ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

502071

Cross-Platform Mobile Application Development

Persistence

1

Outline

- Shared Preferences
- Local File
- Local Database
- Firebase Cloud Firestore

Introduction

- Persistence is the ability of an app to store data even when the app is closed or restarted. Without persistence, apps would lose data every time they are closed or restarted, making it difficult for users to resume their previous activities.
- Flutter provides several ways to implement persistence in apps. Some of the most common techniques include using Shared preferences, SQLite databases, or Firebase Cloud Firestore.

Overview of Persistence

- **Shared Preferences:**
  - Shared Preferences is a simple key-value store that allows you to store small amounts of data, such as user preferences, settings, or application state. It's easy to use and doesn't require any external libraries. However, it's not suitable for storing large amounts of data, and it doesn't provide any data validation or type checking.
- **SQLite Databases:**
  - SQLite is a lightweight relational database that is included in the Flutter framework. It allows you to store structured data, such as user profiles, inventory, or transactions. SQLite provides powerful querying capabilities and supports transactions, which ensures data consistency. However, it requires some knowledge of SQL and can be complex to set up.

Overview of Persistence

- **Firebase Cloud Firestore:**
  - Firebase Cloud Firestore is a NoSQL cloud-based database solution provided by Google. It allows you to store and sync data in real-time across multiple clients and platforms. It's highly scalable and provides offline support, which means that your app can still work even when there's no internet connection. However, it requires an internet connection and can be expensive to use for large-scale applications.
- **File API:**
  - The File API allows Flutter apps to read and write files on the device's storage. This can be useful for storing larger amounts of data, such as images, videos, or audio recordings. The File API provides a set of classes and methods for managing files and directories, and it works on both iOS and Android devices. However, it requires permissions to access the device's storage and can be complex to use.

Overview of Persistence

- **Storing Data using Remote Server via REST API:**
  - Flutter apps can also store data remotely by using a REST API to communicate with a remote server. REST APIs allow apps to send and receive data in a standardized format, such as JSON or XML, over HTTP or HTTPS. This can be useful for storing and retrieving data from a remote database, such as user profiles, inventory, or transactions. However, it requires knowledge of networking and server-side programming, and it's important to ensure the security and reliability of the API.

Shared Preferences

Introduction to Shared Preferences

- Shared Preferences is a simple key-value store that allows you to store small amounts of data in a persistent manner.
- Shared Preferences is used for storing small amounts of data, such as user preferences, settings, or application state.

Explanation of Shared Preferences

- Shared Preferences is based on key-value pairs, where each pair consists of a unique key and a corresponding value.
- The key is a String, and the value can be a Boolean, int, double, String, or a Set of Strings.
- The values are stored in a file on the device's disk and can be accessed by any part of the app.
- Shared Preferences uses a simple API for storing and retrieving data, which makes it easy to use.

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="my_string">hello world!</string>
    <int name="my_integer" value="124243" />
    <boolean name="my_boolean" value="true" />
</map>
```

Shared Preferences

- On **Android**, the shared preferences data is stored in an XML file in the app's data directory, which is typically located at */data/data/<package-name>/shared_prefs/*.
- On **iOS**, the shared preferences data is stored in the UserDefaults system, which is similar to the preferences system used on Android. The data is stored in a plist file in the app's sandbox directory, which is typically located at */var/mobile/Containers/Data/Application/<app-id>/Library/Preferences/*.
- On **Windows**, the shared preferences data is stored in the registry, in the *HKEY_CURRENT_USER\Software\<company-name>\<app-name>\ key*.
- On **web**, the shared preferences data is stored using the browser's localStorage API, which stores data in the browser's local storage.

Usecase of Shared Preferences data

- User preferences, such as language settings, font size, or color theme.
- Application state, such as whether the app is in dark mode, or whether a tutorial has been completed.
- User authentication data, such as login credentials or access tokens.

How to use Shared Preferences

- First, add the Shared Preferences package to your project dependencies in pubspec.yaml:

```
$ flutter pub add shared_preferences
```

- The SharedPreferences object is a singleton, meaning that there is only one instance of it throughout the entire app.
- You can get access to the SharedPreferences instance using the getInstance() method, which returns a Future that resolves to the SharedPreferences instance.

How to use Shared Preferences

- Once you have the SharedPreferences instance, you can use it to read and write data using the key-value store.

```
class _MyHomePageState extends State<MyHomePage> {
    late SharedPreferences _prefs;
    Future<void> _loadPrefs() async {
        _prefs = await SharedPreferences.getInstance();
        setState(() {});
    }
    void initState() {
        super.initState();
        _loadPrefs();
    }
}
```

How to use Shared Preferences

- let's say we want to save the user's name and retrieve it later using Shared Preferences. We can use the SharedPreferences class to save and retrieve the data.

```
void _saveName(String name) {
    _prefs.setString('name', name);
}

String? _getName() {
    return _prefs.getString('name');
}
```

How to use Shared Preferences

- To store a value of a particular data type in SharedPreferences, you can use the corresponding setter method:
  - setString(String key, String value) to store a string value.
  - setBool(String key, bool value) to store a boolean value.
  - setInt(String key, int value) to store an integer value.
  - setDouble(String key, double value) to store a double value.
  - setStringList(String key, List<String> value) to store a list of string value.

How to use Shared Preferences

- If you want to store and retrieve other types of data, such as lists or maps, you can convert them to a string using a serialization method such as json.encode() or dart:convert and then store the resulting string as a String value in SharedPreferences.
- When you retrieve the value, you can convert it back to the original data type using the corresponding deserialization method such as json.decode() or dart:convert.

```
var users = [
    {'id': 1, 'name': 'Khoa', 'age': 20},
    {'id': 2, 'name': 'Toan', 'age': 21},
];
_prefs.setString('users', json.encode(users));
```

How to use Shared Preferences

- The setString() method (and other .setXXX() methods) in SharedPreferences is an asynchronous function that returns a Future<void> indicating that the operation has completed.
- To wait for the operation to complete and ensure that the data has been successfully saved, you can use the await keyword when calling the setString() method:

```
void _saveName(String name) async {
    try {
        await _prefs.setString('name', name);
        print('the data has been successfully saved');
    } catch (e) {
        print('Error $e');
    }
}
```

Best practices for using Shared Preferences

- Avoid storing large amounts of data, as Shared Preferences is not designed for that.
- Use unique and descriptive keys to avoid naming conflicts and make your code more readable.
- Use type-safe methods, such as setBool and getBool, to avoid type errors.
- Use try-catch blocks when accessing Shared Preferences, as it can throw exceptions in case of errors.

Secured Shared Preferences

Secured Shared Preference

- SharedPreferences in Flutter is not considered to be a secure way of storing sensitive data, as it stores the data in plain text in the device's file system. This means that if someone gains access to the device or the device's file system, they can easily read and modify the data stored in SharedPreferences.
- To store sensitive data securely, you can use the flutter_secure_storage package, which provides a secure key-value storage that uses the device's secure storage APIs. This package encrypts the data stored in the device and requires the use of a key to access the data.

```
flutter pub add flutter_secure_storage
```

Secured Shared Preference

- You can then create an instance of the FlutterSecureStorage class and use its methods to store and retrieve data securely. For example, to store a sensitive string value with the key "password", you can use:

```
final storage = FlutterSecureStorage();
await storage.write(key: "password", value: "mypassword");
```

- To retrieve the value stored with the key "password", you can use:

```
String password = await storage.read(key: "password");
```

Secured Shared Preference

- On Android, the flutter_secure_storage package stores data in the Android Keystore, which is a secure storage area on the device that provides hardware-backed encryption. The Android Keystore is typically located at /data/misc/keystore/ on the device.
- On iOS, the flutter_secure_storage package stores data in the Keychain, which is a secure storage area on the device that provides hardware-backed encryption. The Keychain is typically located at /Library/Keychains/ on the device.
- On web, the flutter_secure_storage package uses the browser's built-in localStorage API to store encrypted data in the browser's local storage. The data is encrypted using the Web Crypto API, which provides hardware-backed encryption capabilities in modern web browsers.
- On Windows, the flutter_secure_storage package uses the Windows Data Protection API (DPAPI) to store data securely. The DPAPI is a built-in encryption API provided by the Windows operating system that allows data to be encrypted using a user-specific key.

Secured Shared Preference

- it's important to note that:
  - There is a performance overhead associated with using secured shared preferences, since encryption and decryption take additional processing time. Therefore, it's generally recommended to use secured shared preferences only when necessary, and to use regular shared preferences for non-sensitive data.
  - Even with flutter_secure_storage, you should still take additional steps to secure sensitive data, such as encrypting the data before storing it or using more advanced security techniques such as server-side encryption.

Local File

Local File Overview

- One of these features is the File API, which allows developers to work with files and directories on the device's storage system.
- With the File API, developers can read, write, and manipulate files and directories, which is a critical feature for any app that needs to store or access data on the user's device.

Local File Overview

- Here are some specific use cases where using the Flutter File API might be a better choice than other forms of persistence:
  - Small data sets: Databases can be overkill for small data sets, and using the File API can be faster and more efficient.
  - User-generated content: The File API makes it easy to create and manage user-generated content, such as notes, images, or audio recordings, and it can be more intuitive for users to understand.
  - Offline data caching: The File API can be used to store and retrieve data from the device's local storage, allowing your app to continue to function even when a network connection is unavailable.
  - Data backups: The File API allows you to create a copy of the app's data on the device's local storage, which can be easily uploaded to cloud storage or transferred to another device.

Local File

- **Android**:
  - On Android, the File API uses the underlying file system of the device, which is typically based on the Linux file system.
  - The File API provides access to the device's internal storage, external storage, and shared storage.
  - To access the external storage, the app must request the 'READ_EXTERNAL_STORAGE' and/or 'WRITE_EXTERNAL_STORAGE' permissions.
- **iOS**:
  - On iOS, the File API uses the underlying file system of the device, which is based on the HFS+ file system or the newer APFS file system.
  - The File API provides access to the app's sandboxed directory, which is the only directory the app can access by default.
  - To access other directories, the app must use the FileProvider API or ask the user for permission.

Local File

- **macOS**:
  - On macOS, the File API uses the underlying file system of the device, which is typically based on the HFS+ file system or the newer APFS file system.
  - The File API provides access to the file system, including the user's home directory and other directories.
  - To access other directories, the app must use the App Sandbox feature or ask the user for permission.
- **Windows**:
  - On Windows, the File API uses the underlying file system of the device, which is typically based on the NTFS file system or the newer ReFS file system.
  - The File API provides access to the file system, including the user's home directory and other directories.
  - To access other directories, the app must use the FilePicker API or ask the user for permission.

Local File

- Web:
  - The dart:io library, which provides classes for working with files and directories, is not available in Flutter web because it's designed to work with the underlying operating system of the device running the app.
  - Flutter web apps run in a browser sandbox and don't have direct access to the file system of the user's device.

Local File - Find the correct local path

- 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.

Local File - Find the correct local path

- You can find the path to the documents directory as follows:

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

- 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.

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

Local File - 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.

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

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

Local File - Read data from the file

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

```
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;
    }
}
```

Local File - Recap

- **Platform-specific behavior**: File system paths are represented differently on Android and iOS, and some file operations may not be available on all platforms. You should test your file operations on all platforms your app targets.
- **Permissions**: On mobile and desktop platforms, your app may need to request permission from the user to access certain files or directories. You should handle permission requests gracefully.
- **File size limitations**: You should handle large files or streams of data using async/await to avoid blocking the UI thread.
- **File concurrency**: If your app needs to access files from multiple threads or isolates, you should use a file locking mechanism to prevent multiple processes from modifying the same file simultaneously.
- **Error handling**: When working with files, errors can occur due to various reasons such as file not found, permission denied, and disk full. You should handle these errors gracefully and inform the user about the issue.
- **Storage limitations**: Mobile devices typically have limited storage, so you should be mindful of the amount of space your app uses for file storage. You should also provide options for the user to delete or manage files within your app.

SQLite Database

SQLite Database

- Flutter provides an easy-to-use SQLite database plugin that allows developers to store and retrieve data locally on a device. This makes it easy to create apps that can work offline and are not dependent on network connectivity.
- 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.

SQLite Database

- Flutter's sqflite package provides a simple way to work with SQLite databases in your Flutter app.

  Here are the supported platforms for the sqflite package:

  - Android
  - iOS
  - macOS
  - Windows
  - Linux

- Note that sqflite is a plugin, so it may have some platform-specific behaviors or limitations.

SQLite Database

- Flutter apps can make use of the SQLite databases via the sqflite plugin available on pub.dev. In this lesson, we'll learn how to use the SQLite plugin in Flutter to create a simple database, add data to it, and retrieve data from it.
- To use SQLite in your Flutter app, you need to add the sqlite plugin to your project. To do this, add the following dependency to your pubspec.yaml file:

```
dependencies:
flutter:
    sdk: flutter
sqflite: ^2.0.0+3
```

- Once you've added the dependency, run the flutter pub get command in your terminal to install it.

SQLite Database

- Before creating the table to store information, take a few moments to define the data that needs to be stored. For this example, define a Book class that contains three pieces of data: A unique id, the title, and the author of each book.

```
class Book {
    final int? id;
    final String title;
    final String author;

    Book({this.id, required this.title, required this.author});

    Map<String, dynamic> toMap() {
        return {'id': id, 'title': title, 'author': author};
    }
}
```

Open the database

- Before reading and writing data to the database, open a connection to the database. This involves two steps:
  - Define the path to the database file using getDatabasesPath() from the sqflite package, combined with the join function from the path package.
  - Open the database with the openDatabase() function from sqflite.

```
// 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(
    join(await getDatabasesPath(), ‘book.db'),
);
```

WidgetsFlutterBinding.ensureInitialized()

- In Flutter, WidgetsFlutterBinding is the glue between the Flutter framework and the underlying platform-specific code. It is responsible for initializing various Flutter services and frameworks, such as the rendering engine, the widget tree, and the event loop.
- When you call WidgetsFlutterBinding.ensureInitialized(), you ensure that the WidgetsFlutterBinding is fully initialized before proceeding with any further code execution.
- The sqflite depends on other Flutter services such as the platform channel for performing database operations, which are initialized by WidgetsFlutterBinding.
- If you don't ensure that WidgetsFlutterBinding is fully initialized before using sqflite, you may encounter errors such as platform channel not found errors or other similar issues.

Create the database

- Next, create a table to store information about various Books. For this example, create a table called books that defines the data that can be stored. Each Book contains an id, title, and author.

```
var path = join(await getDatabasesPath(), book.db');
final database = openDatabase(path, version: 1, onCreate: _createDB);

Future<void> _createDB(Database db, int version) async {
    final idType = 'INTEGER PRIMARY KEY AUTOINCREMENT';
    final textType = 'TEXT NOT NULL';
    await db.execute('''
        CREATE TABLE books (
        id $idType,
        title $textType,
        author $textType
        )''');
}
```

```dart
class BookDatabase {
    static Database? _database;
    static final BookDatabase instance = BookDatabase._init();
    BookDatabase._init();

    Future<Database> get database async {
        if (_database != null) return _database!;
        _database = await _initDB('book.db');
        return _database!;
    }

    Future<Database> _initDB(String filePath) async {
        final dbPath = await getDatabasesPath();
        final path = join(dbPath, filePath);

        return await openDatabase(path, version: 1, onCreate: _createDB);
    }
}
```

Adding Data to the Database

- To add data to the database, we can use the insert method provided by the sqflite plugin. In this example, we'll add a new book to the books table.

```
class BookDao {
    Future<int> create(Book book) async {
        final db = await BookDatabase.instance.database;
        return db.insert('books', book.toMap());
    }
}
```

- In the code above, we create a Book class to represent a book object, and a BookDao class with a create method that inserts a new book into the books table. The toMap method in the Book class converts a Book object to a map that can be used with the insert method.

Retrieving Data from the Database

- To retrieve data from the database, we can use the query method provided by the sqflite plugin. In this example, we'll retrieve a list of all books in the books table.

```
class BookDao {
    ...
    Future<List<Book>> getAll() async {
        final db = await BookDatabase.instance.database;
        final books = await db.query('books');
        return books.map((json) => Book.fromMap(json)).toList();
    }
}
```

- In the code above, we create a getAll method in the BookDao class that retrieves all rows from the books table and returns them as a list of Book objects.

Retrieving Data from the Database

- In the code above, we added the fromMap method to the Book class that takes a Map<String, dynamic> as an argument and returns a Book object. The method extracts the id, title, and author values from the map and uses them to create a new Book object.

```
class Book {
    ...
    Map<String, dynamic> toMap() {...}
    static Book fromMap(Map<String, dynamic> map) {
        return Book(
            id: map['id'] as int,
            title: map['title'] as String,
            author: map['author'] as String,
        );
    }
}
```

Using the Database

- Now that we have created our database, added data to it, and retrieved data from it, let's see how we can use it in our app.

```
Scaffold(
    appBar: AppBar(title: Text('My Books')),
    body: FutureBuilder<List<Book>>(
        future: BookDao().getAll(),
        builder: (context, snapshot) {
            if (!snapshot.hasData) {
                return Center(child: CircularProgressIndicator());
            }
            final books = snapshot.data!;
            return ListView.builder(...);
        },
    ),
)
```

Firebase Firestore

Firestore

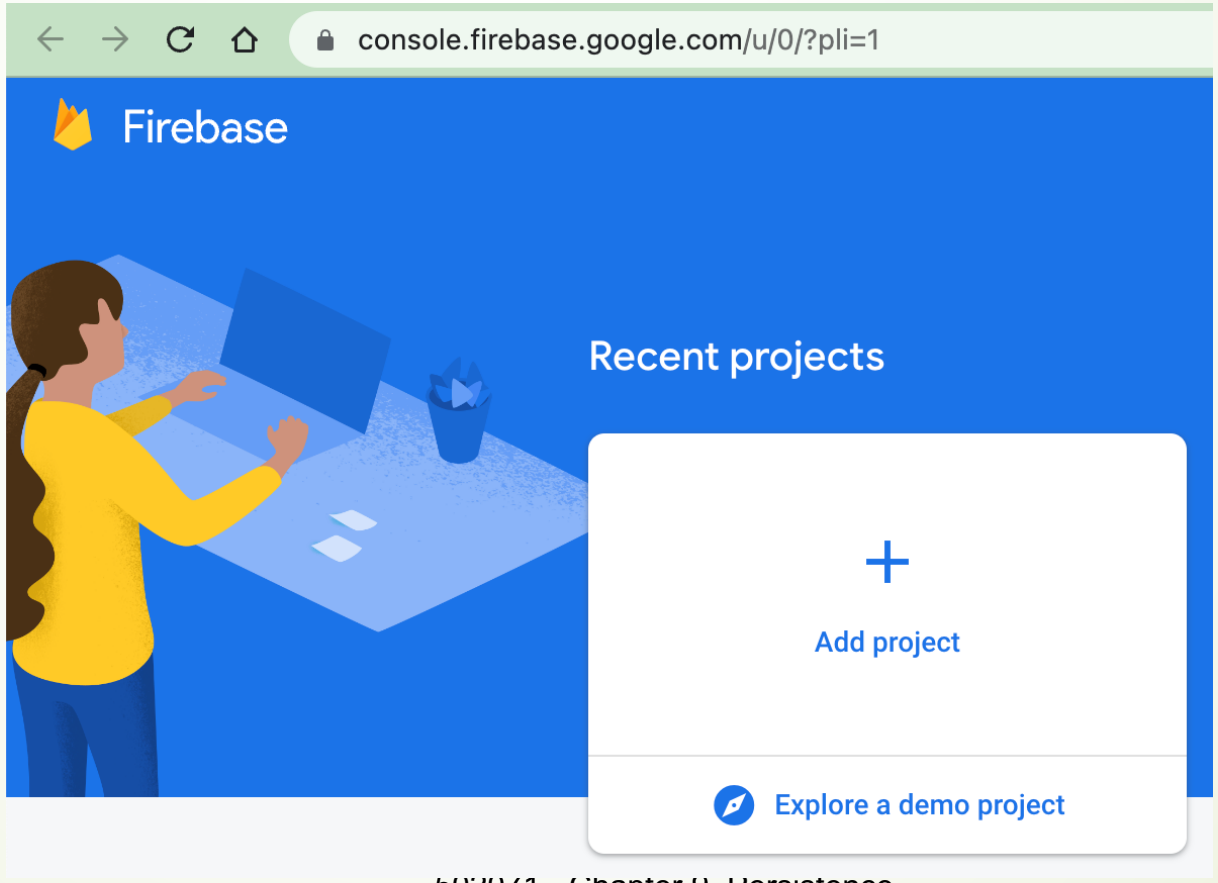
- Firebase Firestore is a cloud-hosted NoSQL document database that offers real-time data synchronization and offline data support for web, mobile, and server development.
- Firestore provides a flexible and scalable database solution that can handle complex queries, large datasets, and complex operations with ease. It also offers seamless integration with other Firebase services, such as authentication, hosting, and cloud functions.
- Firestore's real-time data synchronization allows you to build reactive user interfaces that automatically update when data changes, making it a great choice for real-time applications like chat apps, collaboration tools, and multiplayer games.

Firestore - Key capabilities

- **Flexibility**: The Cloud Firestore data model supports flexible, hierarchical data structures. Store your data in documents, organized into collections. Documents can contain complex nested objects in addition to subcollections.
- **Expressive querying**: You can use queries to retrieve individual, specific documents or to retrieve all the documents in a collection that match your query parameters. Your queries can include multiple, chained filters and combine filtering and sorting. They're also indexed by default, so query performance is proportional to the size of your result set, not your data set.
- **Realtime updates**: Like Realtime Database, Cloud Firestore uses data synchronization to update data on any connected device. However, it's also designed to make simple, one-time fetch queries efficiently.
- **Offline support**: Cloud Firestore caches data that your app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device comes back online, Cloud Firestore synchronizes any local changes back to Cloud Firestore.
- **Designed to scale:** Cloud Firestore brings you the best of Google Cloud's powerful infrastructure: automatic multi-region data replication, strong consistency guarantees, atomic batch operations, and real transaction support. We've designed Cloud Firestore to handle the toughest database workloads from the world's biggest apps.

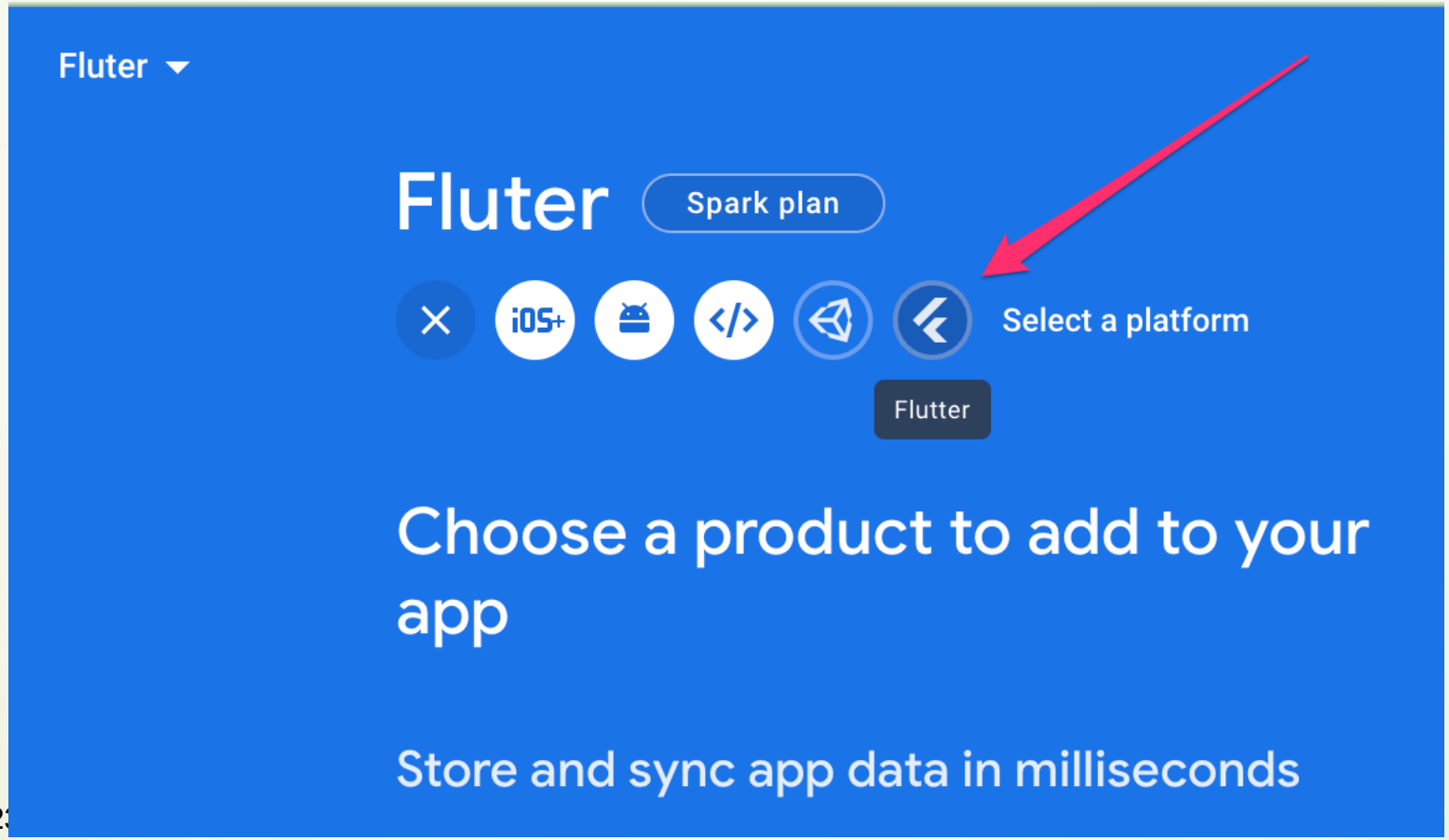
Firestore – Get started

- Create a Firebase project: In the Firebase console, click Add project, then follow the on-screen instructions to create a Firebase project or to add Firebase services to an existing GCP project.



Firestore – Get started

- Create a Firebase project: In the Firebase console, click Add project, then follow the on-screen instructions to create a Firebase project or to add Firebase services to an existing GCP project.



Prepare your workspace

Install and run the FlutterFire CLI

3 **Initialize Firebase and add plugins**

To initialize Firebase, call `Firebase.initializeApp` from the `firebase_core` package with the configuration from your new `firebase_options.dart` file:

```
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';

// ...

await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
);
```

Then, add and begin using the Flutter plugins for the Firebase products you'd like to use.

Note: If you're using Analytics or Performance Monitoring, you may need to follow a few additional setup steps.

Firestore – Get started

- You'll need to first create a Firebase project and enable Firestore in the Firebase console. Once you've set up your project, you can add the Firestore dependency to your Flutter app and start using it to store and retrieve data.

```
dependencies:
firebase_core: ^1.7.0
cloud_firestore: ^3.1.0
```

- Initialize Firebase in your Flutter app by adding the following code to your main.dart file:

```
import 'package:firebase_core/firebase_core.dart';
void main() async {
    WidgetsFlutterBinding.ensureInitialized();
    await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform,);
    await Firebase.initializeApp();
    runApp(MyApp());
}
```

Firestore – Get started

- Once complete, rebuild your Flutter application:

```
flutter run
```

- Create a Firestore instance and reference your Firestore collection in your Flutter app:

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

final FirebaseFirestore db = FirebaseFirestore.instance;
```

Firestore

- Cloud Firestore stores data in Documents, which are stored in Collections. Cloud Firestore creates collections and documents implicitly the first time you add data to the document. You do not need to explicitly create collections or documents.
- Create a new collection and a document using the following example code.

```
// Create a new user with a first and last name
final user = <String, dynamic>{
    "first": "Ada",
    "last": "Lovelace",
    "born": 1815
};

// Add a new document with a generated ID
var doc = await db.collection("users").add(user);
print('DocumentSnapshot added with ID: ${doc.id}'));
```

Firestore

- You can retrieve data from Firestore using the get() method:

```
DocumentSnapshot snapshot = await usersCollection.doc('user_id').get();
if (snapshot.exists) {
    Map<String, dynamic> data = snapshot.data() as Map<String, dynamic>;
    print(data);
}
```

- Update data in Firestore using the update() method:

```
Map<String, dynamic> data = {
    'name': 'Jane Doe',
    'email': 'janedoe@gmail.com',
    'age': 35,
};
await usersCollection.doc('user_id').update(data);
```

Firestore

- Delete data from Firestore using the delete() method:

```
await usersCollection.doc('user_id').delete();
```

Firestore - Secure your data

- If you're using the Web, Android, or Apple platforms SDK, use Firebase Authentication and Cloud Firestore Security Rules to secure your data in Cloud Firestore.
- **Auth requied**

```
// Allow read/write access on all documents to any user signed in to the application
service cloud.firestore {
    match /databases/{database}/documents {
        match /{document=**} {
            allow read, write: if request.auth != null;
        }
    }
}
```

Firestore - Secure your data

- If you're using the Web, Android, or Apple platforms SDK, use Firebase Authentication and Cloud Firestore Security Rules to secure your data in Cloud Firestore.

- **Locked mode**

```
// Deny read/write access to all users under any conditions
service cloud.firestore {
    match /databases/{database}/documents {
        match /{document=**} {
            allow read, write: if false;
        }
    }
}
```

Firestore - Secure your data

- If you're using the Web, Android, or Apple platforms SDK, use Firebase Authentication and Cloud Firestore Security Rules to secure your data in Cloud Firestore.

- **Test mode**

```
// Allow read/write access to all users under any conditions
// Warning: **NEVER** use this rule set in production; it allows
// anyone to overwrite your entire database.
service cloud.firestore {
    match /databases/{database}/documents {
        match /{document=**} {
            allow read, write: if true;
        }
    }
}
```

Firestore - Secure your data

- An example of a Test mode in Firebase Console