[Link](https://docs.flutter.dev/get-started/fundamentals/local-caching): [https://docs.flutter.dev/get-started/fundamentals/local-cachinghttps://docs.flutter.dev/get-started/fundamentals/local-caching](https://docs.flutter.dev/get-started/fundamentals/local-cachinghttps:/docs.flutter.dev/get-started/fundamentals/local-caching)

Local caching

ntroduction to caching

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#introduction-to-caching)

At its most basic, all caching strategies amount to the same three-step operation, represented with the following pseudocode:

dart

Data? \_cachedData;

Future<Data> get data async {

// Step 1: Check whether your cache already contains the desired data

if (\_cachedData == null) {

// Step 2: Load the data if the cache was empty

\_cachedData = await \_readData();

}

// Step 3: Return the value in the cache

return \_cachedData!;

}

content\_copy

There are many interesting ways to vary this strategy, including the location of the cache, the extent to which you preemptively write values to, or "warm", the cache; and others.

Common caching terminology

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#common-caching-terminology)

Caching comes with its own terminology, some of which is defined and explained below.

**Cache hit**

An app is said to have had a cache hit when the cache already contained their desired information and loading it from the real source of truth was unnecessary.

**Cache miss**

An app is said to have had a cache miss when the cache was empty and the desired data is loaded from the real source of truth, and then saved to the cache for future reads.

Risks of caching data

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#risks-of-caching-data)

An app is said to have a **stale cache** when the data within the source of truth has changed, which puts the app at risk of rendering old, outdated information.

All caching strategies run the risk of holding onto stale data. Unfortunately, the action of verifying the freshness of a cache often takes as much time to complete as fully loading the data in question. This means that most apps tend to only benefit from caching data if they trust the data to be fresh at runtime without verification.

To deal with this, most caching systems include a time limit on any individual piece of cached data. After this time limit is exceeded, would-be cache hits are treated as cache misses until fresh data is loaded.

A popular joke among computer scientists is that "The two hardest things in computer science are cache invalidation, naming things, and off-by-one errors." 😄

Despite the risks, almost every app in the world makes heavy use of data caching. The rest of this page explores multiple approaches to caching data in your Flutter app, but know that all of these approaches can be tweaked or combined for your situation.

Caching data in local memory

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#caching-data-in-local-memory)

The simplest and most performant caching strategy is an in-memory cache. The downside of this strategy is that, because the cache is only held in system memory, no data is retained beyond the session in which it is originally cached. (Of course, this "downside" also has the upside of automatically solving most stale cache problems!)

Due to their simplicity, in-memory caches closely mimic the pseudocode seen above. That said, it is best to use proven design principles, like the [repository pattern](https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30), to organize your code and prevent cache checks like the above from appearing all over your code base.

Imagine a UserRepository class that is also tasked with caching users in memory to avoid duplicate network requests. Its implementation might look like this:

dart

class UserRepository {

UserRepository(this.api);

final Api api;

final Map<int, User?> \_userCache = {};

Future<User?> loadUser(int id) async {

if (!\_userCache.containsKey(id)) {

final response = await api.get(id);

if (response.statusCode == 200) {

\_userCache[id] = User.fromJson(response.body);

} else {

\_userCache[id] = null;

}

}

return \_userCache[id];

}

}

content\_copy

This UserRepository follows multiple proven design principles including:

* [dependency injection](https://en.wikipedia.org/wiki/Dependency_injection), which helps with testing
* [loose coupling](https://en.wikipedia.org/wiki/Loose_coupling), which protects surrounding code from its implementation details, and
* [separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns), which prevents its implementation from juggling too many concerns.

And best of all, no matter how many times within a single session a user visits pages in your Flutter app that load a given user, the UserRepository class only loads that data over the network *once*.

However, your users might eventually tire of waiting for data to load every time they relaunch your app. For that, you should choose from one of the persistent caching strategies found below.

Persistent caches

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#persistent-caches)

Caching data in memory will never see your precious cache outlive a single user session. To enjoy the performance benefits of cache hits on fresh launches of your application, you need to cache data somewhere on the device's hard drive.

Caching data with shared\_preferences

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#caching-data-with-shared_preferences)

[shared\_preferences](https://pub.dev/packages/shared_preferences) is a Flutter plugin that wraps platform-specific [key-value storage](https://en.wikipedia.org/wiki/Key%E2%80%93value_database) on all six of Flutter's target platforms. Although these underlying platform key-value stores were designed for small data sizes, they are still suitable for a caching strategy for most applications. For a complete guide, see our other resources on using key-value stores.

* Cookbook: [Store key-value data on disk](https://docs.flutter.dev/cookbook/persistence/key-value)
* Video: [Package of the Week: shared\_preferences](https://www.youtube.com/watch?v=sa_U0jffQII)

Caching data with the file system

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#caching-data-with-the-file-system)

If your Flutter app outgrows the low-throughput scenarios ideal for shared\_preferences, you might be ready to explore caching data with your device's file system. For a more thorough guide, see our other resources on file system caching.

* Cookbook: [Read and write files](https://docs.flutter.dev/cookbook/persistence/reading-writing-files)

Caching data with an on-device database

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#caching-data-with-an-on-device-database)

The final boss of local data caching is any strategy that uses a proper database to read and write data. Multiple flavors exist, including relational and non-relational databases. All approaches offer dramatically improved performance over simple files - especially for large datasets. For a more thorough guide, see the following resources:

* Cookbook: [Persist data with SQLite](https://docs.flutter.dev/cookbook/persistence/sqlite)
* SQLite alternate: [sqlite3 package](https://pub.dev/packages/sqlite3)
* Drift, a relational database: [drift package](https://pub.dev/packages/drift)
* Hive CE, a non-relational database: [hive\_ce package](https://pub.dev/packages/hive_ce)
* Remote Caching, a lightweight caching system for API responses: [remote\_caching package](https://pub.dev/packages/remote_caching)

Caching images

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#caching-images)

Caching images is a similar problem space to caching regular data, though with a one-size-fits-all solution. To direct your Flutter app to use the file system to store images, use the [cached\_network\_image package](https://pub.dev/packages/cached_network_image).

* Video: [Package of the Week: cached\_network\_image](https://www.youtube.com/watch?v=fnHr_rsQwDA)

State restoration

[#](https://docs.flutter.dev/get-started/fundamentals/local-caching#state-restoration)

Along with application data, you might also want to persist other aspects of a user's session, like their navigation stack, scroll positions, and even partial progress filling out forms. This pattern is called "state restoration", and is built in to Flutter.