

# Programmazione Mobile

**Nicola Noviello**

[nicola.noviello@unimol.it](mailto:nicola.noviello@unimol.it)

Corso di Laurea in Informatica  
Dipartimento di Bioscienze e Territorio  
Università degli Studi del Molise  
Anno 2023/2024

# Lezione: Persistenza dei dati

- key-value data / shared preferences
- Scrittura su file
- Persistenza su SQLite

# Persistenza dei dati

# Cos'è la Persistenza dei Dati

La persistenza dei dati si riferisce alla capacità di un'applicazione di conservare informazioni in modo duraturo, garantendo che i dati non vadano persi quando l'App viene chiusa o il dispositivo viene riavviato. Gestire la persistenza permette ai dati di sopravvivere alla chiusura dell'applicazione e di essere disponibili alla successiva riapertura.

# Componenti chiave

- **Storage Locale:** Utilizzo di database locali (come SQLite, Hive) o di semplici file per conservare i dati sul dispositivo;
- **Storage Remoto:** Utilizzo di servizi cloud (come Firebase) per sincronizzare e conservare i dati su server remoti;
- **Tecniche di Sincronizzazione:** Metodi per mantenere i dati aggiornati e coerenti tra il dispositivo locale e il server remoto.

# Scopo

**Esperienza Utente Migliorata:** Assicura che le preferenze e i progressi dell'utente siano salvati;

**Accesso Offline:** Consente l'uso dell'App senza connessione internet, migliorando la funzionalità e l'affidabilità;

**Consistenza dei Dati:** Garantisce che i dati rimangano consistenti tra sessioni diverse e su più dispositivi.



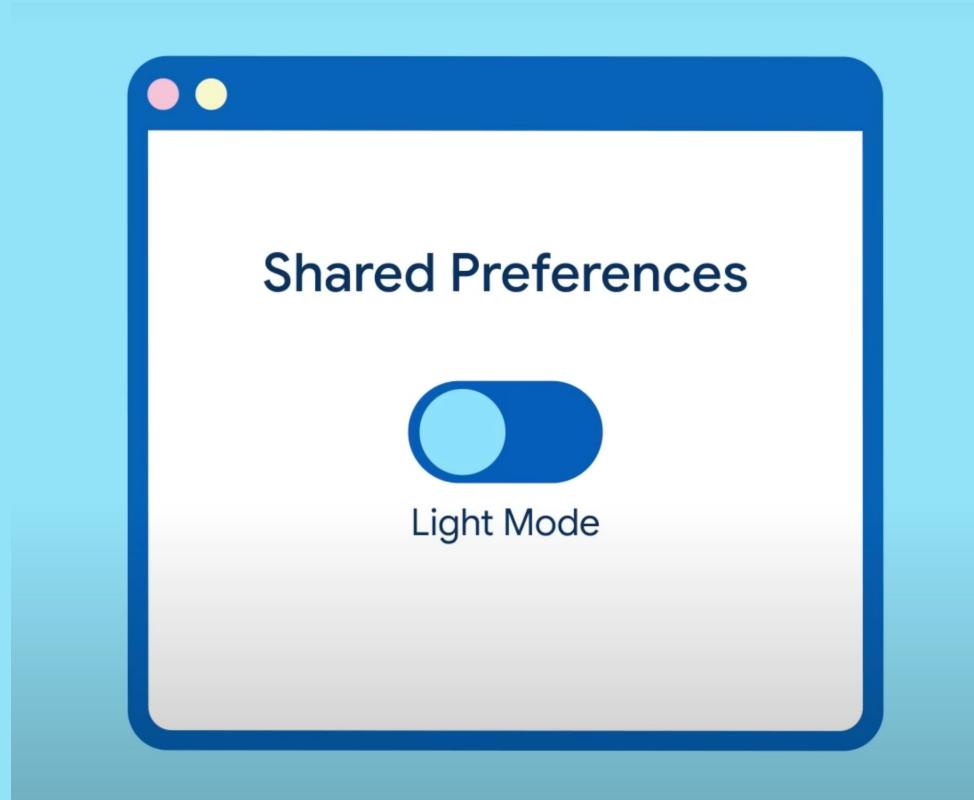
# Salvare key-value su disco

# shared\_preferences -

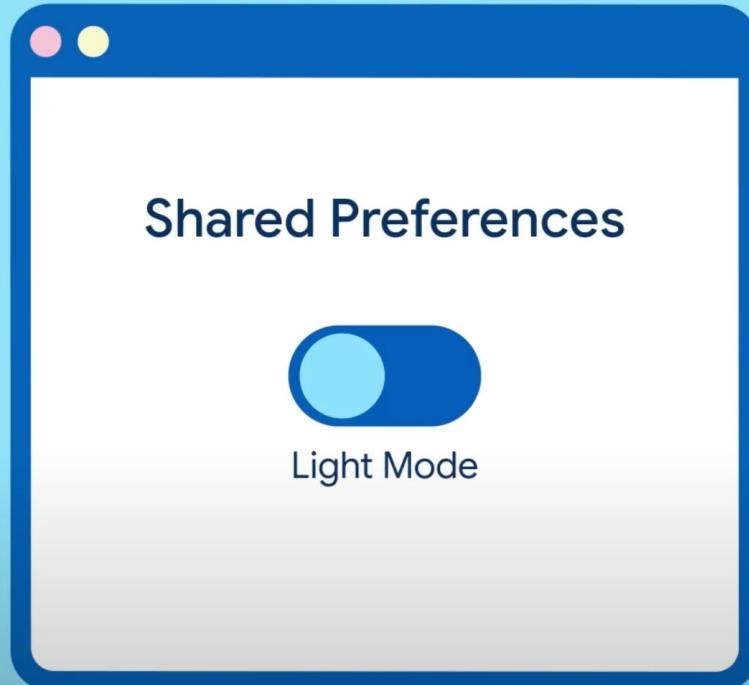
[https://pub.dev/packages/shared\\_preferences](https://pub.dev/packages/shared_preferences)

The screenshot shows the official page for the `shared_preferences` package on `pub.dev`. The page title is `shared_preferences 2.2.3`, published 32 days ago, and is Dart 3 compatible. It features tabs for `Readme`, `Changelog`, `Example`, `Installing`, `Versions`, and `Scores`. The `Readme` tab is selected. The `Readme` content describes the plugin as a Flutter plugin for reading and writing simple key-value pairs. It wraps `NSUserDefaults` on iOS and `SharedPreferences` on Android. The `Readme` also notes that writes are persisted to disk asynchronously. Supported data types are listed as `int`, `double`, `bool`, `String` and `List<String>`. Below this, there are links for `Android`, `iOS`, `Linux`, `macOS`, `Web`, and `Windows`. The `Support` section indicates support for `SDK 16+`, `12.0+`, `Any`, `10.14+`, `Any`, and `Any`. The `Usage` section provides instructions to add the plugin as a dependency in `pubspec.yaml`. Examples and Write data sections follow, along with code snippets for obtaining and saving preferences. The right sidebar contains sections for `Publisher` (`@flutter.dev`), `Metadata` (Flutter plugin for reading and writing simple key-value pairs), `Repository` ([GitHub](#)), `View/report issues`, `Contributing`, `Topics` (`#persistence #shared-preferences #storage`), `Documentation`, `API reference`, `License` ([BSD-3-Clause \(LICENSE\)](#)), and `Dependencies` (`flutter`).

# shared\_preferences

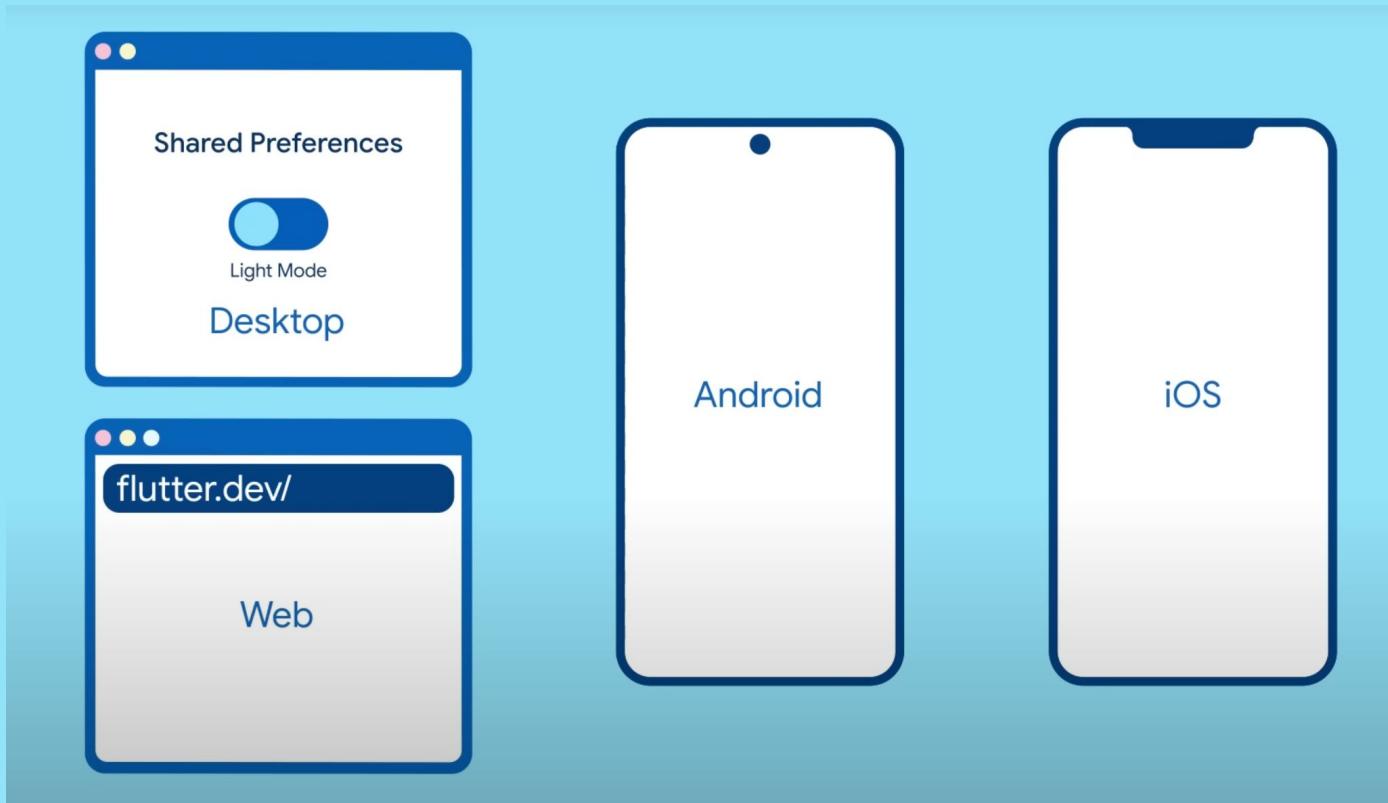


# shared\_preferences



```
user1: {  
    username: sk8erboi1234  
    Password: nicetrybuddy  
    darkModeEnabled: true  
}
```

# shared\_preferences



# shared\_preferences

Platform	Location
Android	Shared Preferences
iOS	NSUserDefaults
Linux	XDG_DATA_HOME
macOS	NSUserDefaults
Windows	AppData
Web	Local Storage

# Salvataggio delle informazioni

Per permettere la persistenza dei dati, vengono utilizzati i metodi setter forniti dalla classe **SharedPreferences**. I metodi setter sono disponibili per vari tipi primitivi, come **setInt**, **setBool** e **setString**.

I metodi setter fanno due cose: prima, aggiornano sincronicamente la coppia chiave-valore in memoria. Poi, salvano i dati su disco.

# Salvataggio delle informazioni

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);
```

# Lettura delle informazioni

Per la lettura dei dati, viene utilizzato il metodo getter appropriato fornito dalla classe **SharedPreferences**. Per ogni setter c'è un corrispondente getter. Ad esempio, è possibile utilizzare i metodi **getInt**, **getBool** e **getString**.

Bisogna utilizzare i metodi nel modo corretto poiché i getter lanciano un'eccezione se il valore salvato ha un tipo diverso da quello che il metodo getter si aspetta.

# Lettura delle informazioni

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;
```

# Cancellare le informazioni

Per cancellare le informazioni viene utilizzato il metodo **remove()**

# Cancellare le informazioni

dart

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

// Remove the counter key-value pair from persistent storage.

await prefs.remove('counter');
```

# Tipi di dati supportati

L'archiviazione a coppie chiave-valore fornita da **shared\_preferences** è facile e conveniente da usare ma presenta delle limitazioni:

- **Possono essere utilizzati solo tipi primitivi:** int, double, bool, String e List<String>;
- Non è progettata per memorizzare grandi quantità di dati;
- *Non c'è garanzia che i dati vengano persistiti attraverso i riavvii dell'App.*

# Progetto di esempio

```
1 import 'package:flutter/material.dart';
2 import 'package:shared_preferences/shared_preferences.dart';
3
4 class MyHomePage extends StatefulWidget {
5   const MyHomePage({super.key, required this.title});
6
7   final String title;
8
9   @override
10  State<MyHomePage> createState() => _MyHomePageState();
11 }
12
13 class _MyHomePageState extends State<MyHomePage> {
14   int _counter = 0;
15
16   @override
17   void initState() {
18     super.initState();
19     _loadCounter();
20   }
21
22   /// Carica il valore iniziale del contatore dallo storage persistente all'avvio,
23   /// o lo setta a zero a 0 se non esiste.
24   Future<void> _loadCounter() async {
25     final prefs = await SharedPreferences.getInstance();
26     setState(() {
27       _counter = prefs.getInt('counter') ?? 0;
28     });
29   }
30
31   /// Dopo un clic, incrementa lo stato del contatore e
32   /// lo salva in modo asincrono nello storage persistente.
33   Future<void> _incrementCounter() async {
34     final prefs = await SharedPreferences.getInstance();
35     setState(() {
36       _counter = (prefs.getInt('counter') ?? 0) + 1;
37       prefs.setInt('counter', _counter);
38     });
39   }
40 }
```

# Progetto di esempio

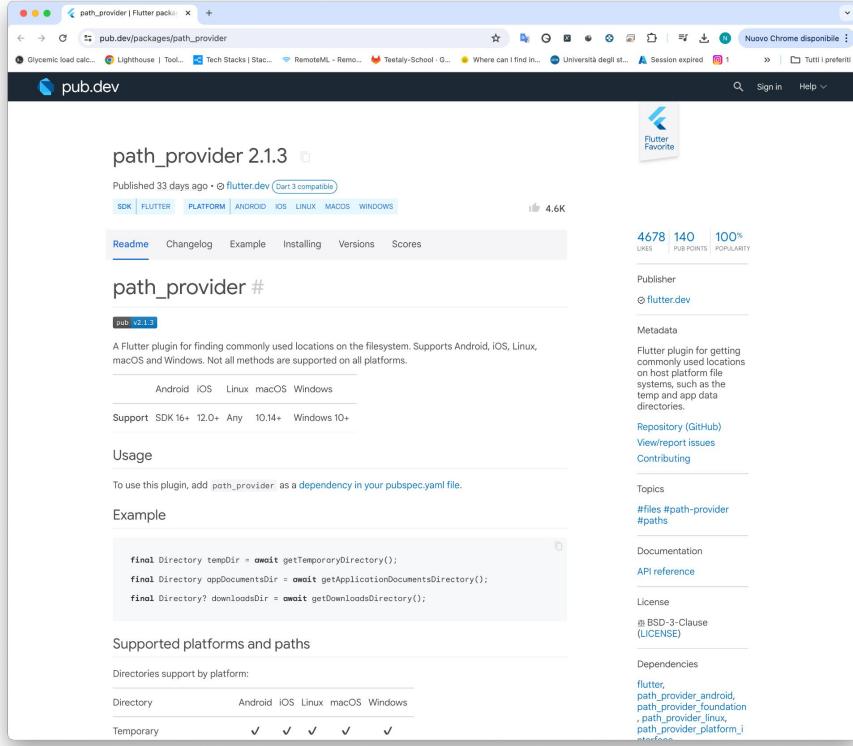
```
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text(widget.title),
        ), // AppBar
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              const Text(
                'Hai toccato il bottone questo numero di volte: ',
              ), // Text
              Text(
                '_counter',
                style: Theme.of(context).textTheme.headlineMedium,
              ), // Text
            ],
          ), // Column
        ), // Center
        floatingActionButton: FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Aumenta il contatore',
          child: const Icon(Icons.add),
        ), // FloatingActionButton
      ); // Scaffold
    }
}
```

# Persistenza su file

# Leggere e scrivere su file

In alcuni casi, è necessario leggere e scrivere file su disco. Ad esempio, potrebbe essere necessario dover persistere i dati tra i vari avvii dell'App, o scaricare dati da internet e salvarli per un uso offline successivo.

# path\_provider - [https://pub.dev/packages/path\\_provider](https://pub.dev/packages/path_provider)



The screenshot shows the pub.dev page for the `path_provider` package. At the top, it displays the package name `path_provider 2.1.3`, its compatibility with `Dart 3`, and a rating of `4.6K`. Below this, there are tabs for `Readme`, `Changelog`, `Example`, `Installing`, `Versions`, and `Scores`. The `Readme` tab is currently selected. The `Readme` content includes a brief description of the plugin, support for multiple platforms (Android, iOS, Linux, macOS, Windows), and required SDK versions. It also features sections for `Usage` and `Example`, which contains sample code for getting temporary and application documents directories. The right sidebar provides additional information such as popularity metrics (4678 likes, 140 pub points, 100% popularity), publisher details (@flutter.dev), metadata (Flutter plugin for common file operations), repository links (GitHub, issues, contributing), topics (#files #path-provider #paths), documentation (API reference), license information (BSD-3-Clause), and dependency details (flutter, path\_provider\_android, path\_provider\_foundation, path\_provider\_linux, path\_provider\_platform\_i).

# Attenzione!!!

## path\_provider 2.1.3



Published 33 days ago • flutter.dev Dart 3 compatible

SDK

FLUTTER

PLATFORM

ANDROID

IOS

LINUX

MACOS

WINDOWS

# Attenzione!!!

[path\_provider] Add support for web #45296

[Open](#) harryterkelsen opened this issue on Nov 20, 2019 · 43 comments

We should add web support for path\_provider:

harryterkelsen added c: new feature, plugin, platform-web, p: path\_provider, package, labels on Nov 20, 2019

harryterkelsen added this to Not Started in Flutter Web Plugins via automation on Nov 20, 2019

MichelFR commented on Dec 6, 2019

Since its not possible to access the filesystem on the web... how should this be done?

yibanov added this to the Near-term Goals milestone on Dec 13, 2019

mchome mentioned this issue on Feb 2, 2020

Overflow on channel when used with flutter web  
mchome/flutter\_advanced\_networkimage#144

TimWhiting mentioned this issue on Feb 15, 2020

[path\_provider] Path provider platform interface flutter/plugins#2532

TimWhiting commented on Feb 15, 2020

I can imagine that this might be useful in some contexts on the web. Like which folder to open a file-download picker to. (For user content). For cached content, I'm not sure it makes sense on the web. But it's pretty annoying to get a flood of errors in the console on web when trying to use a package like google\_fonts which uses this plugin.

Assignees  
No one assigned

Labels  
c: new feature, p: path\_provider, package, platform-web, p: path\_provider, package, team-web, triggered-web

Projects  
Flutter Web Plugins  
Not Started

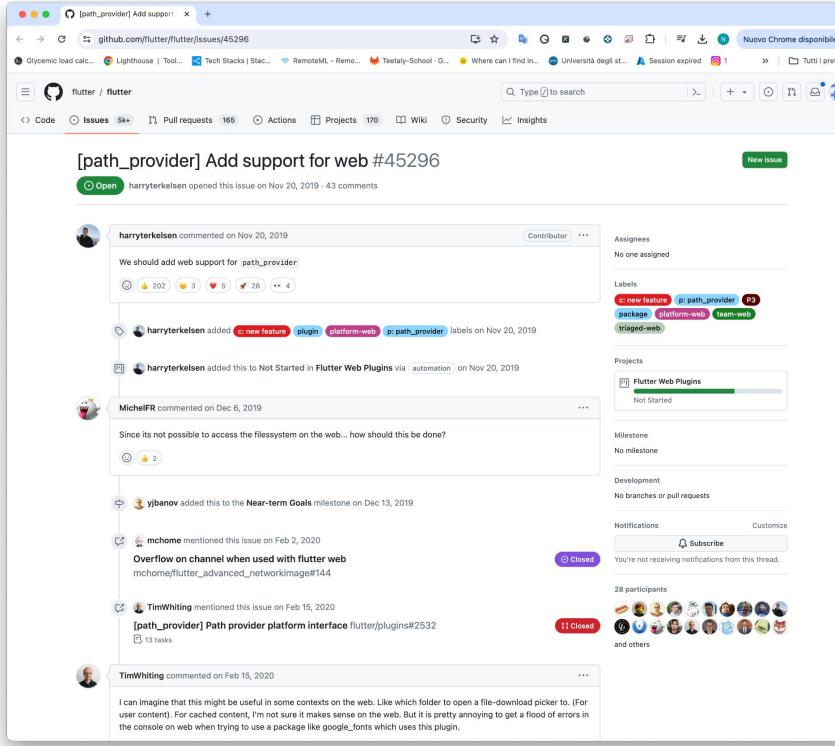
Milestone  
No milestone

Development  
No branches or pull requests

Notifications  
Subscribe  
Customize  
You're not receiving notifications from this thread.

28 participants

and others



# Trova il percorso locale corretto

Il pacchetto **path\_provider** fornisce un modo agnostico rispetto alla piattaforma per accedere a posizioni comunemente utilizzate nel file system del dispositivo.

Attualmente, il plugin supporta l'accesso a due posizioni nel file system:

- **Directory temporanea:** Una directory temporanea (cache) che il sistema può cancellare in qualsiasi momento. Su iOS, corrisponde a **NSCachesDirectory**. Su Android, questo è il valore restituito da **getCacheDir()**;
- **Directory documenti:** Una directory in cui l'App può memorizzare file a cui solo essa può accedere. Il sistema cancella la directory solo quando l'App viene eliminata. Su iOS, corrisponde a **NSDocumentDirectory**. Su Android, la directory è **AppData**.

# Trova il percorso locale corretto

dart

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

    return directory.path;
}
```

# Riferimento alla posizione del file

Una volta definito l'elemento per memorizzare il file, è sufficiente creare un riferimento alla posizione completa del file.

Per farlo è possibile utilizzare la classe **File** dalla libreria **dart:io**.

# Riferimento alla posizione del file

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

# Scrivere all'interno del file

Una volta predisposto un file con cui lavorare, è possibile usarlo per leggere e scrivere dati. Il primo step è quello di provare a scrivere alcuni dati sul file.

Nell'esempio il contatore è un intero, ma viene scritto sul file come una stringa utilizzando la sintassi '\$counter'.

# Scrivere all'interno del file

```
dart  
Future<File> writeCounter(int counter) async {  
    final file = await _localFile;  
  
    // Write the file  
    return file.writeAsString('$counter');  
}
```

# Leggere da file

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;
  }
}
```

# Progetto di esempio

```
lib > main.dart > ...
1 import 'package:flutter/material.dart';
2 import 'counter_storage.dart';
3 import 'flutter_demo.dart';
4
Run | Debug | Profile
5 void main() {
6   runApp(
7     MaterialApp(
8       title: 'Reading and Writing Files',
9       home: FlutterDemo(storage: CounterStorage()),
10      ), // MaterialApp
11    );
12 }
13 }
```

# Progetto di esempio

```
lib > counter_storage.dart > ...
1 import 'dart:io';
2 import 'package:path_provider/path_provider.dart';
3
4 class CounterStorage {
5   Future<String> get _localPath async {
6     final directory = await getApplicationDocumentsDirectory();
7
8     return directory.path;
9   }
10
11  Future<File> get _localFile async {
12    final path = await _localPath;
13    return File('$path/counter.txt');
14  }
15
16  Future<int> readCounter() async {
17    try {
18      final file = await _localFile;
19
20      // Read the file
21      final contents = await file.readAsString();
22
23      return int.parse(contents);
24    } catch (e) {
25      // If encountering an error, return 0
26      return 0;
27    }
28  }
29
30  Future<File> writeCounter(int counter) async {
31    final file = await _localFile;
32
33    // Write the file
34    return file.writeAsString('$counter');
35  }
36}★
```

# Progetto di esempio

```
lib > flutter_demo.dart > ...
1  import 'package:flutter/material.dart';
2  import 'dart:io';
3  import 'counter_storage.dart';
4
5  class FlutterDemo extends StatefulWidget {
6      const FlutterDemo({super.key, required this.storage});
7
8      final CounterStorage storage;
9
10     @override
11     State<FlutterDemo> createState() => _FlutterDemoState();
12 }
13
14 class _FlutterDemoState extends State<FlutterDemo> {
15     int _counter = 0;
16
17     @override
18     void initState() {
19         super.initState();
20         widget.storage.readCounter().then((value) {
21             setState(() {
22                 _counter = value;
23             });
24         });
25     }
}
```

# Progetto di esempio

```
Future<File> _incrementCounter() {
  setState(() {
    _counter++;
  });

  // Scrive il contatore nel file
  return widget.storage.writeCounter(_counter);
}

@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Leggendo e scrivendo su file'),
    ), // AppBar
    body: Center(
      child: Text(
        'Bottone premuto ${_counter} volte.',
      ), // Text
    ), // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Aumenta il contatore',
      child: const Icon(Icons.add),
    ), // FloatingActionButton
  ); // Scaffold
}
```

# Persistenza su database locale

# Quando usare un database locale

Se si sta sviluppando un'App che necessita di memorizzare e interrogare grandi quantità di dati sul dispositivo locale, è necessario prendere in considerazione l'utilizzo di un database invece di un file locale o di uno storage chiave-valore. In generale, i database offrono inserimenti, aggiornamenti e interrogazioni più veloci rispetto ad altre soluzioni di persistenza locale.

Le app Flutter possono fare uso dei database SQLite tramite il plugin **sqflite** disponibile su pub.dev.

# sqflite - <https://pub.dev/packages/sqflite>

The screenshot shows the official Flutter package page for sqflite on pub.dev. The page title is "sqflite 2.3.3+1". It includes a "Flutter Favorite" badge. The package was published 12 days ago by @tekartik.com (Dart 3 compatible). It has 4.6K reviews, 4692 likes, 140 pub points, and 100% popularity. The page features tabs for Readme, Changelog, Example, Installing, Versions, and Scores. The "Readme" tab is selected. The "Readme" content describes the SQLite plugin for Flutter, supporting iOS, Android, and MacOS. It lists features such as support for transactions and batches, automatic version management, helpers for insert/query/update/delete queries, and DB operations in a background thread. Other platforms supported include Linux/Windows/Dart VM using sqflite\_common\_ffI and experimental Web support using sqflite\_common\_ffI\_web. Usage examples mention notepad\_sqflite. A "Getting Started" section provides steps to add the package to a project. A terminal window shows the command "flutter pub add sqflite". A note at the bottom states that this command adds a line to the pubspec.yaml file and runs an implicit flutter pub get. The right sidebar contains links for Publisher (@tekartik.com), Metadata (Flutter plugin for SQLite), Repository (GitHub), View/report issues, Topics (#sql #database), Documentation (API reference), Funding (Consider supporting this project: github.com), License (BSD-2-Clause (LICENSE)), and Dependencies (flutter, path, sqflite\_common).

# Attenzione agli import!

Per lavorare con database SQLite, è necessario importare i pacchetti `sqflite` e `path`.

- Il pacchetto **sqflite** fornisce classi e funzioni per interagire con un database SQLite;
- Il pacchetto **path** fornisce funzioni per definire la posizione in cui memorizzare il database su disco.

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

# Definizione del data model

Per lavorare con un database è necessario in prima battuta definire un data model.

# Definizione del data model

```
dart

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

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

# Connessione al database

Prima di leggere e scrivere dati nel database, è necessario stabilire una connessione da effettuare in due passaggi:

- Bisogna definire il percorso del file del database utilizzando **getDatabasesPath()** dal pacchetto **sqflite**, combinato con la funzione **join** dal pacchetto **path**;
- Aprire il database con la funzione **openDatabase()** di **sqflite**.

# Connessione al database

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'),  
);
```

# Creazione di una tabella

Dopo aver creato la connessione è possibile creare una tabella per memorizzare le informazioni di cui abbiamo bisogno.

Nel caso specifico di questo esempio, viene creata una tabella chiamata **dogs** che definisce i dati che possono essere memorizzati. Ogni “cane” contiene un **id**, un **name** e un **age**. Pertanto, questi sono rappresentati come tre colonne nella tabella **dogs**.

- L'**id** è un int di Dart e viene memorizzato come un tipo di dato INTEGER in SQLite. È anche una buona pratica usare un **id** come chiave primaria per la tabella per migliorare i tempi di interrogazione e aggiornamento;
- Il **name** è una String di Dart e viene memorizzato come un tipo di dato TEXT in SQLite;
- L'**age** è anche un int di Dart e viene memorizzato come un tipo di dato INTEGER.

# Creazione di una tabella

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,  
);
```

# Inserimento di un elemento nel db

Una volta predisposto il database e creata una tabella adatta a memorizzare informazioni, è possibile leggere e scrivere dati.

Per cominciare a popolare la nostra tabella abbiamo bisogno di un ulteriore passaggio preliminare:

- Convertire l'oggetto del data model in formato **Map**

Questo ci permetterà di definire le chiavi e poter, in uno step successivo::

- Usare il metodo **insert()** per memorizzare l'oggetto map nella tabella del db.

# Inserimento di un elemento nel db

```
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)';  
    }  
}
```

# Inserimento di un elemento nel db

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,
    );
}
```

# Inserimento di un elemento nel db

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

await insertDog(fido);
```

dart

# Acquisire la lista degli elementi

Una volta inseriti i dati, è possibile interrogare il database per ottenere un elemento specifico o un elenco di tutti gli elementi. Questo comporta due passaggi:

- Esegui una query sulla tabella **dogs**. Questa funzione restituisce una **List<Map>**.
- Converti la **List<Map>** in una **List<Dog>**.

# Acquisire la lista degli elementi

```
// 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.
```

# Aggiornamento di un elemento nel db

Dopo aver inserito informazioni nel database, è possibile ovviamente aggiornare tali informazioni in un secondo momento. È possibile farlo usando il metodo **update()** della libreria **sqflite**.

Questo comporta due passaggi:

- Converti il l'oggetto in una mappa (**Map**).
- Usa una clausola **where** per assicurarti di aggiornare l'elemento corretto.



```
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],
    );
}
```

dart

```
// 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.
```

# Attenzione!!!!!!

Quando si eseguono aggiornamenti nel database, è importante evitare tecniche che possano esporre l'applicazione a vulnerabilità di sicurezza come gli attacchi SQL injection. Un attacco SQL injection si verifica quando un malintenzionato può inserire comandi SQL malevoli in un'istruzione SQL tramite input non controllato.

Utilizzare sempre **whereArgs** per passare argomenti a una clausola **where**. Questo aiuta a proteggere contro attacchi di SQL injection.

Non utilizzare l'interpolazione di stringhe, come where: "**id = \${dog.id}**"!

Vantaggi di usare **whereArgs**:

- **Sicurezza:** Previene gli attacchi SQL injection.
- **Manutenibilità:** Rende il codice più chiaro e facile da leggere.
- **Affidabilità:** Gestisce automaticamente l'escaping dei valori, prevenendo errori sintattici.

# Cancellare un elemento dal db

È possibile anche rimuovere degli elementi dal database. Per eliminare i dati, + sufficiente usare il metodo **delete()** della libreria **sqflite**.

In questa sezione, crea una funzione che prende un **id** ed elimina l'elemento con l'**id** corrispondente dal database. Per fare questo, bisogna fornire una clausola **where** per limitare i record da eliminare.

# Cancellare un elemento dal db

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],
    );
}
```

# Progetto completo

Il progetto contenente la sequenza dei comandi mostrati è disponibile nel file sqlite.zip e può essere utile nel caso vogliate integrare questa tipologia di persistenza all'interno delle vostre App.