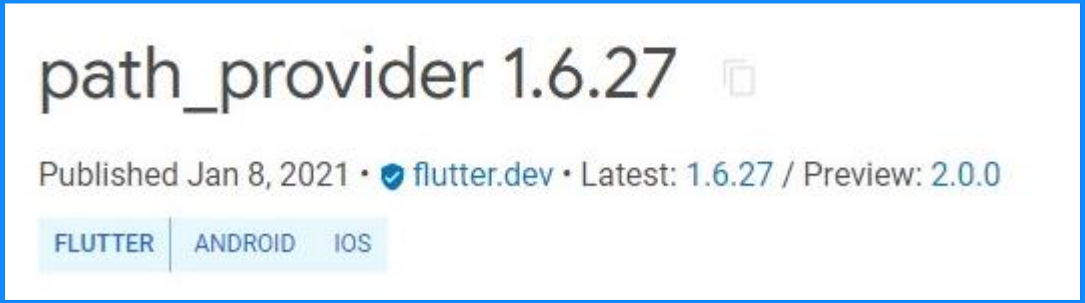
Data storages

Files


path_provider 1.6.27

Published Jan 8, 2021 • flutter.dev • Latest: 1.6.27 / Preview: 2.0.0

FLUTTER | ANDROID IOS

1.

```
dependencies:
    path_provider: ^1.6.27
```

2.

```
$ flutter pub get
```

3.

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

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.

```
 Future<String> get _localPath async {
   final directory = await getApplicationDocumentsDirectory();
   return directory.path;
 }
```

## getTemporaryDirectory();	iOS	Android
## getApplicationDocumentsDirectory();	iOS	Android
## getApplicationSupportDirectory()	iOS	Android
## getLibraryDirectory();	iOS	
## getExternalStorageDirectory();		Android
## getExternalCacheDirectories()		Android

Create file

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

Write data to the file

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

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

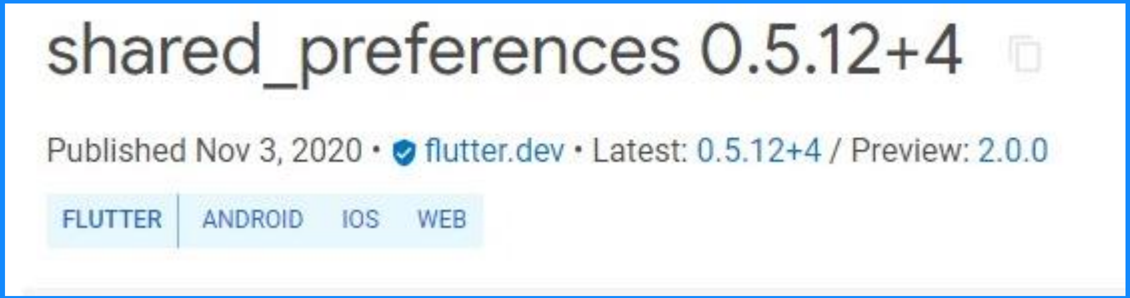
Read data from the file

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

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

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

Shared preferences


shared_preferences 0.5.12+4

Published Nov 3, 2020 • flutter.dev • Latest: 0.5.12+4 / Preview: 2.0.0

FLUTTER | ANDROID IOS WEB

1.

```
dependencies:
    shared_preferences: ^0.5.12+4
```

2.

```
$ flutter pub get
```

3.

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

Save data

```
 final prefs = await SharedPreferences.getInstance();
prefs.setInt('counter', counter);
```

```
 Future<void> _incrementCounter() async {
    final SharedPreferences prefs = await _prefs;
    final int counter = (prefs.getInt('counter') ?? 0) + 1;

    setState(() {
      _counter = prefs.setInt("counter", counter).then((bool success) {
        return counter;
      });
    });
  }
```

Read data

```
final prefs = await SharedPreferences.getInstance();
final counter = prefs.getInt('counter') ?? 0;
```

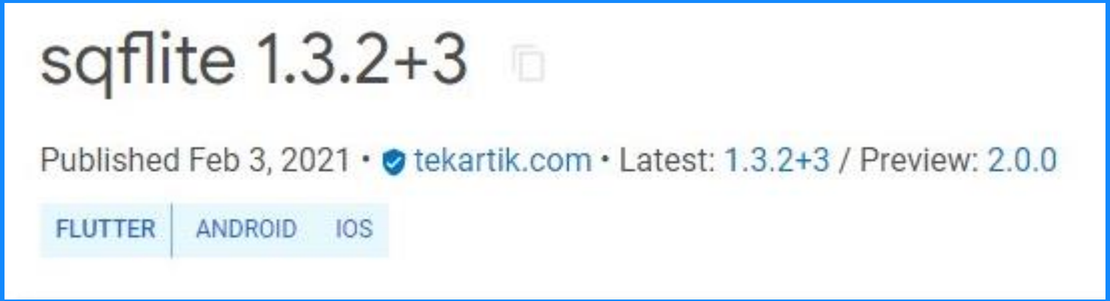
____

```
  @override
  void initState() {
    super.initState();
    _counter = _prefs.then((SharedPreferences prefs) {
      return (prefs.getInt('counter') ?? 0);
    });
  }
```

Remove data

```
final prefs = await SharedPreferences.getInstance();
prefs.remove('counter');
```

SQLite



sqflite 1.3.2+3

Published Feb 3, 2021 • tekartik.com • Latest: 1.3.2+3 / Preview: 2.0.0

1.

```
dependencies:
    sqflite: ^1.3.2+3
```

2.

```
$ flutter pub get
```

3.

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

Model

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

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

Open the database

```
// Open the database and store the reference.
final Future<Database> 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'),
);
```

Create the dogs table

```
final Future<Database> database = openDatabase(
  // Set the path to the database.
  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,
);
```

Insert

```
// Update the Dog class to include a `toMap` method.
class Dog {
  final int id;
  final String name;
  final int age;

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

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

```
// Define a function that inserts dogs into the database
Future<void> insertDog(Dog dog) async {
  // Get a reference to the database.
  final Database 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.
final fido = Dog(
  id: 0,
  name: 'Fido',
  age: 35,
);

await insertDog(fido);
```

Retrieve

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

  // Query the table for all The Dogs.
  final List<Map<String, dynamic>> maps = await db.query('dogs');

  // Convert the List<Map<String, dynamic> into a List<Dog>.
  return List.generate(maps.length, (i) {
    return Dog(
      id: maps[i]['id'],
      name: maps[i]['name'],
      age: maps[i]['age'],
    );
  });
}

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

Update

```
Future<void> updateDog(Dog dog) async {
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.
await updateDog(Dog(
  id: 0,
  name: 'Fido',
  age: 42,
));

print(await dogs()); // Prints Fido with age 42.
```

Delete

```
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],
 );
}
```

Hive

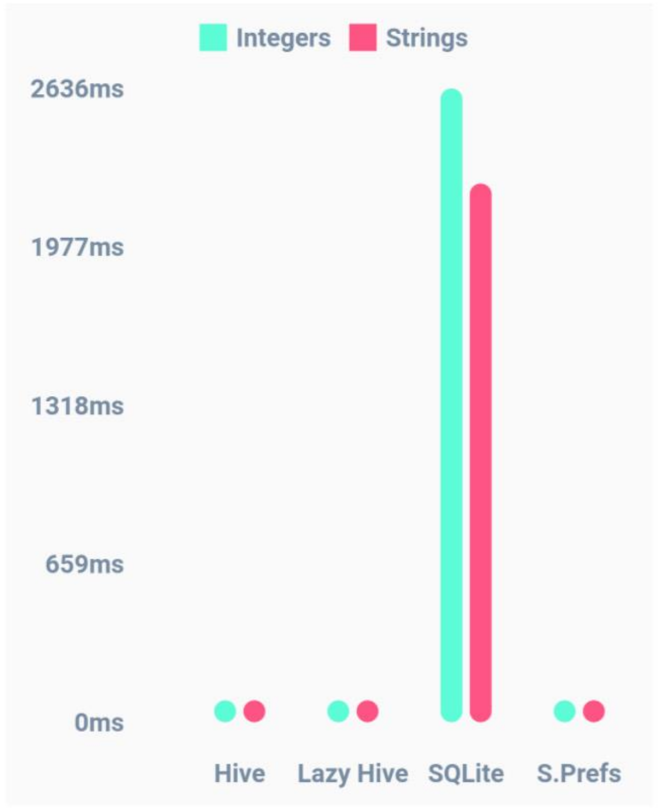
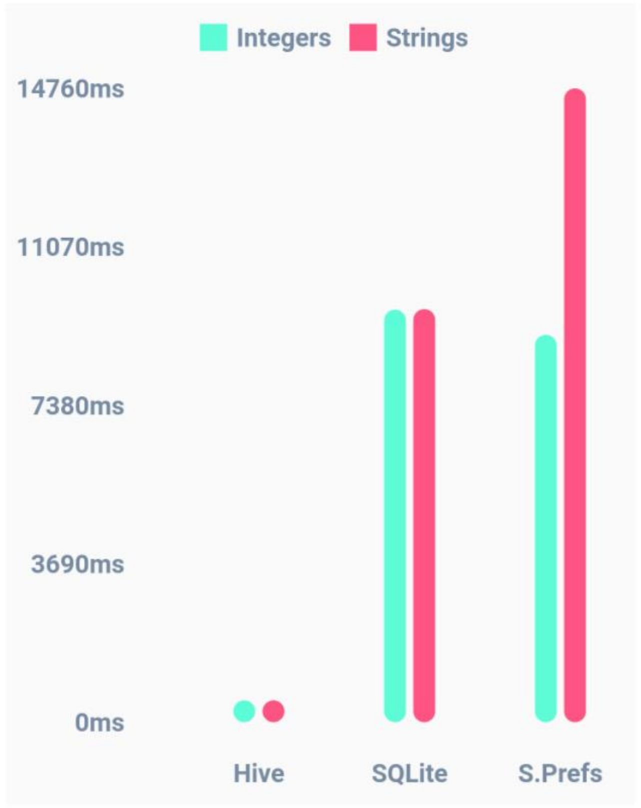
https://docs.hivedb.dev/#/basics/read_write

Hive — noSql база, написанная на чистом Dart, очень быстрая. Кроме этого плюсы Hive:

Кросс-платформенность — так как на чистом Dart и нет нативных зависимостей — mobile, desktop, browser.

Высокая производительность.

Встроенное сильное шифрование.

1000 read iterations	1000 write iterations
	
SharedPreferences is on par with Hive when it comes to read performance. SQLite performs much worse.	Hive greatly outperforms SQLite and SharedPreferences when it comes to writing or deleting.

hive

Dart

Flutter

Где хранятся данные

Все данные, хранящиеся в Hive, организованы в ящики. Ящик можно сравнить с таблицей в SQL, но он не имеет структуры и может содержать что-либо.

Box

Lazy box

Ящики поддерживают шифрование AES-256 и сжатие из коробки

Когда стоит использовать

Key-value databases can be used to store almost any kind of data. For example:

User profiles

Session information

Article/blog comments

Messages

Shopping cart contents

Product categories

Binary data

etc.

Когда не стоит

Если ваши данные имеют сложные отношения и вы в значительной степени полагаетесь на индексы и сложные запросы, вам следует рассмотреть возможность использования SQLite.