
MOBILE PROGRAMMING
WITH FLUTTER - DAY7
เริ่มพัฒนาแอปพลิเคชันบนมือถือด้วย Flutter
API
REST
JSON
Shared Preferences
HTTP
Back
Next
Content
SQLite
HIVE

Local Storage & Offline-first Thinking

“ต่อให้เน็ตตาย แอปก็ยัง ‘ใช้งานได้’
และพอเน็ตกลับมา แอปก็ ‘Sync’ ได้”

BORNTO DEV

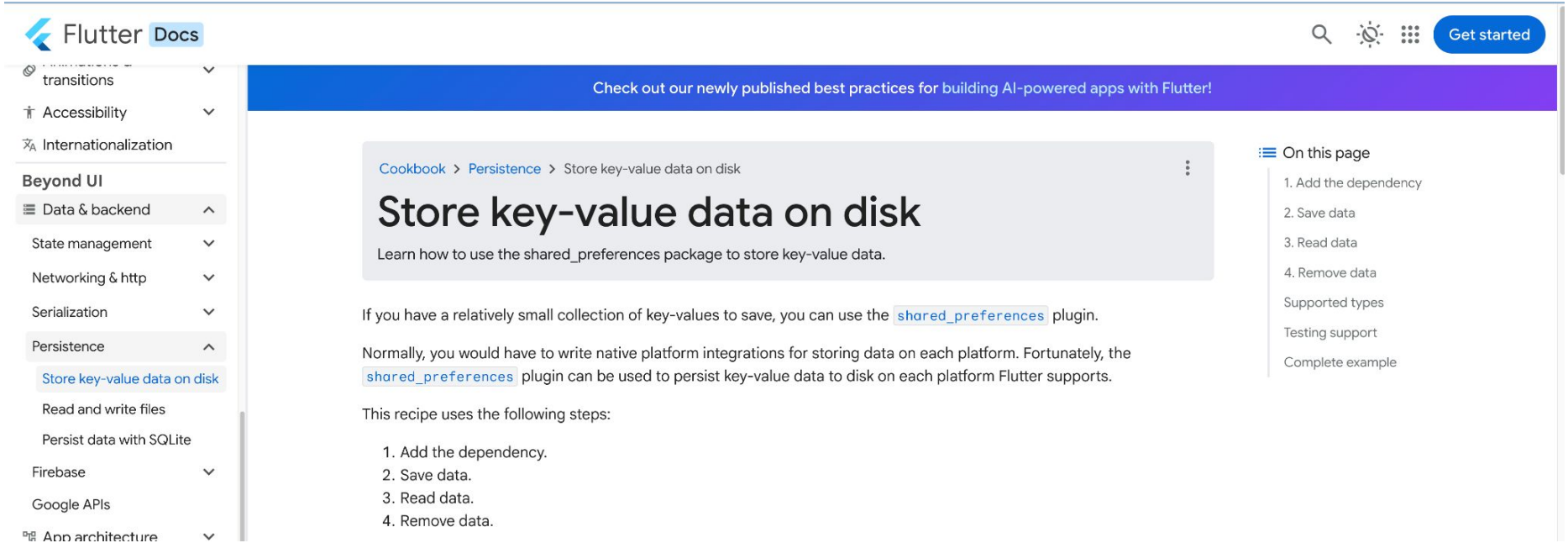
เช็คชื่อรอบนี้

พิมพ์ชื่อ นามสกุล เลขที่นศ และ ตอบคำถาม

อย่างน้อยคนละ 1 คำตอบ

“เราจะเก็บอะไรไว้ในเครื่อง?”

Feature	SharedPreferences	Hive	sqflite
Type	Key-Value	NoSQL (Object-based)	SQL (Relational)
Use Case	Theme, Auth, Tokens	Caching, User Data	Complex Relational Data
Model Support	❌	✅ (via Adapter)	✅ (Manual Map)
Speed	Fast	Very Fast	Slower than Hive
Structured Query	❌	❌	✅
Offline Support	✅	✅	✅
Easy to Set Up	✅	Medium	❌ (Manual DB setup)


Flutter Docs
Get started
transitions
Accessibility
Internationalization
Beyond UI
Data & backend
State management
Networking & http
Serialization
Persistence
Store key-value data on disk
Read and write files
Persist data with SQLite
Firebase
Google APIs
App architecture
Check out our newly published best practices for building AI-powered apps with Flutter!
Cookbook > Persistence > Store key-value data on disk
Store key-value data on disk
Learn how to use the shared_preferences package to store key-value data.
If you have a relatively small collection of key-values to save, you can use the shared_preferences plugin.
Normally, you would have to write native platform integrations for storing data on each platform. Fortunately, the shared_preferences plugin can be used to persist key-value data to disk on each platform Flutter supports.
This recipe uses the following steps:
1. Add the dependency.
2. Save data.
3. Read data.
4. Remove data.
On this page
1. Add the dependency
2. Save data
3. Read data
4. Remove data
Supported types
Testing support
Complete example

1. Add the dependency

Before starting, add the `shared_preferences` package as a dependency.

To add the `shared_preferences` package as a dependency, run `flutter pub add`:

```
flutter pub add shared_preferences
```

2. Save data

To persist data, use the setter methods provided by the `SharedPreferences` class. Setter methods are available for various primitive types, such as `setInt`, `setBool`, and `setString`.

Setter methods do two things: First, synchronously update the key-value pair in memory. Then, persist the data to disk.

dart

```dart
// Load and obtain the shared preferences for this app.
final prefs = await SharedPreferences.getInstance();

// Save the counter value to persistent storage under the 'counter' key.
await prefs.setInt('counter', counter);
```

3. Read data

To read data, use the appropriate getter method provided by the `SharedPreferences` class. For each setter there is a corresponding getter. For example, you can use the `getInt`, `getBool`, and `getString` methods.

```dart
final prefs = await SharedPreferences.getInstance();

// Try reading the counter value from persistent storage.
// If not present, null is returned, so default to 0.
final counter = prefs.getInt('counter') ?? 0;
```

Note that the getter methods throw an exception if the persisted value has a different type than the getter method expects.

4. Remove data

To delete data, use the `remove()` method.

```dart
final prefs = await SharedPreferences.getInstance();

// Remove the counter key-value pair from persistent storage.
await prefs.remove('counter');
```

Supported types

Although the key-value storage provided by `shared_preferences` is easy and convenient to use, it has limitations:

- Only primitive types can be used: `int`, `double`, `bool`, `String`, and `List<String>`.
- It's not designed to store large amounts of data.
- There is no guarantee that data will be persisted across app restarts.

Testing support

It's a good idea to test code that persists data using `shared_preferences`. To enable this, the package provides an in-memory mock implementation of the preference store.

To set up your tests to use the mock implementation, call the `setMockInitialValues` static method in a `setUpAll()` method in your test files. Pass in a map of key-value pairs to use as the initial values.

```dart
SharedPreferences.setMockInitialValues(<String, Object>{'counter': 2});
```

```
(base) Kittikorns-MacBook-Pro:flutter_application_1 kittikornprasertsak$ flutter pub add shared_preferences
  material_color_utilities 0.11.1 (0.13.0 available)
+ path_provider_linux 2.2.1
+ path_provider_platform_interface 2.1.2
+ path_provider_windows 2.3.0
+ platform 3.1.6
+ plugin_platform_interface 2.1.8
+ shared_preferences 2.5.4
```

Complete example

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
```

อ่าน และ เขียนไฟล์

Read and write files

How to read from and write to files on disk.

In some cases, you need to read and write files to disk. For example, you might need to persist data across app launches, or download data from the internet and save it for later offline use.

To save files to disk on mobile or desktop apps, combine the `path_provider` plugin with the `dart:io` library.

This recipe uses the following steps:

1. Find the correct local path.
2. Create a reference to the file location.
3. Write data to the file.
4. Read data from the file.

To learn more, watch this Package of the Week video on the `path_provider` package:

On this page

> **Note**
>
> This recipe doesn't work with web apps at this time. To follow the discussion on this issue, check out `flutter/flutter` issue #45296.

1. Find the correct local path

This example displays a counter. When the counter changes, write data on disk so you can read it again when the app loads. Where should you store this data?

The `path_provider` package provides a platform-agnostic way to access commonly used locations on the device's file system. The plugin currently supports access to two file system locations:

Temporary directory

A temporary directory (cache) that the system can clear at any time. On iOS, this corresponds to the `NSCachesDirectory`. On Android, this is the value that `getCacheDir()` returns.

Documents directory

A directory for the app to store files that only it can access. The system clears the directory only when the app is deleted. On iOS, this corresponds to the `NSDocumentDirectory`. On Android, this is the `AppData` directory.

This example stores information in the documents directory. You can find the path to the documents directory as follows:

```dart
import 'package:path_provider/path_provider.dart';
  // ···
  Future<String> get _localPath async {
    final directory = await getApplicationDocumentsDirectory();

    return directory.path;
  }
```

2. Create a reference to the file location

Once you know where to store the file, create a reference to the file's full location. You can use the `File` class from the `dart:io` library to achieve this.

```dart
Future<File> get _localFile async {
  final path = await _localPath;
  return File('$path/counter.txt');
}
```

3. Write data to the file

Now that you have a `File` to work with, use it to read and write data. First, write some data to the file. The counter is an integer, but is written to the file as a string using the `'$counter'` syntax.

```dart
Future<File> writeCounter(int counter) async {
  final file = await _localFile;

  // Write the file
  return file.writeAsString('$counter');
}
```

4. Read data from the file

Now that you have some data on disk, you can read it. Once again, use the `File` class.

```dart
Future<int> readCounter() async {
  try {
    final file = await _localFile;

    // Read the file
    final contents = await file.readAsString();

    return int.parse(contents);
  } catch (e) {
    // If encountering an error, return 0
    return 0;
  }
}
```

SQLite

Persist data with SQLite

How to use SQLite to store and retrieve data.

> **Note**
>
> This guide uses the sqflite package. This package only supports apps that run on macOS, iOS, or Android.

If you are writing an app that needs to persist and query large amounts of data on the local device, consider using a database instead of a local file or key-value store. In general, databases provide faster inserts, updates, and queries compared to other local persistence solutions.

Flutter apps can make use of the SQLite databases via the `sqflite` plugin available on pub.dev. This recipe demonstrates the basics of using `sqflite` to insert, read, update, and remove data about various Dogs.

If you are new to SQLite and SQL statements, review the SQLite Tutorial to learn the basics before completing this recipe.

This recipe uses the following steps:

1. Add the dependencies.
2. Define the `Dog` data model.
3. Open the database.
4. Create the `dogs` table.
5. Insert a `Dog` into the database.
6. Retrieve the list of dogs.
7. Update a `Dog` in the database.
8. Delete a `Dog` from the database.

1. Add the dependencies

To work with SQLite databases, import the `sqflite` and `path` packages.

- The `sqflite` package provides classes and functions to interact with a SQLite database.
- The `path` package provides functions to define the location for storing the database on disk.

To add the packages as a dependency, run `flutter pub add`:

```
$ flutter pub add sqflite path
```

Make sure to import the packages in the file you'll be working in.

```dart
import 'dart:async';

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

2. Define the Dog data model

Before creating the table to store information on Dogs, take a few moments to define the data that needs to be stored. For this example, define a Dog class that contains three pieces of data: A unique `id`, the `name`, and the `age` of each dog.

```dart
class Dog {
  final int id;
  final String name;
  final int age;

  const Dog({required this.id, required this.name, required this.age});
}
```

3. Open the database

Before reading and writing data to the database, open a connection to the database. This involves two steps:

1. Define the path to the database file using `getDatabasesPath()` from the `sqflite` package, combined with the `join` function from the `path` package.
2. Open the database with the `openDatabase()` function from `sqflite`.

> **Note**
>
> In order to use the keyword `await`, the code must be placed inside an `async` function. You should place all the following table functions inside `void main() async {}`.

```dart
// Avoid errors caused by flutter upgrade.
// Importing 'package:flutter/widgets.dart' is required.
WidgetsFlutterBinding.ensureInitialized();
// Open the database and store the reference.
final database = openDatabase(
  // Set the path to the database. Note: Using the `join` function from the
  // `path` package is best practice to ensure the path is correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
);
```

4. Create the `dogs` table

Next, create a table to store information about various Dogs. For this example, create a table called `dogs` that defines the data that can be stored. Each `Dog` contains an `id`, `name`, and `age`. Therefore, these are represented as three columns in the `dogs` table.

1. The `id` is a Dart `int`, and is stored as an `INTEGER` SQLite Datatype. It is also good practice to use an `id` as the primary key for the table to improve query and update times.
2. The `name` is a Dart `String`, and is stored as a `TEXT` SQLite Datatype.
3. The `age` is also a Dart `int`, and is stored as an `INTEGER` Datatype.

For more information about the available Datatypes that can be stored in a SQLite database, see the official SQLite Datatypes documentation.

```dart
final database = openDatabase(
  // Set the path to the database. Note: Using the `join` function from the
  // `path` package is best practice to ensure the path is correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
  // When the database is first created, create a table to store dogs.
  onCreate: (db, version) {
    // Run the CREATE TABLE statement on the database.
    return db.execute(
      'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
    );
  },
  // Set the version. This executes the onCreate function and provides a
  // path to perform database upgrades and downgrades.
  version: 1,
);
```

5. Insert a Dog into the database

Now that you have a database with a table suitable for storing information about various dogs, it's time to read and write data.

First, insert a `Dog` into the `dogs` table. This involves two steps:

1. Convert the `Dog` into a `Map`
2. Use the `insert()` method to store the `Map` in the `dogs` table.

```
class Dog {
  final int id;
  final String name;
  final int age;

  Dog({required this.id, required this.name, required this.age});

  // Convert a Dog into a Map. The keys must correspond to the names of the
  // columns in the database.
  Map<String, Object?> toMap() {
    return {'id': id, 'name': name, 'age': age};
  }

  // Implement toString to make it easier to see information about
  // each dog when using the print statement.
  @override
  String toString() {
    return 'Dog{id: $id, name: $name, age: $age}';
  }
}
```

```dart
class Dog {
  final int id;
  final String name;
  final int age;

  Dog({required this.id, required this.name, required this.age});

  // Convert a Dog into a Map. The keys must correspond to the names of the
  // columns in the database.
  Map<String, Object?> toMap() {
    return {'id': id, 'name': name, 'age': age};
  }

  // Implement toString to make it easier to see information about
  // each dog when using the print statement.
  @override
  String toString() {
    return 'Dog{id: $id, name: $name, age: $age}';
  }
}
```

```dart
// Define a function that inserts dogs into the database
Future<void> insertDog(Dog dog) async {
  // Get a reference to the database.
  final db = await database;

  // Insert the Dog into the correct table. You might also specify the
  // `conflictAlgorithm` to use in case the same dog is inserted twice.
  //
  // In this case, replace any previous data.
  await db.insert(
    'dogs',
    dog.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}
```

```
// Create a Dog and add it to the dogs table
var fido = Dog(id: 0, name: 'Fido', age: 35);

await insertDog(fido);
```

6. Retrieve the list of Dogs

Now that a `Dog` is stored in the database, query the database for a specific dog or a list of all dogs. This involves two steps:

1. Run a `query` against the `dogs` table. This returns a `List<Map>`.
2. Convert the `List<Map>` into a `List<Dog>`.

dart

```dart
// A method that retrieves all the dogs from the dogs table.
Future<List<Dog>> dogs() async {
  // Get a reference to the database.
  final db = await database;

  // Query the table for all the dogs.
  final List<Map<String, Object?>> dogMaps = await db.query('dogs');

  // Convert the list of each dog's fields into a list of `Dog` objects.
  return [
    for (final {'id': id as int, 'name': name as String, 'age': age as int}
        in dogMaps)
      Dog(id: id, name: name, age: age),
  ];
}
```

```
// Now, use the method above to retrieve all the dogs.
print(await dogs()); // Prints a list that include Fido.
```

7. Update a `Dog` in the database

After inserting information into the database, you might want to update that information at a later time. You can do this by using the `update()` method from the `sqflite` library.

This involves two steps:

1. Convert the Dog into a Map.
2. Use a `where` clause to ensure you update the correct Dog.

```dart
Future<void> updateDog(Dog dog) async {
  // Get a reference to the database.
  final db = await database;

  // Update the given Dog.
  await db.update(
    'dogs',
    dog.toMap(),
    // Ensure that the Dog has a matching id.
    where: 'id = ?',
    // Pass the Dog's id as a whereArg to prevent SQL injection.
    whereArgs: [dog.id],
  );
}
```

```
// Update Fido's age and save it to the database.
fido = Dog(id: fido.id, name: fido.name, age: fido.age + 7);
await updateDog(fido);

// Print the updated results.
print(await dogs()); // Prints Fido with age 42.
```

> ⚠ **Warning**
>
> Always use `whereArgs` to pass arguments to a `where` statement. This helps safeguard against SQL injection attacks.
>
> Do not use string interpolation, such as `where: "id = ${dog.id}"`!

8. Delete a `Dog` from the database

In addition to inserting and updating information about Dogs, you can also remove dogs from the database. To delete data, use the `delete()` method from the `sqflite` library.

In this section, create a function that takes an id and deletes the dog with a matching id from the database. To make this work, you must provide a `where` clause to limit the records being deleted.

```dart
Future<void> deleteDog(int id) async {
  // Get a reference to the database.
  final db = await database;

  // Remove the Dog from the database.
  await db.delete(
    'dogs',
    // Use a `where` clause to delete a specific dog.
    where: 'id = ?',
    // Pass the Dog's id as a whereArg to prevent SQL injection.
    whereArgs: [id],
  );
}
```

Example

To run the example:

1. Create a new Flutter project.
2. Add the `sqflite` and `path` packages to your `pubspec.yaml`.
3. Paste the following code into a new file called `lib/db_test.dart`.
4. Run the code with `flutter run lib/db_test.dart`.

```dart
import 'dart:async';

import 'package:flutter/widgets.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

void main() async {
  // Avoid errors caused by flutter upgrade.
  // Importing 'package:flutter/widgets.dart' is required.
  WidgetsFlutterBinding.ensureInitialized();
  // Open the database and store the reference.
  final database = openDatabase(
    // Set the path to the database. Note: Using the `join` function from the
    // `path` package is best practice to ensure the path is correctly
    // constructed for each platform.
    join(await getDatabasesPath(), 'doggie_database.db'),
    // When the database is first created, create a table to store dogs.
    onCreate: (db, version) {
      // Run the CREATE TABLE statement on the database.
      return db.execute(
        'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
      );
```

🧠 When to Use What?

Situation	Best Choice
🪙 Store a simple token or bool flag	SharedPreferences
🛒👤 Save complex user object or cart items	Hive
🗄️🔍 Create an offline database with queries	sqflite