opens your app; they open up the settings and change the app to dark mode. They come back a minute or two later, only to be blinded by the bright flash that is light mode.

The user writes a scathing 1-star review of your app and blames his vision loss on the fact that you didn't store that he was in dark mode.

Somehow, we need to store little details like that that don't require large objects or highly structured databases — simple fields, like "IsDarkMode" or "LastTabUsed" or similar. Things that don't require secure storage, but are nice to have and are generally quite tiny (think single fields).

This data is perfect for `SharedPreferences` ! While this is a flutter package, it wraps native API's for iOS and Android that are designed to do exactly what I just described. This data is quick to retrieve but is not secure, and balks at extremely high loads.

I use this in my app, among other things, to store some local settings (did you disable a gesture? do you want to disable auto-correct when writing messages?) Keep in mind that in memory-constrained environments, the system won't hesitate to delete these if they take up a lot of space — meaning none of your shared preferences should be vital functionality of the app.

Shared Preferences Example (Stolen from docs):

```
SharedPreferences prefs = await SharedPreferences.getInstance();
  int counter = (prefs.getInt('counter') ?? 0) + 1;
  print('Pressed $counter times.');
  await prefs.setInt('counter', counter);
```

Sensitive Key Value — Secure Storage

In contrast to the previous `SharedPreferences` , enter `flutter_secure_storage` . This is another package that is a wrapper over both iOS and Android API's.

Imagine someone breaks into a user's phone and checks the data behind `shared_preferences` — they'd see trivial data that doesn't matter at all, and doesn't really help the hacker get an edge in any way.

Now imagine you store a refresh token or a local pin in the shared preference — the hacker now can use these credentials to impersonate the user, and thus has successfully hacked into your app.

This is a situation that us developers should do our best to avoid — albeit unlikely, it's always possible that a phone gets stolen, and we should take the necessary precautions against a situation like this.

Enter secure storage. Secure storage is designed for things that are sensitive to the user. It's also a simple key value store like the previous option — meaning it's not designed for big data or large values.

(Side tip: If there's a large object that you need to store in memory, but it's sensitive, it might be best to not put it in Secure Storage: Instead, serialize the object, encrypt it using a cipher likes AES or similar, and store it in file store. Then, store the key in secure storage. That way, if a hacker gets to it, it's still useless to them, but it's totally readable with a call to file storage and secure storage.)

In my apps, I've only *personally* come across a single usecase for secure storage, and that's refresh tokens. I can imagine many other usecases, though: for instance, if you have a client-side app that stores data securely like KeepSafe or similar, you somehow need to compare the password to a hash. You could store the hash in SecureStorage, and then use the in-memory password as a key to decrypt the rest of the data.

Secure Storage Example (Adapted from docs):

```
final storage = new FlutterSecureStorage();
```

```
String myVal = "Thanks for reading my article!";

await storage.write(key: key, value: value);

String readValue = await storage.read(key: key);

assert(myVal == readValue); //passes
```

Highly Structured Tables — Hive / SQLite

I prefer to use Hive on the frontend, but it's really up to you — if you enjoy the rigidity that using relational tables affords you, go for it! In general, for lots of data (**at least** 1gb — and this is a low estimate) go the SQLite path (or, if you're bold, isar) otherwise, you're perfectly fine using Hive.

If a user is in an active group chat, you don't want to re-fetch from the backend all 3k+ messages prior to loading the app — this creates needlessly long wait times and requires remote connection every load. It also doesn't make sense to have to load all the data in one burst — for instance, why would you load a message from 2 years ago if the user is only looking at the messages sent today?

These are the usecases for table storages: many objects that can be loaded in parts — data that is structured enough for you to only be provided a couple keys and find what you're looking for, but you can't realistically store it all in one object.

If I have more than one row, (i.e. I'm storing many user's data, not just the one using the app) I always go the path of creating a table for that data.

No example for this one, because there's a lot of packages that serve this purpose. Instead, here are the pub.dev links of 3 that I recommend:

- hive — Key Value Database
- isar —**ALPHA** Large, Flexible Relational Databases (made by Hive creator)
- sqflite — SQLite based database with plain SQL queries

Large Temporary/Permanent Single Object — Local File Storage

This is the first one that doesn't *require* a package, but having path_provider installed will make your life a whole, whole lot easier when using this method of file storage.

Key value is great for a bunch of unrelated data — but what happens when you need to store a user's first name… and last name, and date of birth, and nickname… It wouldn't make sense to store this in a key value format — and it wouldn't make sense to create a whole local database table to only store your local user.

Enter Local File Storage! This is exactly what it sounds like — you can store files locally. For my usecases, I usually serialize objects to jsons, and then store them as a json object in memory. This allows me to serialize and de-serialize with ease without losing any data, and I can go as large with my objects as I'd like.

There's two different types of file stores in this bullet — temporary and permanent.

For temporary, think data that doesn't need to be backed up — you can easily re-fetch from the database if need be, but it would be easier on you and your back-end if the user just stored it locally.

For permanent, this is something that you need to store — this data will never be cleared and consistently lives on the device until app deletion (or you delete it manually!). Unless you're storing copious amounts of data here, you probably won't run into problems, but it's best to choose your directory wisely so that Apple or Android don't get mad at you for abusing their systems.

This, for me, is reserved for singletons or places where I have "constant keys". In my app, I store my local user data here (API Key, Name, etc.) and user data (Time spent in app, messages sent) as well as data that holds the primary keys of my Hive database — when the app loads in from fresh start, and all my hive databases are indexed like `Box('{groupUuid}')` , I need to store all the group uuid's somewhere. I store it in a csv file in temporary local storage (because I can rebuild what groups the user is in from the backend).

Example (Adapted from my own code):

```
final filename = "local_user";
final file = File('${(await
getApplicationDocumentsDirectory()).path}/$filename.json');

file.writeAsString(json.encode(user.toJson()));

User.fromJson(file.readAsString());
```

Bonus! Shared Data — Remote

Here's the bonus one — when talking about where to store data locally, it helps to take a step back and remember that storing data locally is not your only option. You can also store data remotely — on your servers or databases somewhere in the cloud. Generally, if you answer yes to any of these following questions, you're going to want to store in on your backend:

- Are you performing any data analysis on the data?
- Are you sharing this data between users?
- If the user deletes the app, should this data still persist?
- Should this data be shared between devices (mobile, desktop, etc.)?

A single yes to any of those questions means you probably want to atleast store some of the data on a backend. This doesn't mean you can't have it on the device, though: I frequently use local data storage as a cache of sorts — if the data needs to be shared between users, It's uploaded to the cloud, but there's no point in deleting it locally if the user still wants to see it.

In this way, you may be doing double work, (writing schema for your database and for your frontend) but in the end, we're trying to delight our users so what's a little extra work to us?

That's it! Hopefully, I covered all the basis of storing data locally in Flutter. If you find a mistake, please don't hesitate to leave a comment — I want to correct it ASAP.

I’m always looking to learn, so if there’s something that I missed please leave a comment so that I can learn from you.

If you’re looking for my package recommendations, here’s 7 Flutter packages I love:

https://levelup.gitconnected.com/the-7-flutter-packages-i-cant-live-without-9c18ac8540bd

I also wrote an article on my favorite state management solutions:

https://levelup.gitconnected.com/flutter-state-management-in-2021-when-to-use-what-98722093b8bc