Chapter 7:
Data storage

Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Data persistence

- **Definition:** refers to the ability of an application to **store** and **retrieve** data even after the **application is closed** or the **device is restarted**.

- It ensures that **valuable** data (e.g. user information, settings data, application state, game score, music playlist, data history, …) are **preserved**.

A mobile app without data persistence is a lifeless app

Data persistence

- **Use case :** basic counter app.

Data persistence

- Data can be stored in : a database, a file system, or any other type of data store.

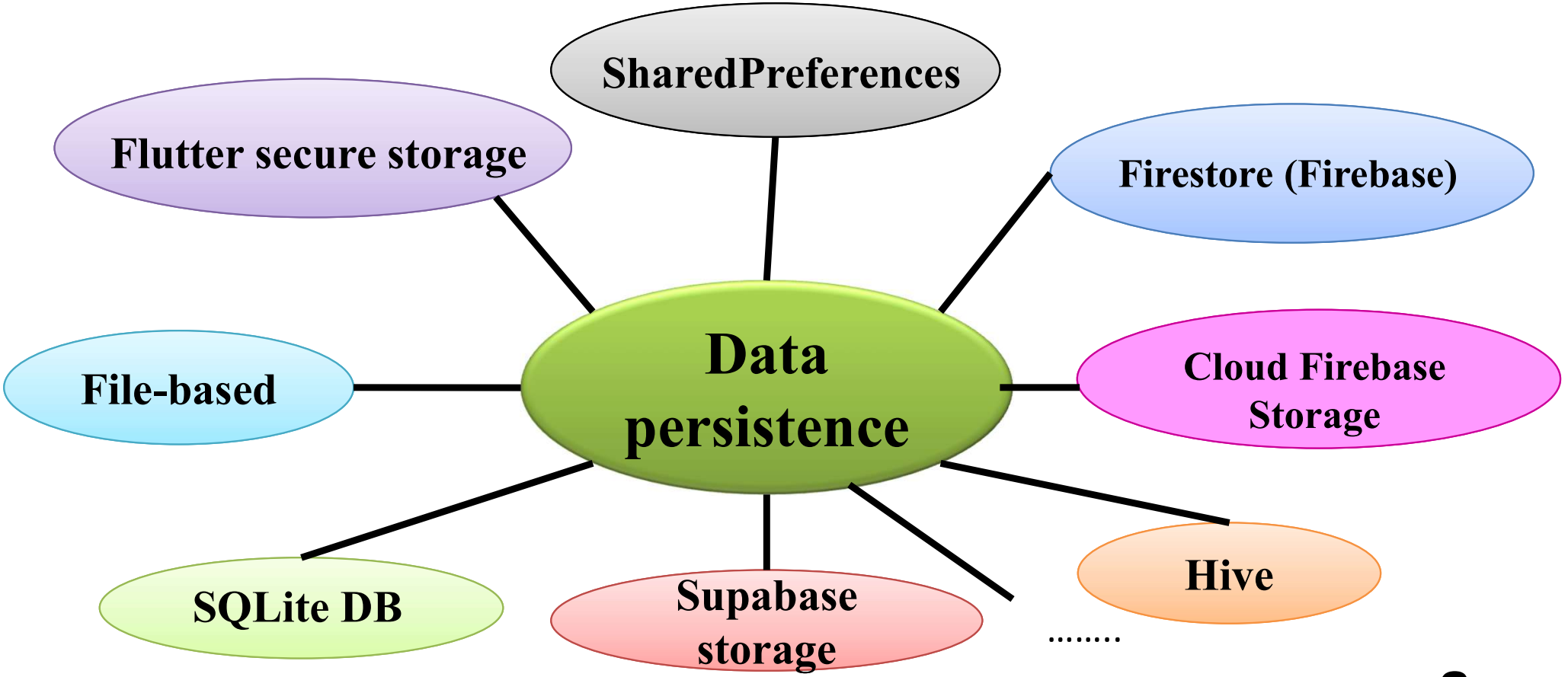
- Data can be stored in a variety of formats: plain text, JSON, XML, binary…

Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Data storage methods

- In Flutter there are various data storage solutions, and the choice depends on the type of the app being developed.



Data storage methods

- **Classification criteria :**

  - Data location:
    - ✓ **Offline** locally on the device.
    - ✓ **Online** remotely on the server.
  - Data structure:
    - ✓ Structured data (**relational database (DB)**).
    - ✓ Unstructured data (**key-value, document DB**).
  - Data size:
    - ✓ **Small** set of data.
    - ✓ **Large** set of data.
  - Data security.

Data storage methods

- **Classification :**

Classification criteria	Local storage (offline)	Cloud storage (remote access)
Data location	• SharedPreferences. • FlutterSecureStorage. • Hive. • SQLite (sqflite, Drift). • Isar. • Path provider. (*) • ObjectBox.	• Supabase (PostgresSQL). • Firebase storage. • Firebase Firestore. • Supabase storage.

Classification criteria	Secure	Not secure by default
Data security	• FlutterSecureStorage. • Supabase (PostgresSQL). • Firebase storage. • Firebase Firestore. • Supabase storage.	• SharedPreferences. • Path provider. • Hive (can be encrypted). • SQLite (sqflite, Drift) (can be encrypted). • Isar (can be encrypted). • ObjectBox (can be encrypted).

(*) Path provider method is generally called File-based storage method.

Data storage methods

- **Classification :**

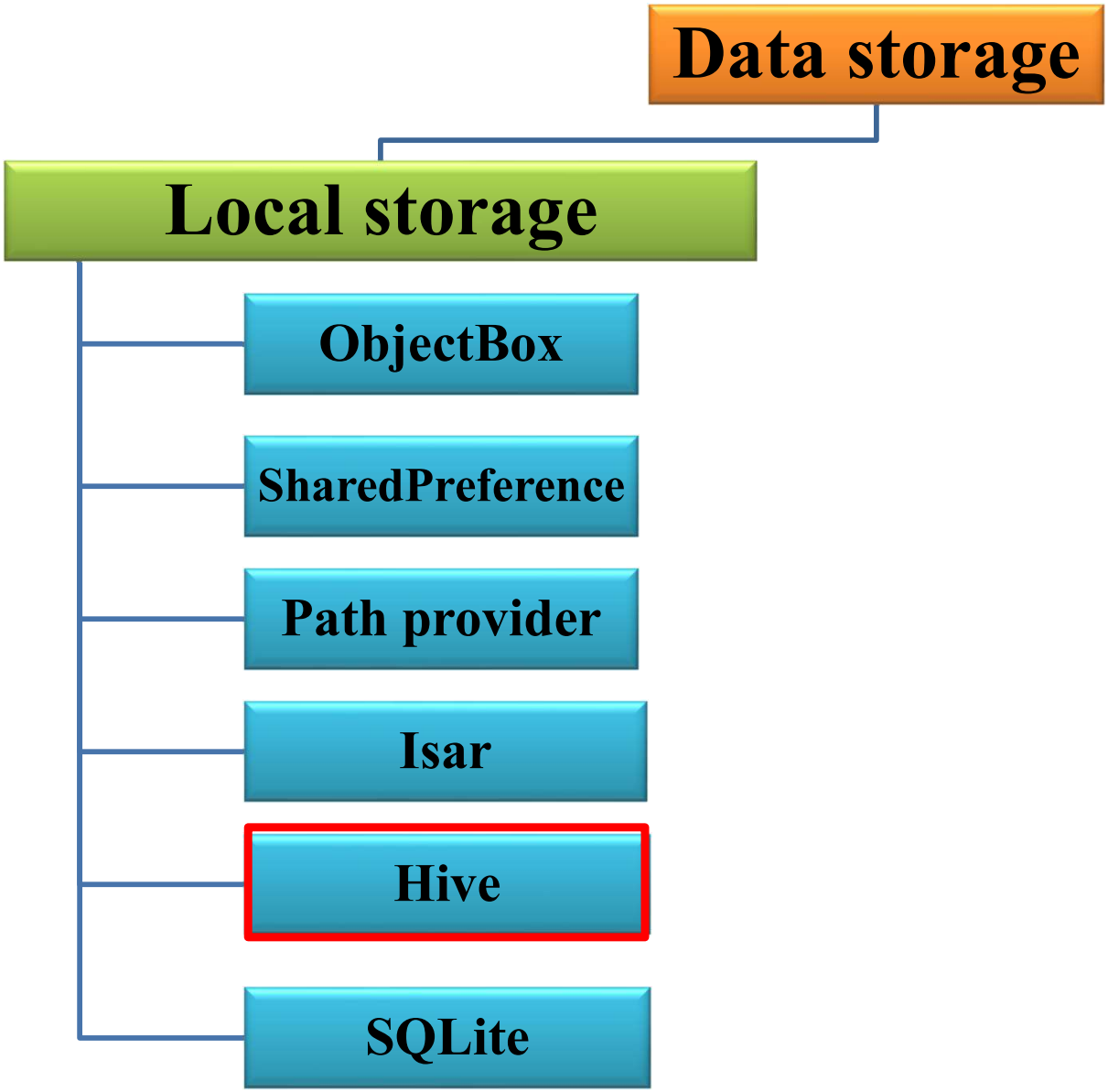
Classification criteria	Key-value (simple & fast)	Structured databases (Relational SQL-Based)	Unstructured databases (NoSQL)	File storage (images, PDFs, Videos, …)
Type of stored data (Data structure)	• SharedPreferences. • FlutterSecureStorage. • Hive.	• SQLite (sqflite, Drift). • Supabase (PostgresSQL).	• Firebase Firestore. • Hive. (*) • Isar. • ObjectBox.	• Path provider. • Firebase storage. • Supabase storage.

Classification criteria	Small volume (KB – a few MB)	Medium volume (MB – hundreds of MB)	Large volume (GB and above)
Data size	• SharedPreferences. • FlutterSecureStorage.	• Hive. • SQLite (sqflite, Drift). • Isar. • ObjectBox.	• Supabase (PostgresSQL). • Firebase storage. • Firebase Firestore. • Supabase storage. • Path provider (for local file storage)

(*) Hive is mainly a key-value store, but it has some NoSQL-like features.

Data storage methods

- Classification :



Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Local storage with Hive

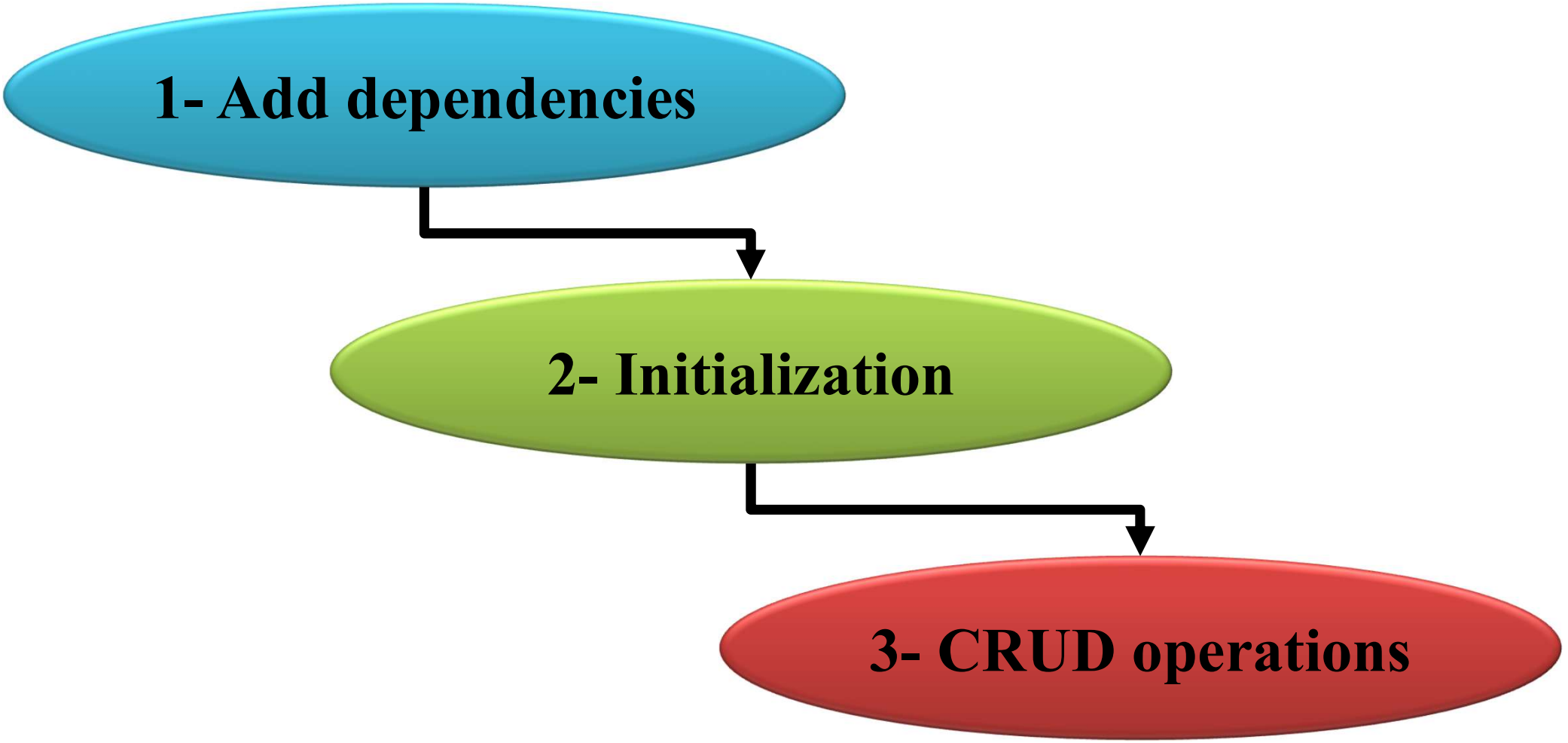
- **Hive** is lightweight and blazing **fast key-value** database.

- It is simple to use and has no native dependencies.

- A key-value store is called **Box**.

Local storage with Hive

- **Implementation steps :**



Local storage with Hive

- **Implementation: add dependencies.**

1. From the terminal run the commands:

```
$ flutter pub add hive
$ flutter pub add hive_flutter
$ flutter pub add hive_generator
$ flutter pub add build_runner
```

Local storage with Hive

- **Implementation: add dependencies.**

2. Or, add dependencies directly to "*pubspec.yaml*"

```
dependencies:
        hive: ^2.2.3
        hive_flutter: ^1.1.0
```

```
dev_dependencies:
        hive_generator: ^2.0.1
        build_runner: ^2.4.8
```

3. Import the packages from the project:

```
import 'package:hive/hive.dart';
import 'package:hive_flutter/hive_flutter.dart';
import 'package:hive_generator/hive_generator.dart';
import 'package:build_runner/build_runner.dart';
```

Local storage with Hive

- **Implementation: initialization.**

```
await Hive.initFlutter ( ); // in the main
```

- **Implementation: Create a box:**

```
var mybox = await Hive.openBox ('boxName'); // in the main

OR

var mybox = await Hive.openBox <DataModel> ('boxName');
```

- **Implementation: Get an opened box** (i.e. get the singleton instance of an already opened box).

```
var mybox =  Hive.box ('boxName');
```

Local storage with Hive

- **Implementation: write data to a box (CRUD).**

```
mybox.put(' name', 'TOTO'); // primitive string type
mybox.put(' birthday',DateTime.parse('2024-01-31'));//DateTime
mybox.put('hobbies', ['sport', 'music', 'traveling']); // List
mybox.putAll({'key1': 'value1', 'key2': 'value2'}); // Map
```

- **Implementation: read data from a box (CRUD).**

```
String name = mybox.get('name');
DateTime birthday = mybox.get('birthday');
```

Local storage with Hive

- **Implementation: read data from a box (CRUD).**

  ☺ **Returns null => use default value.**

```
double length= mybox.get(' lenght', defaultValue:15.5);
```

- **Update data from a box (CRUD).**

```
mybox.put(' name', 'TITI');
```

- **Delete data from a box (CRUD).**

```
mybox.delete('key');
```

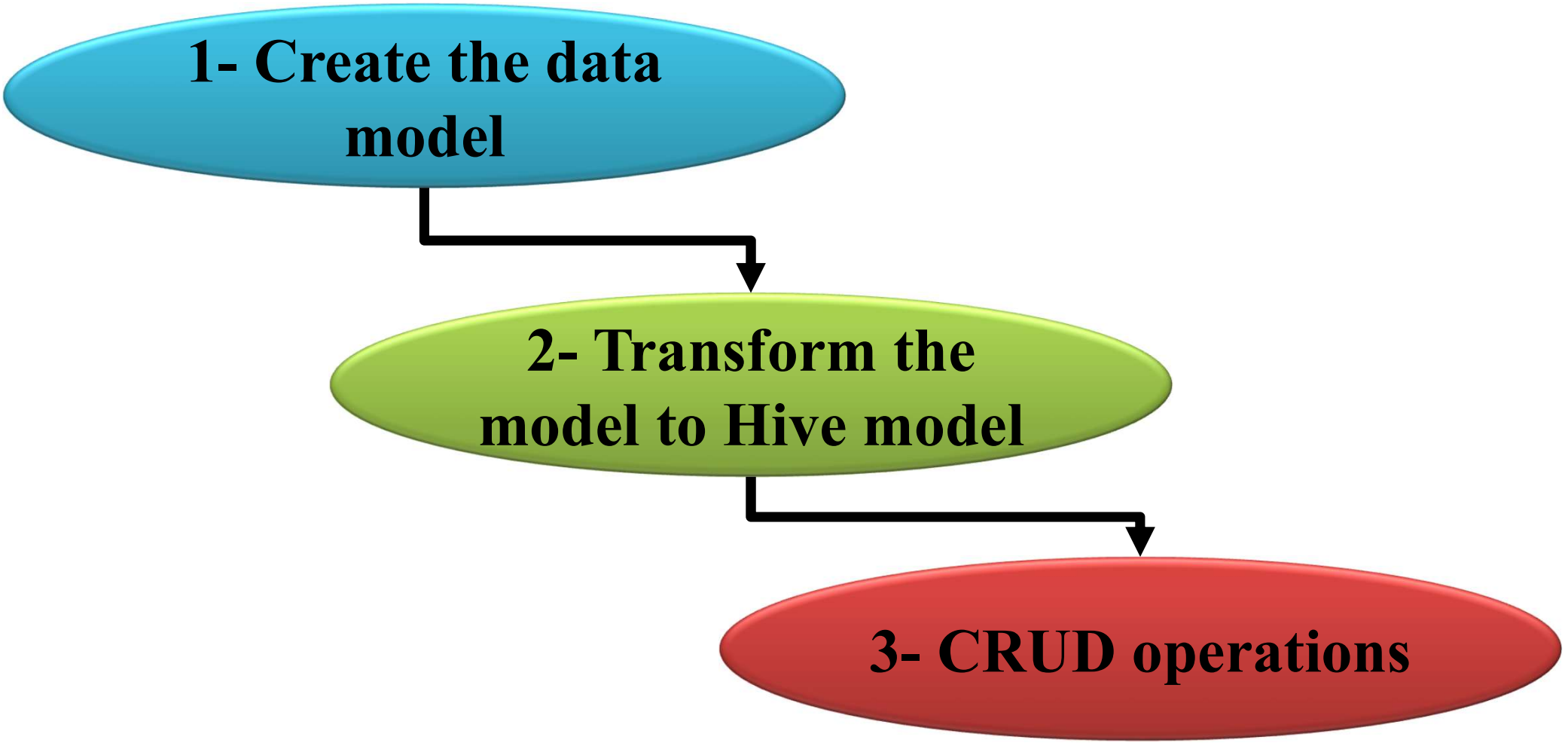
Local storage with Hive

- **Implementation: close a box after all read and write operations are completed to release the memory.**

```
await mybox.close ( );
```

Local storage with Hive

- Implementation for non-primitive types using a type adapter :



Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

1. Create a custom model (a Model Class):

```
// person.dart file
class Person {
    int id;
    String name;
    DateTime birthDate;
    PersonModel (this.id, this.name, this.birthDate);
}
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.A. Modify the Model Class

```
import 'package:hive/hive.dart';
part 'person.g.dart'  //  Model Class Adapter that will be generated
@HiveType (typeId: 1) // Annotate the model class with hive type.
class Person extends HiveObject {
      @HiveField (0) // Annotate all fields of the model class
            int id;
      @HiveField (1)
            String name;
      @HiveField (2)
            DateTime birthDate;
PersonModel (this.id, this.name, this.birthDate); }
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.A. Modify the Model Class

```
import 'package:hive/hive.dart';
part 'person.g.dart'  //  Model Class Adapter that will be generated
@HiveType (typeId: 1) // Annotate the model class with hive type.
class Person extends HiveObject {
      @HiveField (0) // Annotate all fields of the model class
            int id;
      @HiveField (1)
            String name;
      @HiveField (2)
            DateTime birthDate;
PersonModel (this.id, this.name, this.birthDate); }
```

Extends HiveObject is optionally. It is useful to use save( ) & delete( ) methods.

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.B. Generate Model Class Adapter:

From the terminal run the command:
$ flutter pub run build_runner build

It will generate the file 'person.g.dart' which represents the Hive model (Adapter class) of the person Model Class.

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.B. Generate Model Class Adapter:

From the terminal run the command:
$ flutter pub run build_runner build

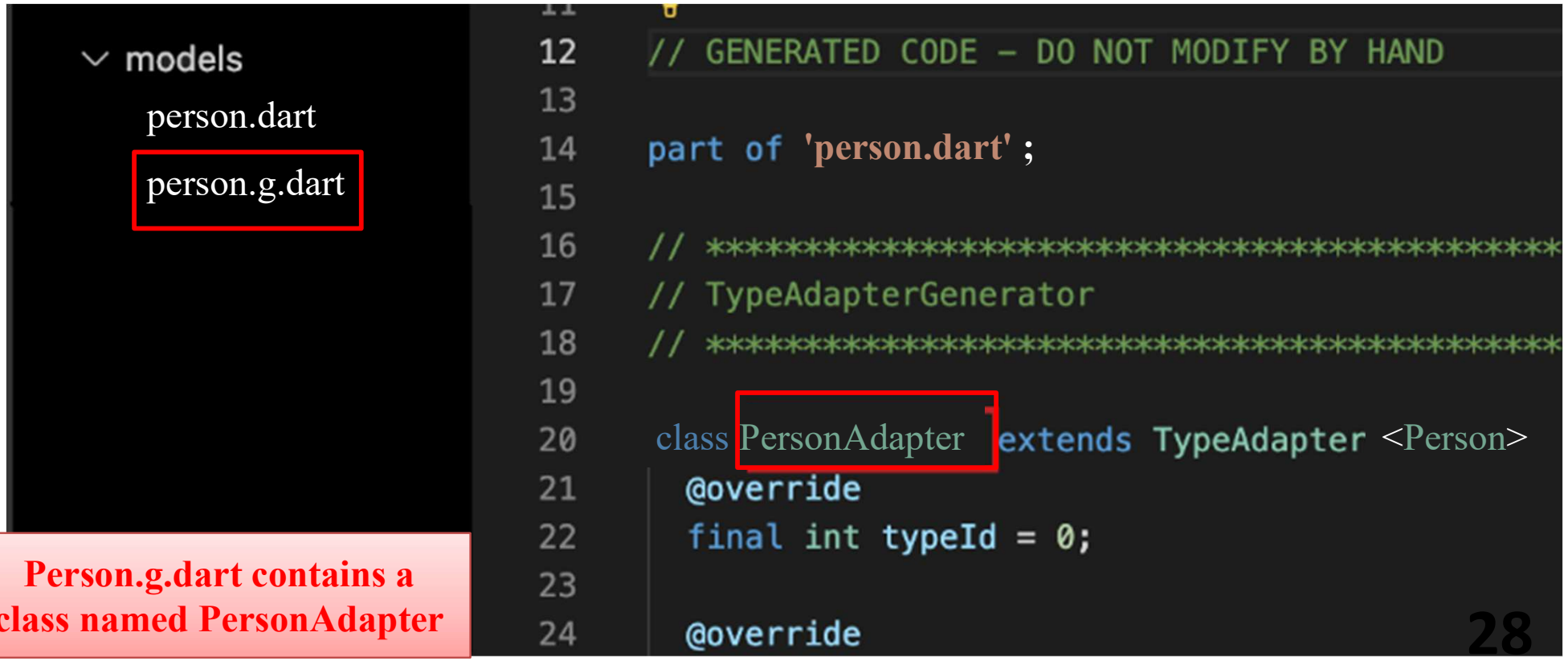
It will generate the file 'person.g.dart' which represents the Hive model (Adapter class) of the person Model Class.

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.B. Generate Model Class Adapter:



Person.g.dart contains a class named PersonAdapter

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

2. Transform Model Class to Hive model:

2.C. Register Model Adapter : add the following line just after Hive initialization in the main method.

```
Hive.registerAdapter( PersonAdapter );
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

3. Use CRUD (Create, Read, Update, Delete) operations:

3.A. Open the box : add the following line just after Model Adapter registration.

```
var mybox = await Hive.openBox <Person> ('boxName');
```

3.B. Add a new value in the box (CRUD):

```
Person  person1 = Person ( 001, 'TOTO', DateTime.parse('01-5-1990'));
 mybox.add(person1); // addAll to add multiple values
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

3. Use CRUD (Create, Read, Update, Delete) operations:

3.C. Retrieve data from the box (CRUD):

```
ValueListenableBuilder( // Updates automatically the UI if changes (it's connect UI to Hive).
                        // it takes a listenable value and a builder function
    valueListenable: mybox.listenable ( ), // Assign the box's listener to the valueListenable
                                           // parameter
    builder: (context, box, child) {//  the builder function
    return ListView.builder( itemCount: mybox.length, // build the ListView using ListTile
                             itemBuilder: (context, index) {
                             final Person person= mybox.getAt [index] ;
                                  return ListTile(title: Text(person.name),
                                                  leading: Text(person.id.toString()),
                                                  subtitle: Text(
                                                 person.birthDate.toLocal().toString()),);
    },);},)
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

3. Use CRUD (Create, Read, Update, Delete) operations:

3.D. Update data of the box (CRUD):

```
int lastIndex = mybox.toMap().length - 1;
    if (lastIndex < 0) return;
    Person person2 = mybox.values.toList( )[lastIndex];
     person2.birthDate = DateTime.now();
    mybox.putAt(lastIndex, person2); // put(key,value),
    putAt(index, value), putAll (Map <key,value> entries)
```

OR

```
person2.save ( ); // use save method to get benefits from HiveObject cla
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

3. Use CRUD (Create, Read, Update, Delete) operations:

3.E. Delete data from the box (CRUD):

```
int lastIndex = mybox.toMap().length - 1; //get last index
if (lastIndex >= 0) mybox.deleteAt(lastIndex); //delete (key),
    deleteAt (index), deleteAll(Iterable <dynamic> keys)
```

OR

```
person2.delete ( ); // use delete method to get benefits from
                    HiveObject class
```

Local storage with Hive

- **Implementation: use type adapter for any custom type (non-primitive types).**

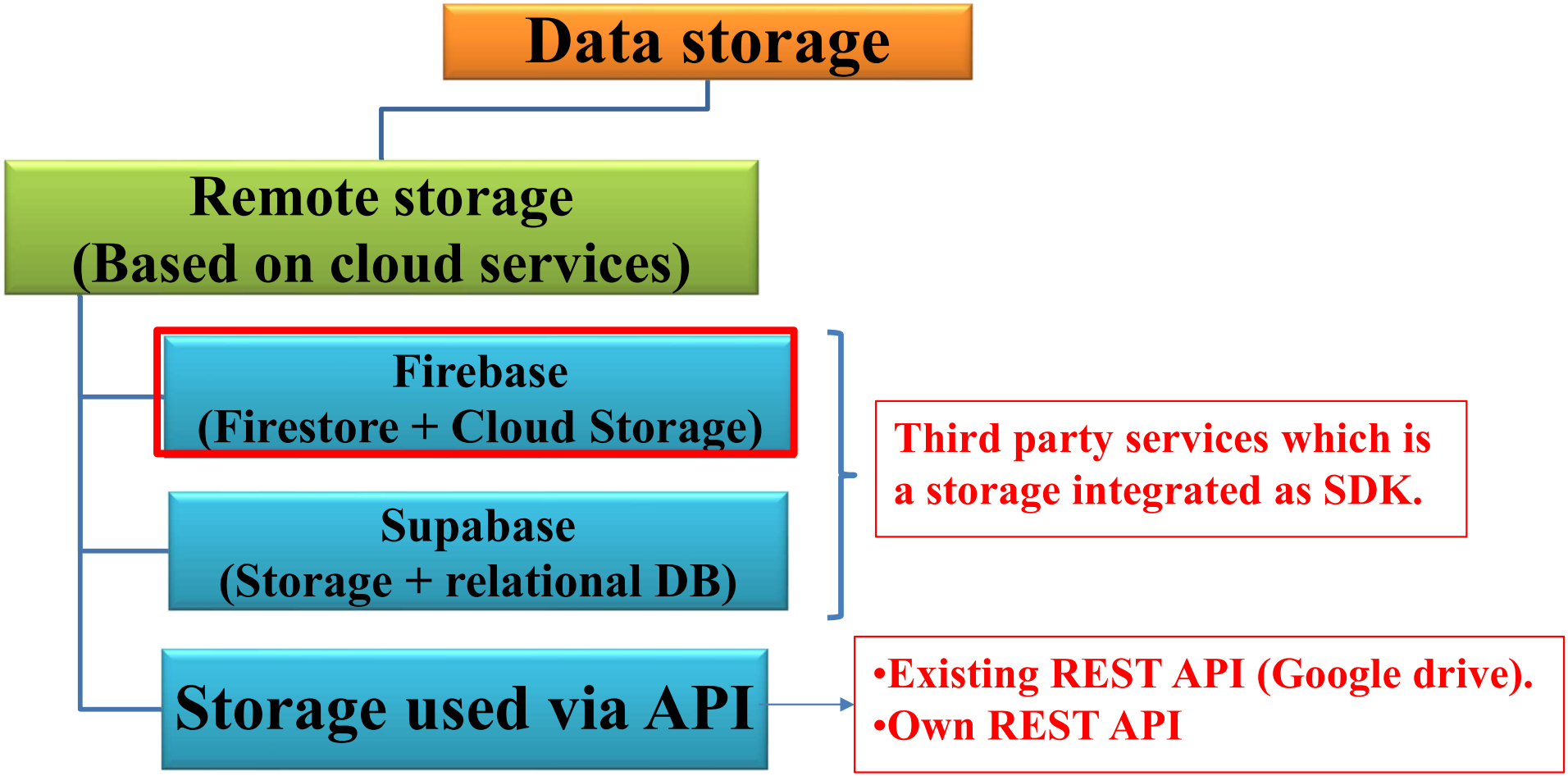
3. Use CRUD (Create, Read, Update, Delete) operations:

3.F. Close the box : add the following line in the dispose method.

```
Hive.close ( ); // Close all boxes
Hive.box ('boxName').close ( ); //  Close a specific box
```

Data storage methods

- Classification :



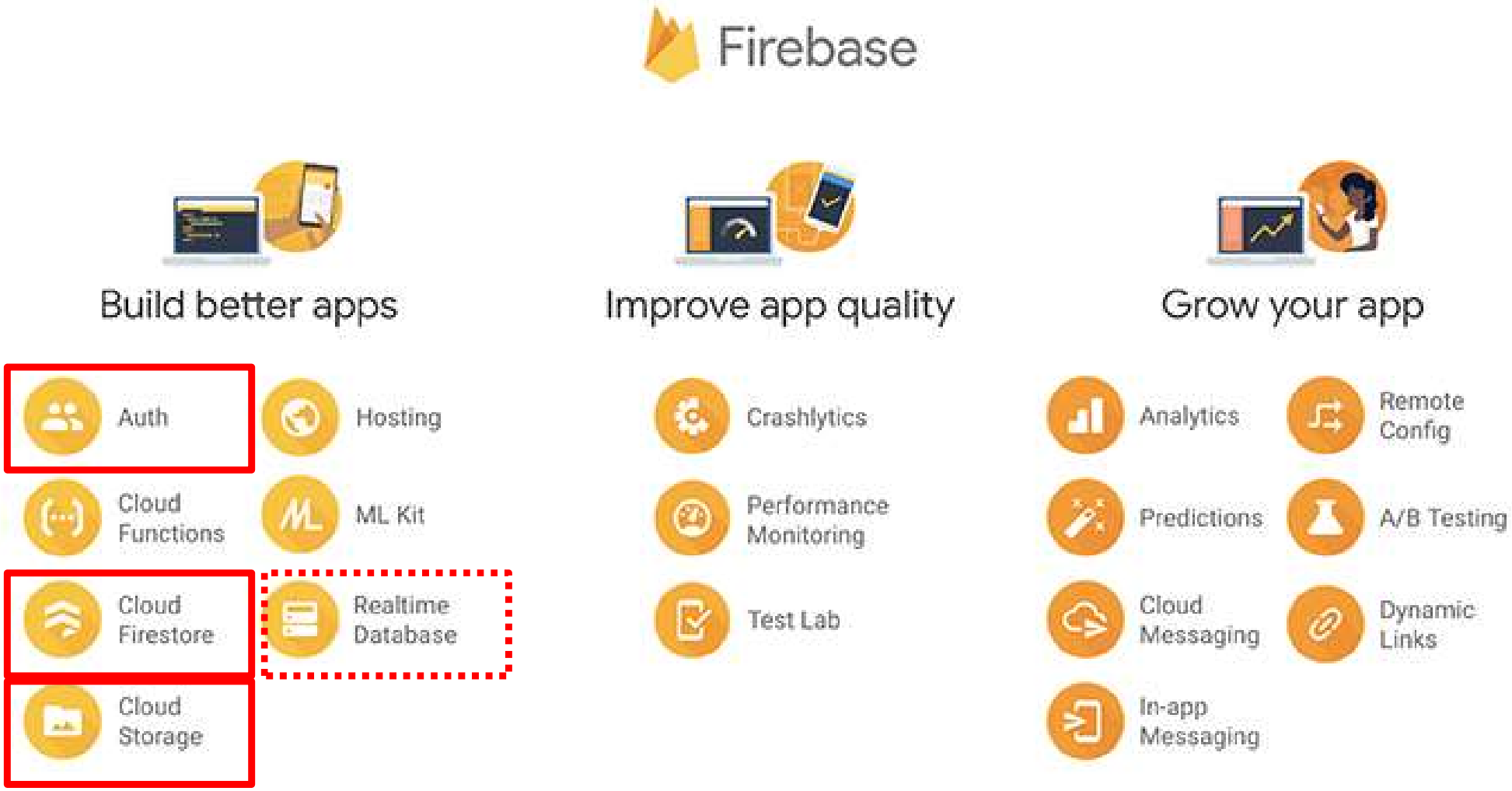
Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Google Firebase

- **Google Firebase** is a powerful backend-as-a-service (**BAAS**) platform that offers a suite of tools and services to help developers build, scale, and maintain web and mobile applications.
- It provides developers with easy-to-use features like real-time & **firestore** databases, **authentication**, hosting, storage, and machine learning capabilities.
- It's ideal for applications that require **remote** data storage, user **authentication**, and real-time **updates**.

Google Firebase services (products)



Firestore vs. Realtime database:
https://firebase.google.com/docs/database/rtdb-vs-firestore?hl=fr

Google Firebase setup

=> FlutterFire

- **Using firebase with flutter requires some setup steps.**

1) Prerequisites :

   a) Create a Firebase account on: *https://firebase.google.com/*

   b) Create a Firebase project from the console (Disable Google Analytics).

   c) Create a flutter project.

Google Firebase setup

Flutter + Firebase => FlutterFire

---

2) Adding Firebase to Flutter app :

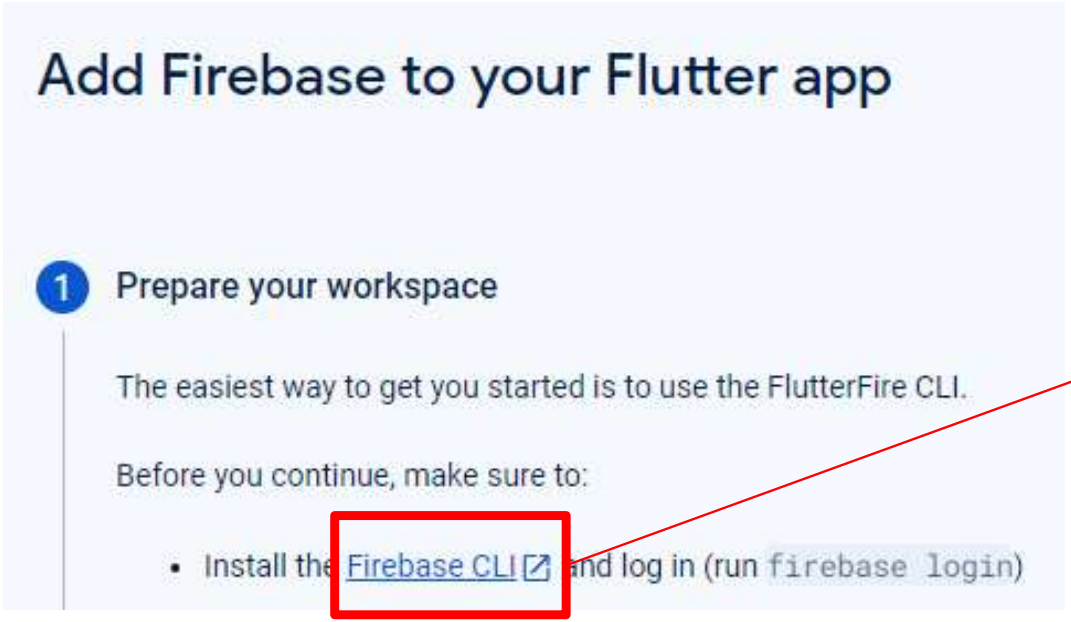
FlutterFire CLI, a **new** tool that provides commands to help **ease** installation process of FlutterFire across all supported platforms.

Google Firebase setup

Flutter + Firebase => FlutterFire

---

a) Install the **Firebase** CLI:



Google Firebase setup

=> FlutterFire

a) Install the **Firebase** CLI:

Open the terminal from the binary & edit environment variable of the system (PATH)

Provides npm commands tool

Two possible options	
Standalone binary	*npm*
1. Download the Firebase CLI binary for Windows. 2. Open the shell from the binary to run firebase command. 3. Login : ***firebase login*** 4. Test Firebase CLI : ***firebase projects:list***	1. Install Node.js 2. Install the Firebase CLI via npm by running the command: ***npm install -g firebase-tools*** 3. Login : ***firebase login*** 4. Test Firebase CLI : ***firebase projects:list***

Google Firebase setup

=> FlutterFire

b) Install the **FlutterFire** CLI by running (from any directory) the command :

```
dart pub global activate flutterfire_cli
```

c) Execute **FlutterFire** CLI by running the command

```
flutterfire configure --project=example-7f6bb
```

from the **root** of the Flutter project.

- **This automatically registers your per-platform apps with Firebase and adds a lib/firebase_options.dart configuration file to your Flutter project.**
- **This is the only step to repeat for each new project or new target platform.**

Google Firebase setup

=> FlutterFire

d) Add Firebase plugins:

```
flutter pub add firebase_core
```

```
dependencies: firebase_core: ^2.25.3
```

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

e) Initialize Firebase :

```
void main() async {
        WidgetsFlutterBinding.ensureInitialized();
        await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform,);
runApp(MyApp()); }
```

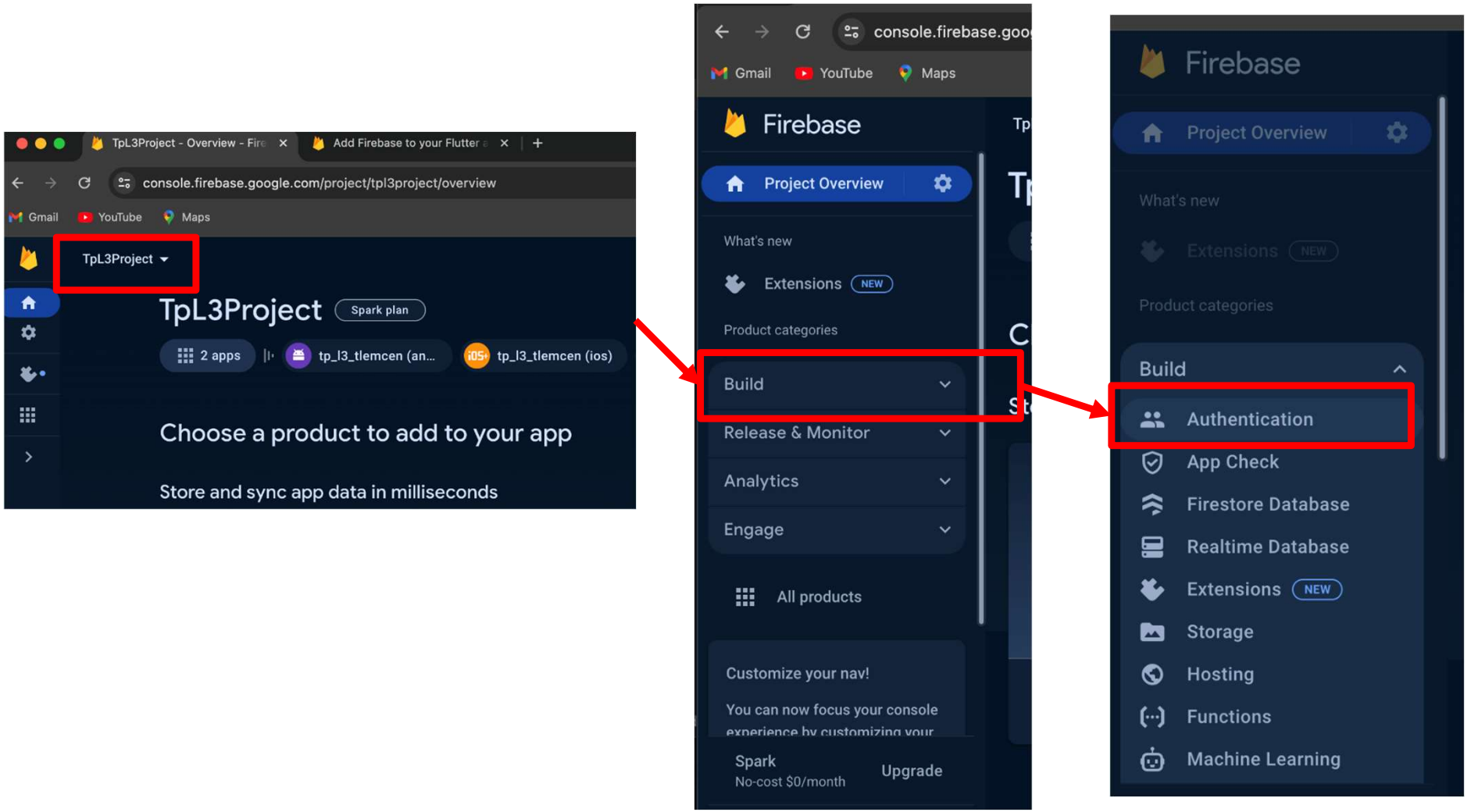
Plan

1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

User authentication with Firebase

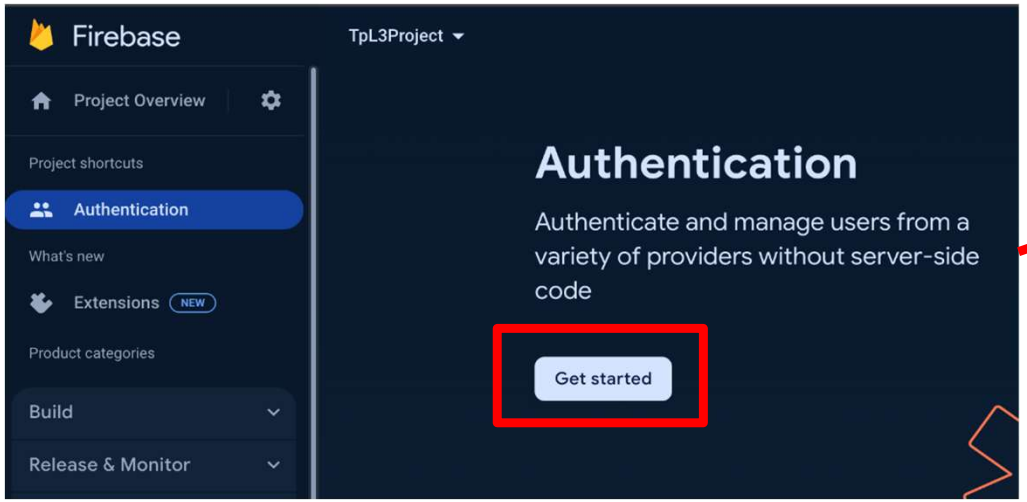
- Integrate secure authentication into Flutter App.

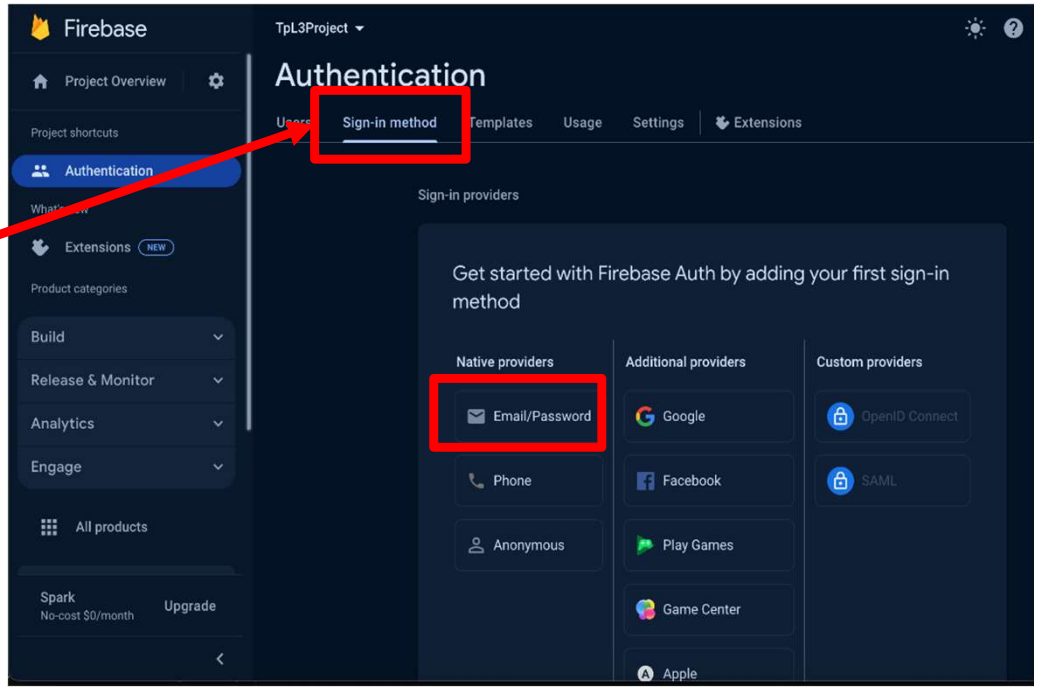
1. Activate Sign-in provider (Sign-in method):



User authentication with Firebase

- Integrate secure authentication into Flutter App.
  1. Activate Sign-in provider (Sign-in method):

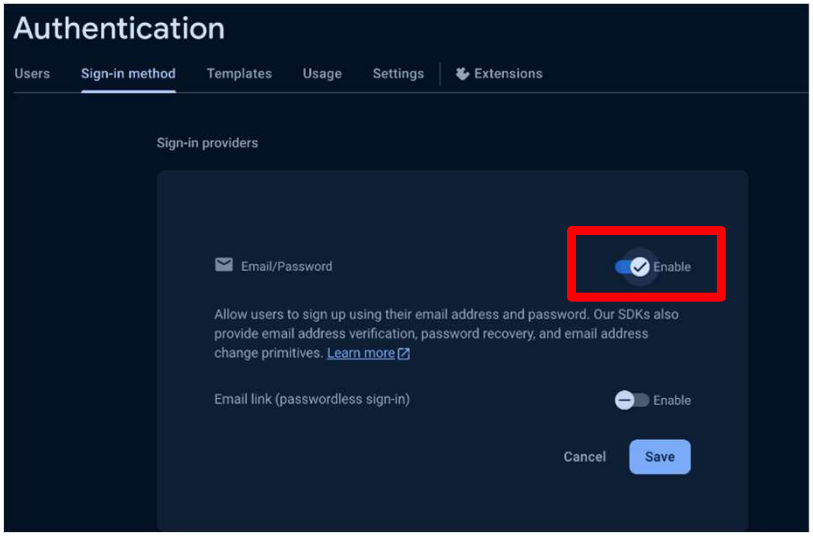


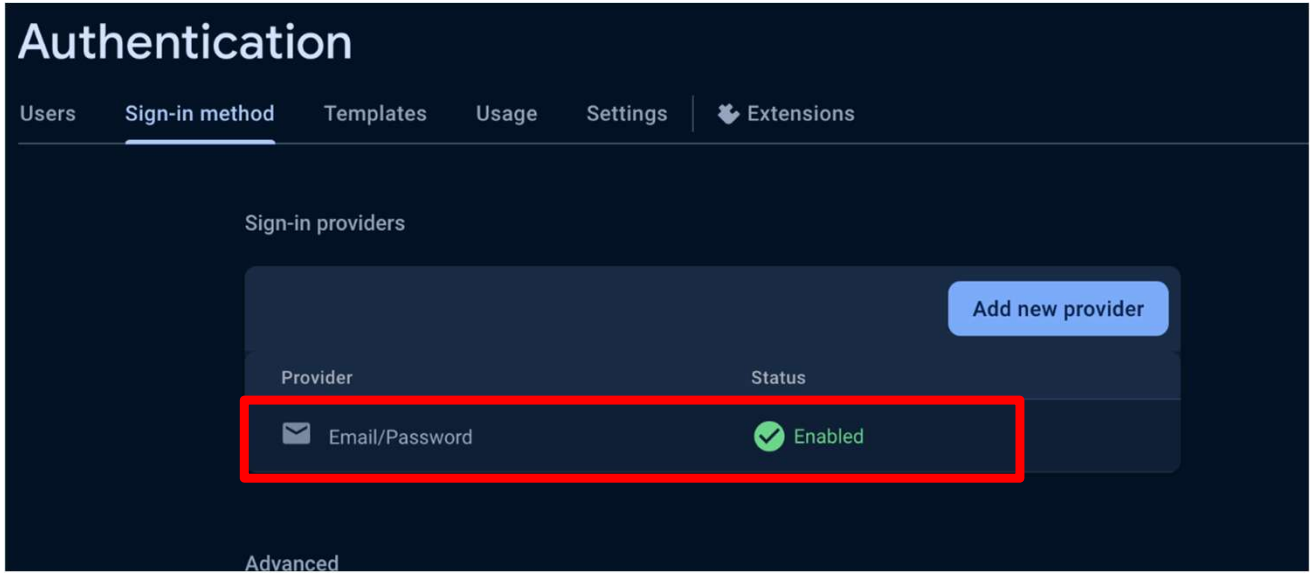


User authentication with Firebase

- Integrate secure authentication into Flutter App.

1. Activate Sign-in provider (Sign-in method):





User authentication with Firebase

- Integrate secure authentication into Flutter App.

2. Add firebase_auth plugin :

```
flutter pub add firebase_auth
```

```
dependencies: firebase_auth: ^4.17.4
```

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

3. Choose the *createUserWithEmailAndPassword* method:

```
UserCredential userCredential = await FirebaseAuth.instance.
createUserWithEmailAndPassword (email: "myEmail", password:  "myPWD");
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

  3. Choose the ***createUserWithEmailAndPassword*** method: **catch exceptions.**

```
try {
UserCredential userCredential = await FirebaseAuth.instance
.createUserWithEmailAndPassword (email: "myEmail", password:  "myPWD");
} on FirebaseAuthException catch (e) {
                        if (e.code == 'weak-password') {print('The password is too weak');}
                        else if (e.code == 'email-already-in-use') {
                        print('Account already exists for that email');}
                        else{print('Failed to signup: error code: ${e.code}, ${e.message}');}
                       }
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

  4. Sign in using the ***signInWithEmailAndPassword*** method:

```
UserCredential userCredential = await FirebaseAuth.instance
.signInWithEmailAndPassword (email: "myEmail", password:  "myPWD");
```

catch exceptions:

```
try {
UserCredential userCredential = await FirebaseAuth.instance
.signInWithEmailAndPassword (email: "myEmail", password:  "myPWD");
} on FirebaseAuthException catch (e) {
                    if (e.code == 'invalid-email') {print('Invalid Email');}
                    else if (e.code == 'invalid-credential') {
                    print('user-not-found OR Wrong Password');}
                    else{print('Failed to login with error code: ${e.code}, ${e.message}');}
                    }
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

5. Check current authentication state:

```
// Listen to the current user authentication state
FirebaseAuth.instance.authStateChanges().listen((User? user) {
        if (user == null) {  print ('User is currently signed out!'); }
        else {print ('User is signed in!'); }
        });
```

```
// Condition to check if the current user is sign in or sign out (null).
FirebaseAuth.instance.currentUser == null
```

6. Call signOut() method to sign out a user :

```
await FirebaseAuth.instance.signOut();
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

  7. Verify the address email by sending a link:

```
// Send the link to the email address specified by the current user
FirebaseAuth.instance.currentUser!.sendEmailVerification();

// Check if the current user has clicked on the received link
FirebaseAuth.instance.currentUser!.emailVerified
```

```
// If the user doesn't activate the account, it can be deleted
FirebaseAuth.instance.currentUser!.delete();
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

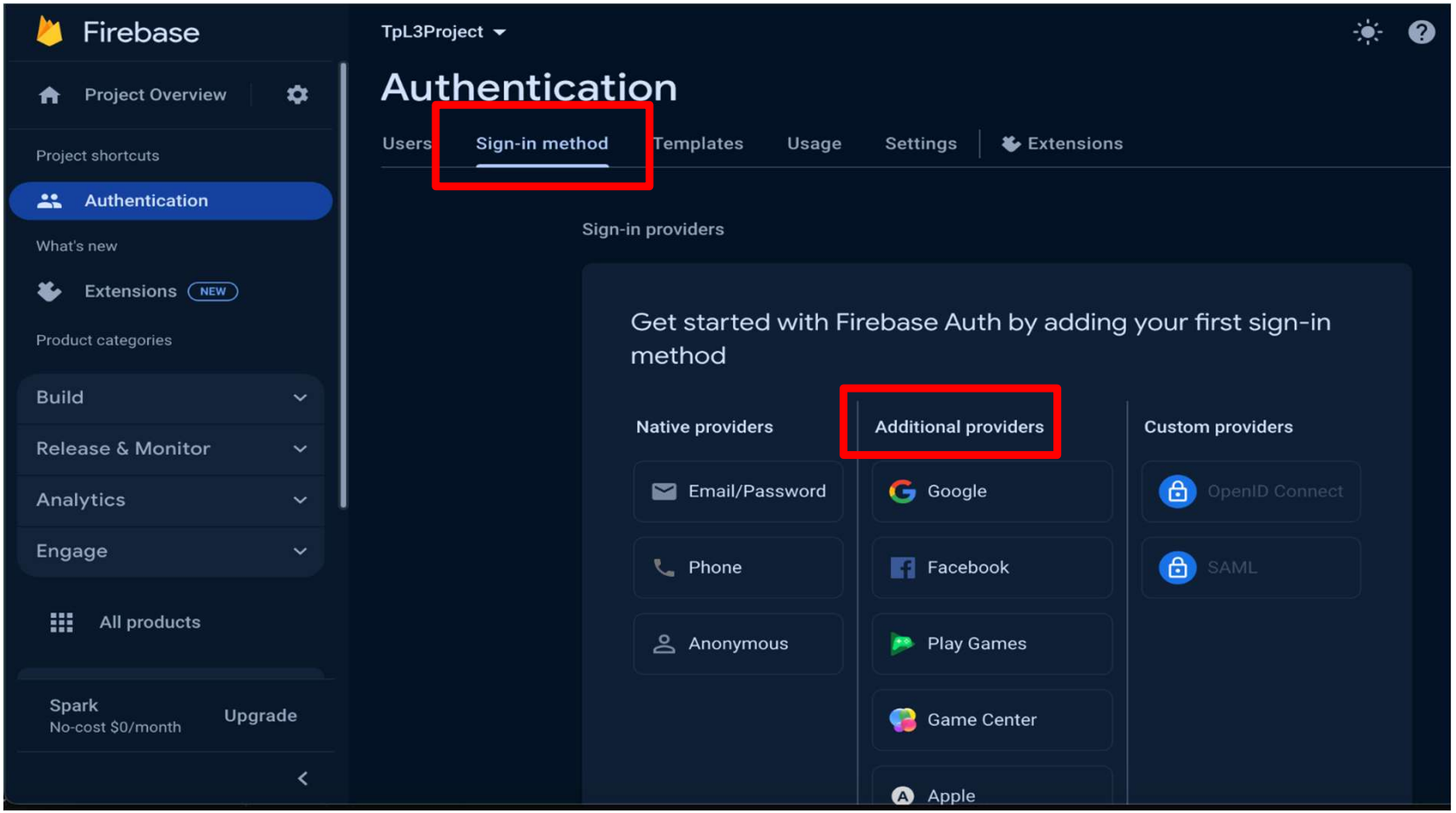
8. Reset the password :

```
await FirebaseAuth.instance.sendPasswordResetEmail
                                (email:"MyEmail");
```

User authentication with Firebase

- Integrate secure authentication into Flutter App.

9. Sign in with Google account:



User authentication with Firebase

- Integrate secure authentication into Flutter App.

  9. Sign in with Google account:

```
// Trigger the authentication flow
final GoogleSignInAccount? googleUser = await GoogleSignIn().signIn();

// Obtain the auth details from the request
final GoogleSignInAuthentication? googleAuth = await googleUser?.authentication;

// Create a new credential
final credential = GoogleAuthProvider.credential ( accessToken:
googleAuth?.accessToken, idToken: googleAuth?.idToken,);

// Sign in using the UserCredential
await FirebaseAuth.instance.signInWithCredential(credential);
```

```
// Sign out a user connected with a Google account
GoogleSignIn googleSignIn = GoogleSignIn();
googleSignIn.disconnect();
```

Plan

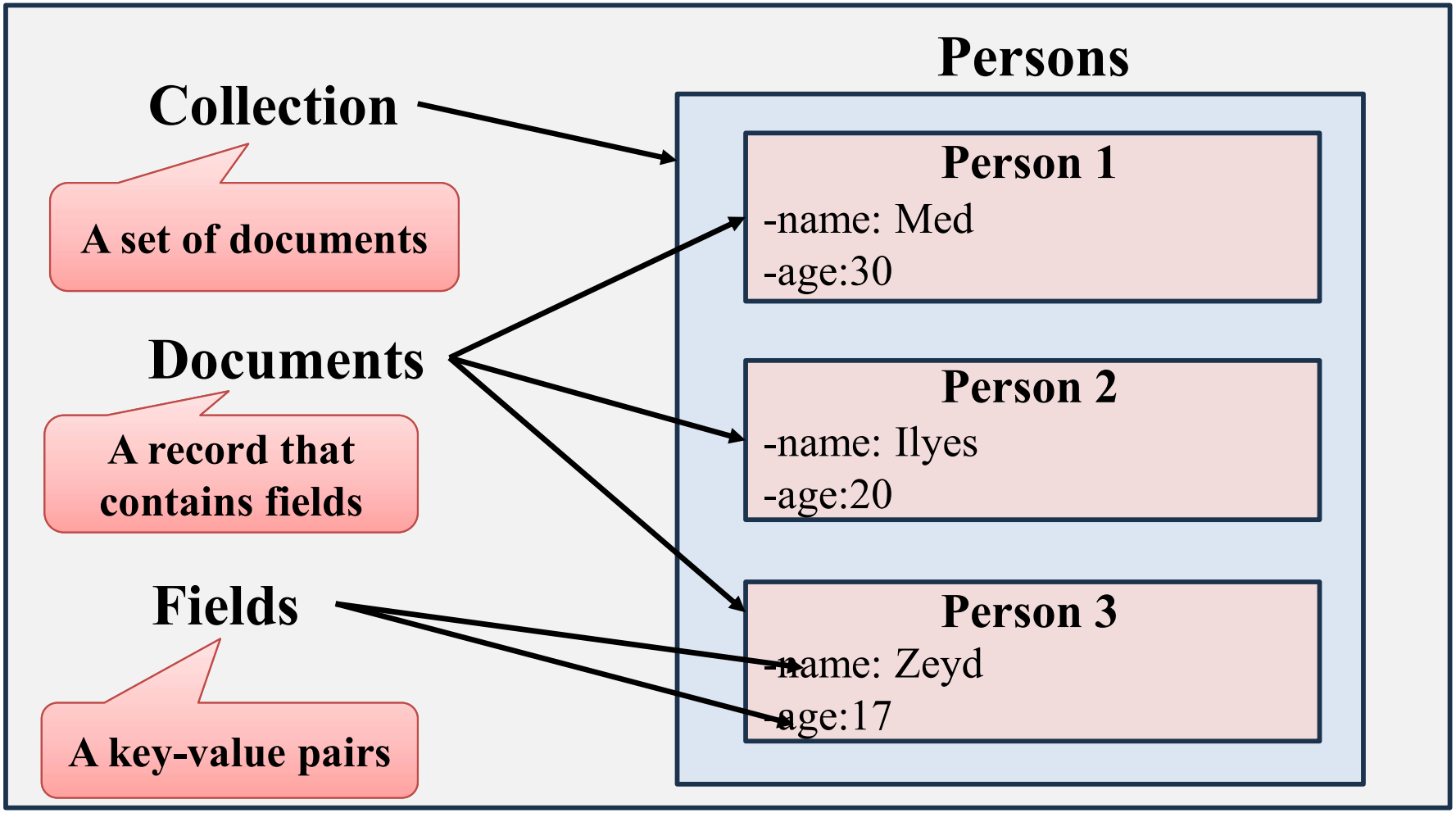
1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

Remote storage with Firestore

- **Firebase** is a popular choice for **cloud** storage in Flutter. It provides a **NoSQL** database called **Firestore**.
- It allows the storage of **files** such as JSON documents.
- It's optimized for **large** collections of **small** documents.

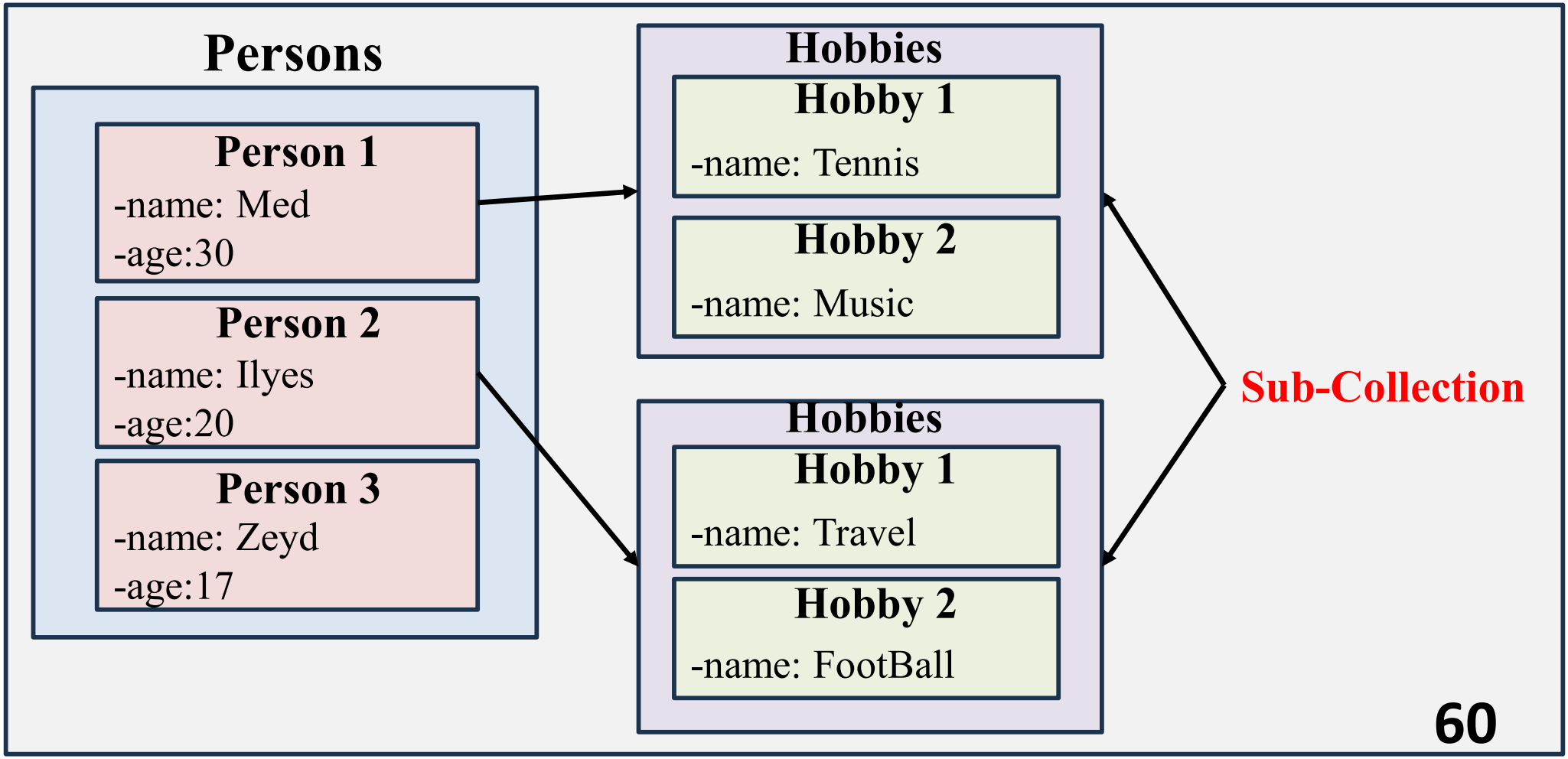
Remote storage with Firestore

- **Storage principle:** Firestore database uses the **collection-document** model to store the data.



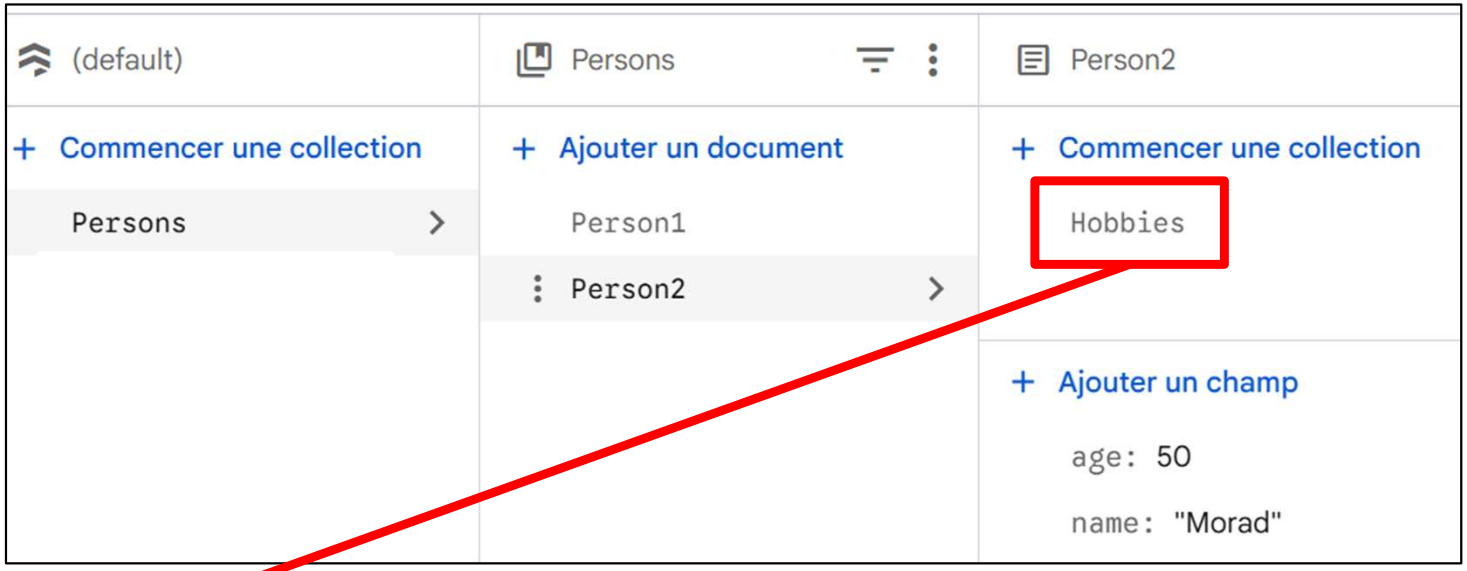
Remote storage with Firestore

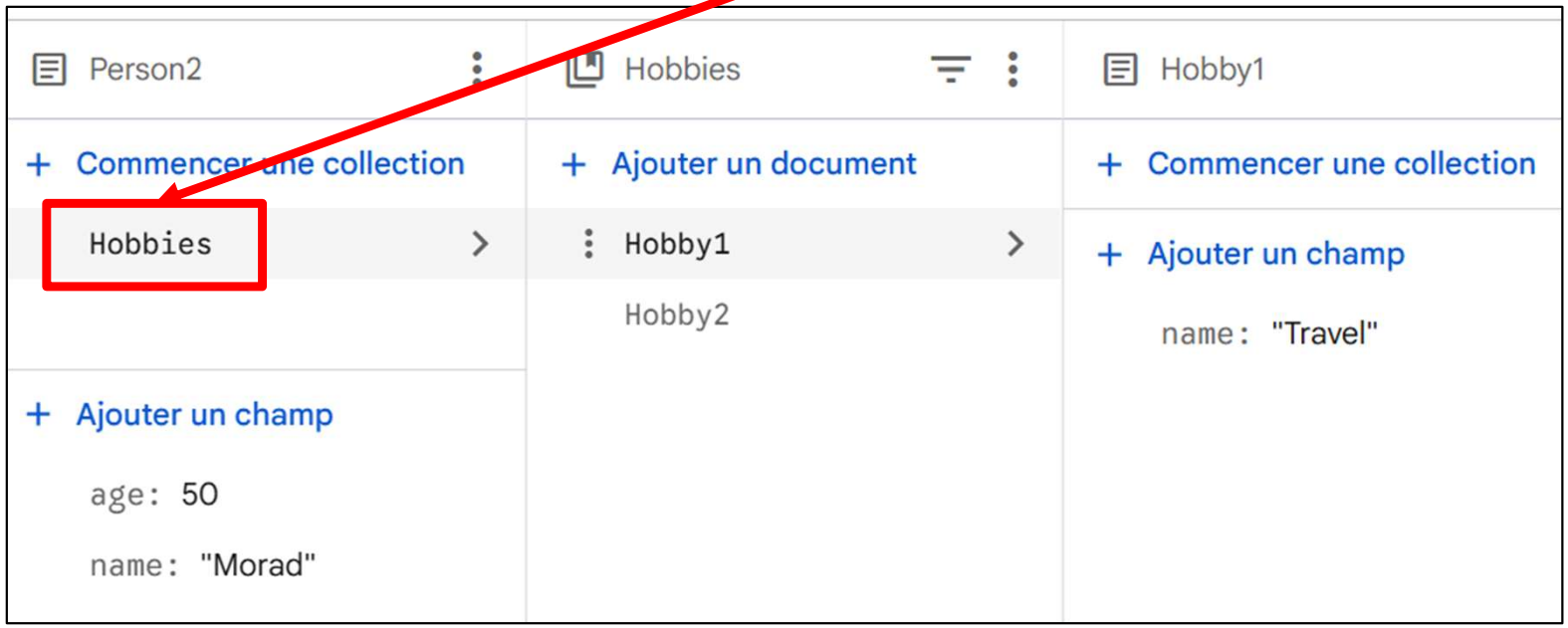
- **Storage principle:** Firestore database uses the **collection-document** model to store the data.



Remote storage with Firestore

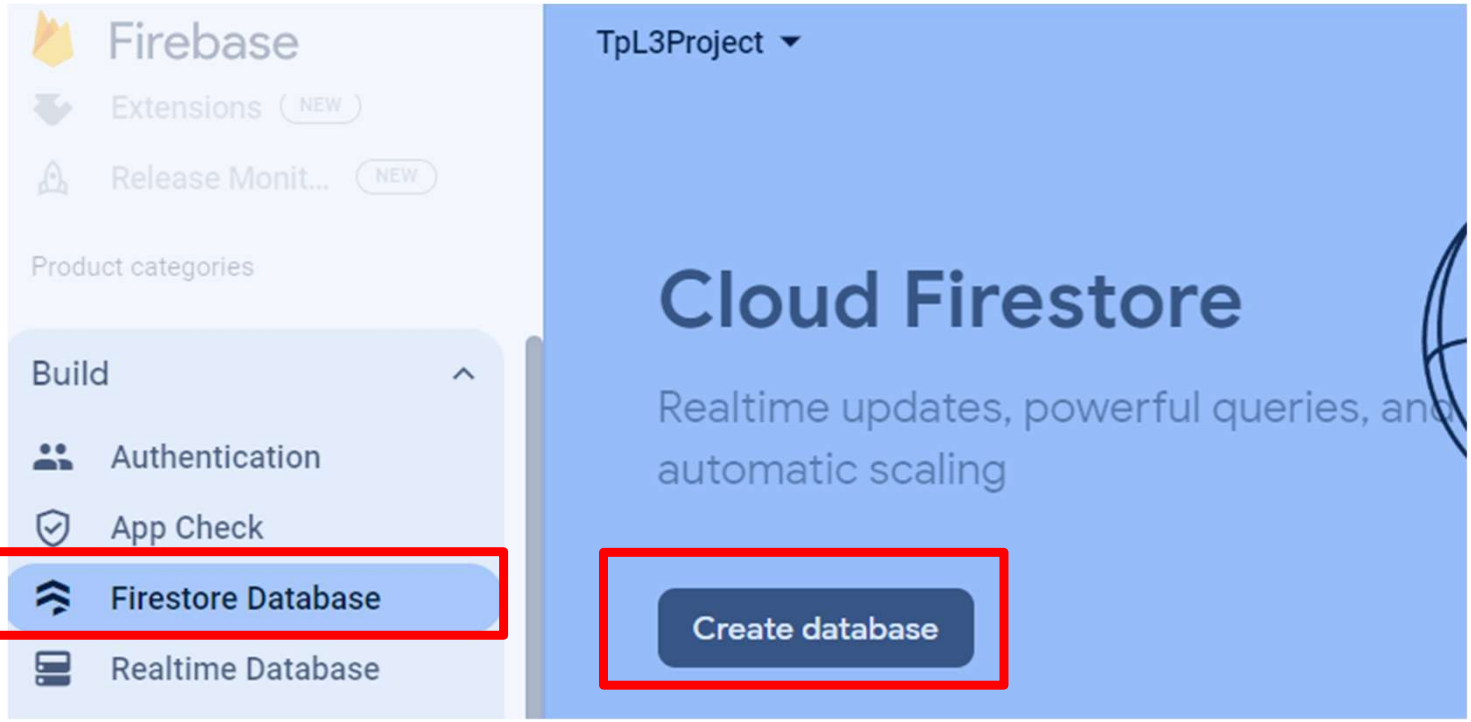
- On the server side the Firestore database looks like this:





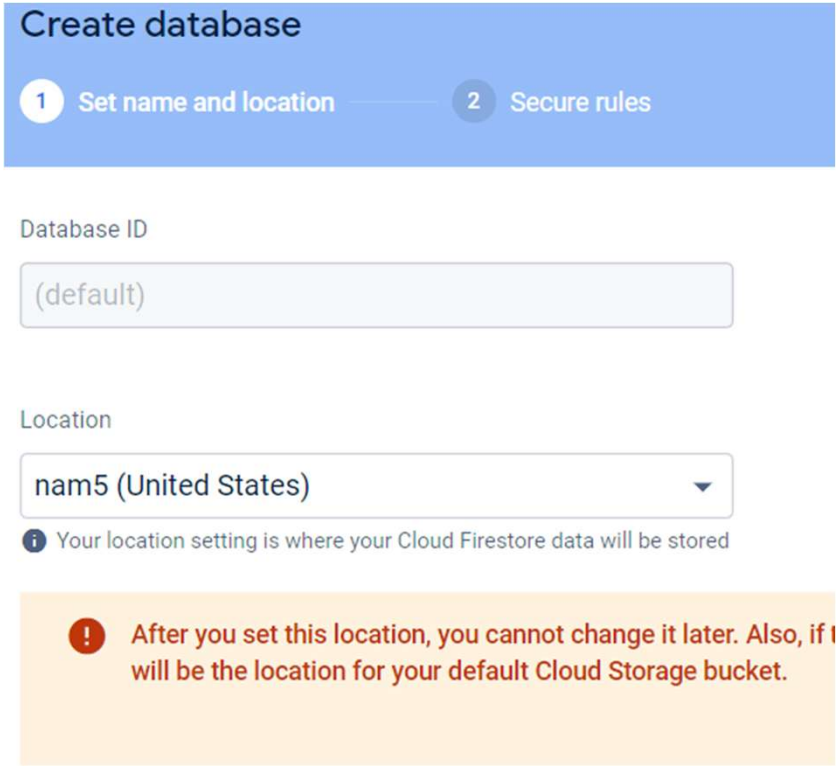
Remote storage with Firestore

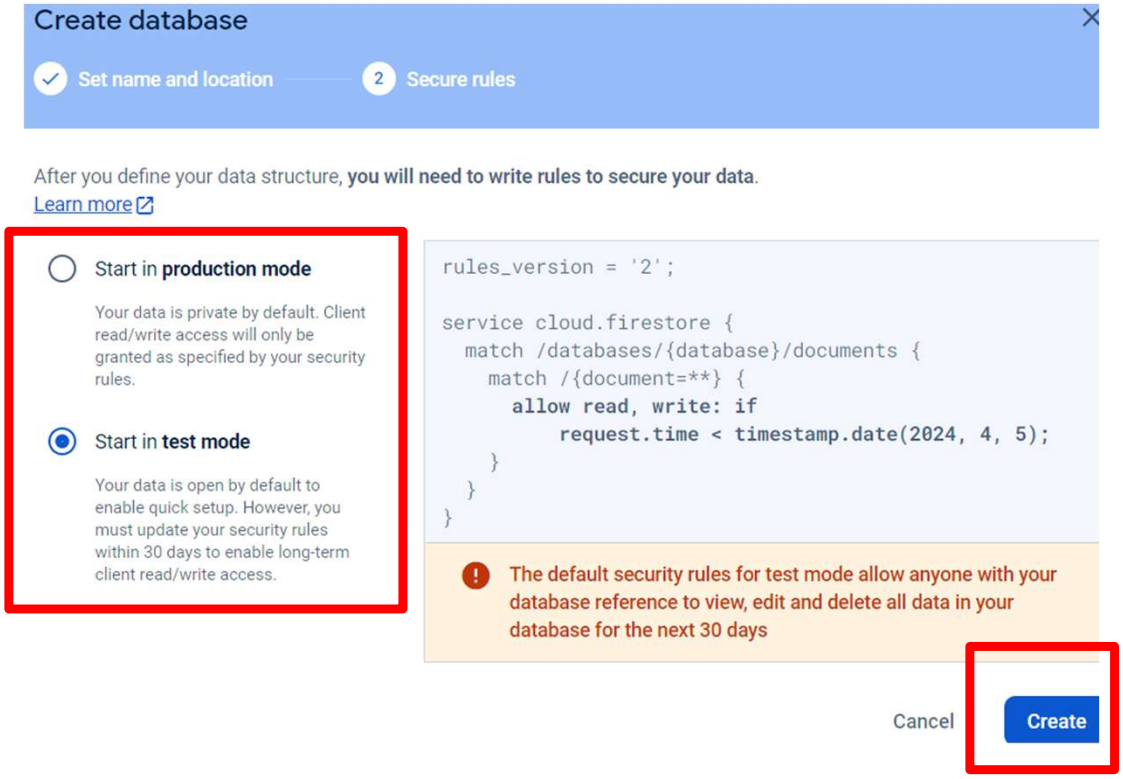
- **Implementation:** create the Firestore Database from Firebase Console.



Remote storage with Firestore

- **Implementation:** create the Firestore Database from Firebase Console.





After database creation, don't forget to import the cloud_firestore package from the Flutter project.

Remote storage with Firestore

- **Implementation:** add data.

```
// Get a reference (myPersons) to a collection named persons
CollectionReference myPersons = FirebaseFirestore.instance.collection('persons');

// Add a data (document + fields) to the collection
addPerson (personName, personAge) async {

try { await myPersons.add({"name": personName, "age": personAge,});
      print("Person added");
} catch (error) {
      print("Failed: $error");
  }
}
```

Remote storage with Firestore

- **Implementation:** add data as a subcollection.

```
// Add a subcollection (nested collection) to a document
  CollectionReference myHobbies =
FirebaseFirestore.instance.collection('persons').doc(personId). collection('hobbies');

// Add a data (document + fields) to the subcollection

addHobby (hobbyName) async {

try { await myHobbies.add({"name": hobbyName});
        print(" Hobby added");
} catch (error) {
        print("Failed: $error");
    }
}
```

Remote storage with Firestore

- **Implementation:** get data by calling the "**docs**" property from the ***QuerySnapshot*** returned by a collection query (***get*** method).

```
List<QueryDocumentSnapshot> myData = [];
// Get documents from a collection
getPersons ( ) async {
QuerySnapshot result = await FirebaseFirestore.instance.collection('persons').get( );
myData.addAll (result.docs);
}
```

```
List<Map<String, dynamic>> myData = [];
// Get a document from a collection
getAPerson (String docID ) async {
DocumentSnapshot result = await
FirebaseFirestore.instance.collection('persons').doc(docID).get( );
myData.add (result.data ( ) as Map<String, dynamic> );
}
```

Remote storage with Firestore

- **Implementation:** get data of a subcollection.

```
List<QueryDocumentSnapshot> myData = [];

 // Get documents from the 'hobbies' sub-collection of a person document

getHobbies (String personId ) async {
QuerySnapshot result = await FirebaseFirestore.instance.collection('persons')
.doc(personId).collection('hobbies').get ( );

myData.addAll (result.docs);
}
```

Remote storage with Firestore

Implementation: retrieve filtered data from a collection based on a condition.

```
// Get a reference to the 'persons' collection

CollectionReference myPersons = FirebaseFirestore.instance.collection('persons');
List<QueryDocumentSnapshot> myData = [];

// Get documents by filtering
getFilteredMyPersons ( ) async {
QuerySnapshot result = await myPersons.where( 'age' , isEqualTo : 30).get( );
myData.addAll (result.docs);
}
```

Remote storage with Firestore

- **Implementation:** retrieve filtered data from a collection based on a condition.

```
where( 'age' , isGreaterThan : 30).get( );
where( 'age' , isGreaterThanOrEqualTo : 30).get( );
where( 'age' , isLessThan : 30).get( );
where( 'age' , isNotEqualTo : 30).get( );
```

```
where( 'age' , whereIn : [30,50,20]).get( );
where( 'age' , whereNotIn : [30,50,20]).get( );
where( 'lang' , arrayContains : 'fr').get( );
where( 'lang' , arrayContainsAny : ['fr','ar']).get( );
```

Remote storage with Firestore

- **Implementation:** retrieve data from a collection ordered by a specific field .

```
// Get a reference to the 'persons' collection
CollectionReference myPersons = FirebaseFirestore.instance.collection('persons');
List<QueryDocumentSnapshot> myData = [];

// Get documents ordered by age
getOrderedMyPersons ( ) async {
QuerySnapshot result = await myPersons.orderBy( 'age' ,descending: true ).get( );
myData.addAll (result.docs);
}
```

```
// Combine order with startAt/startAfter (endAt/endBefore)
myPersons.orderBy( 'age').startAt([20]).get( ); // age >= 20
myPersons.orderBy( 'age' ,descending: true).startAt([20]).get( ); // age <= 20
myPersons.orderBy( 'age').startAfter([20]).get( ); // age > 20
myPersons.orderBy( 'age' ,descending: true).startAfter([20]).get( ); // age < 20
```

Remote storage with Firestore

- **Implementation:** retrieve a limited amount of data from a collection.

```
// Get a reference to the 'persons' collection
CollectionReference myPersons = FirebaseFirestore.instance.collection('persons');
List<QueryDocumentSnapshot> myData = [];

// Get a limited number of documents
getLimitedMyPersons ( ) async {
QuerySnapshot result = await myPersons.limit( 10 ).get( );
myData.addAll (result.docs);
}
```

```
// Combine order & limit
myPersons.orderBy( 'age').limit(10).get( );
```

Remote storage with Firestore

- **Implementation:** update data.

```
// Update a specific document in the 'persons' collection based on its id
updtaeAPerson (String newName, String personId) async {
await FirebaseFirestore.instance.collection('persons').doc(personId).update( {
"name": newName});
}
```

- Update a subcollection data.

```
// Update a specific document in a sub-collection based on its id

updtaeAHobby (String newName , String personId, String hobbyId) async {
await FirebaseFirestore.instance.collection('persons').doc(personId).
collection('hobbies').doc(hobbyId).update( { "name": newName});
}
```

Remote storage with Firestore

- **Implementation:** set data. ***Set*** method works as ***Update*** if the document exists, and as ***Add*** if the document doesn't.

```
// Set (create or overwrite) a specific document in the 'persons' collection using its id
setAPerson (String newName , String personId) async {
await FirebaseFirestore.instance.collection('persons').doc(personId ).
                                                    set( { "name": newName});
}
```

- If the document exists ***SetOptions*** is used to avoid deleting the no changed fields (e.g. "age").

```
await FirebaseFirestore.instance.collection('persons').doc(personId).
    set ({ "name": newName}, SetOptions ( merge: true));
```

Remote storage with Firestore

- **Implementation:** delete data.

```
// Delete a specific document from the 'persons' collection based on its id

deleteAPerson (String personId ) async {
await FirebaseFirestore.instance.collection('persons').doc(personId).delete( );
}
```

- Delete a subcollection data.

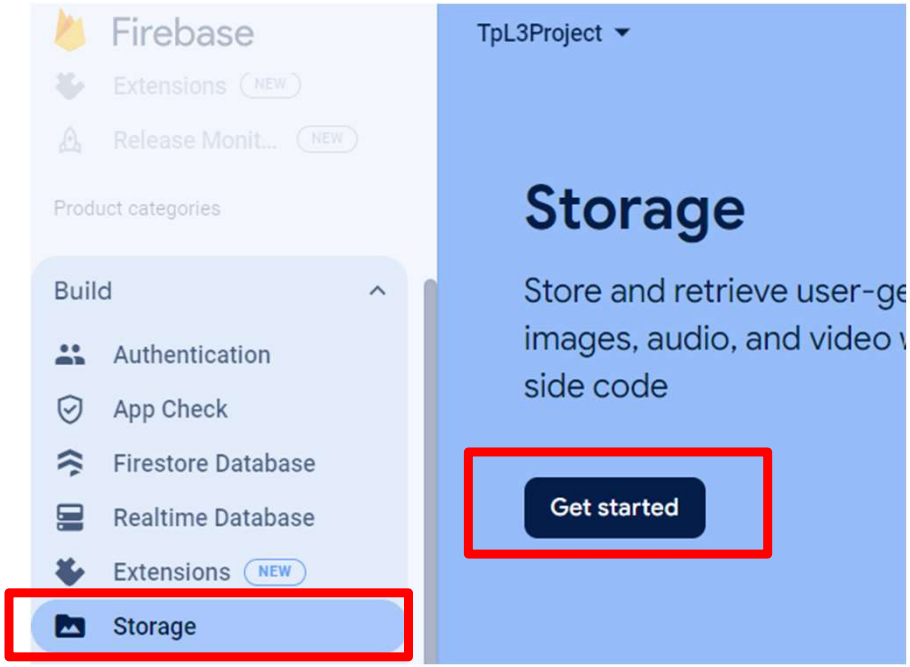
```
// Delete a specific document from the 'hobbies' sub-collection using its document ID

deleteAPerson (String personId, String hobId) async {
await FirebaseFirestore.instance.collection('persons').doc(personId).
collection('hobbies').doc(hobId). delete( );
}
```

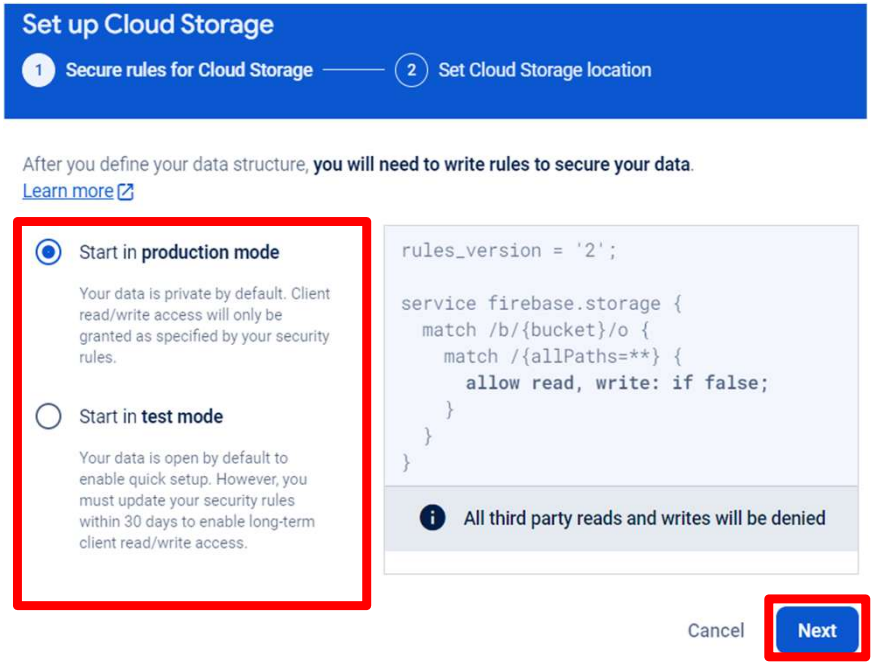
Plan

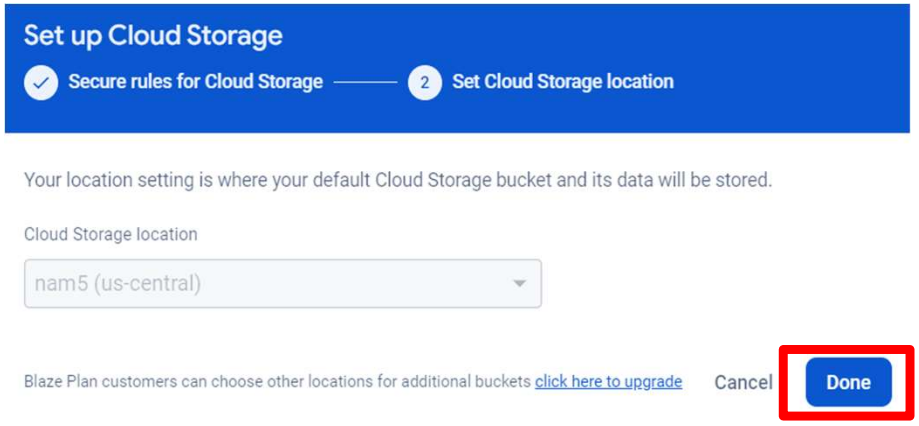
1. **Introduction.**
   - Data persistence.
   - Data storage methods.
2. **Local storage with "Hive".**
3. **Remote storage with "Firebase".**
   - User authentication with *Firebase Auth*.
   - NoSQL storage with *Firestore*.
   - Cloud file storage with *Firebase Storage*.

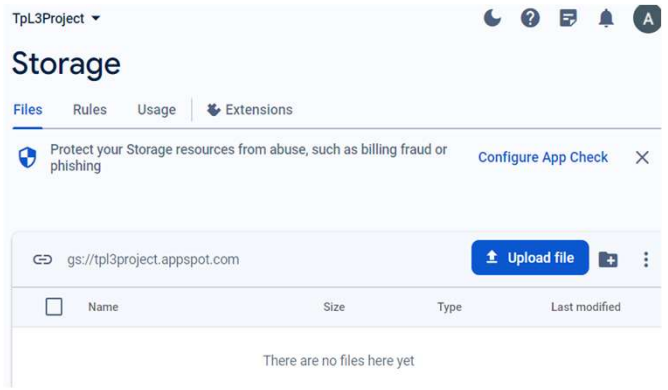
Remote storage techniques: Cloud storage

- **Implementation:** create the Storage from Firebase Console.



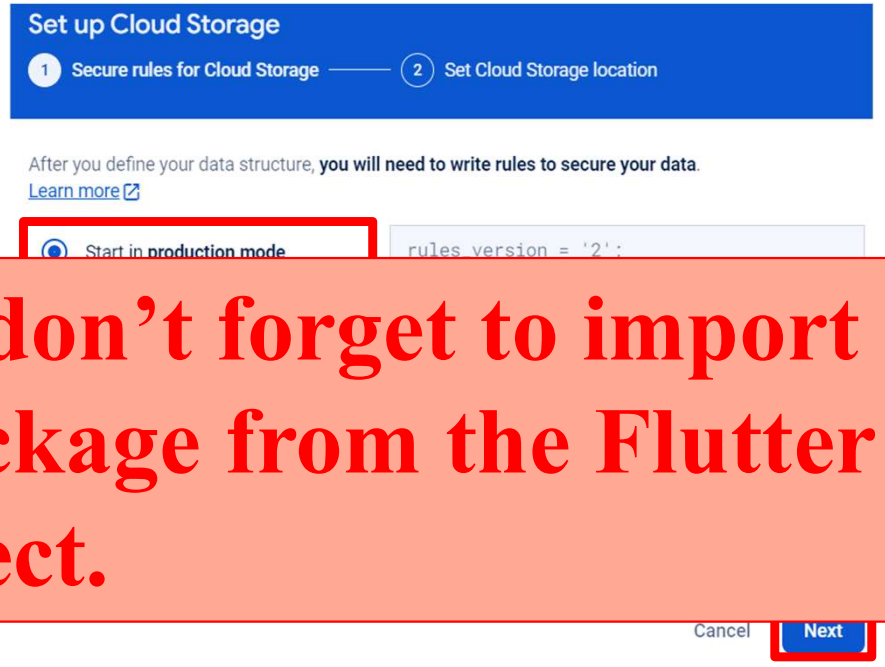




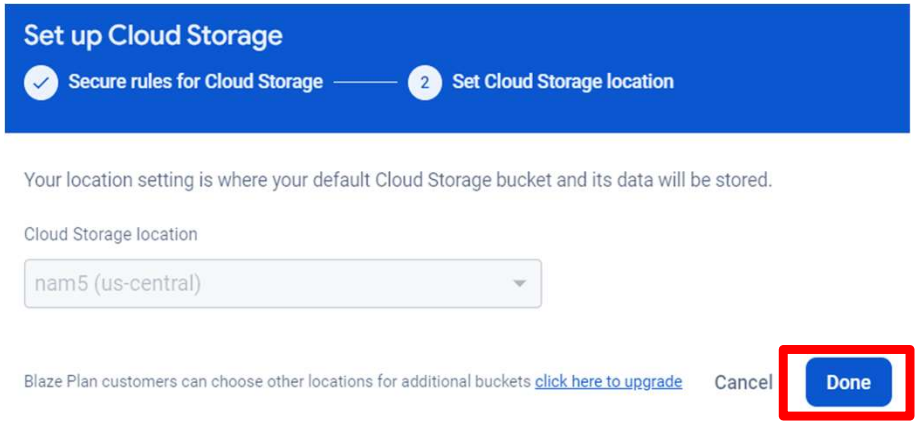


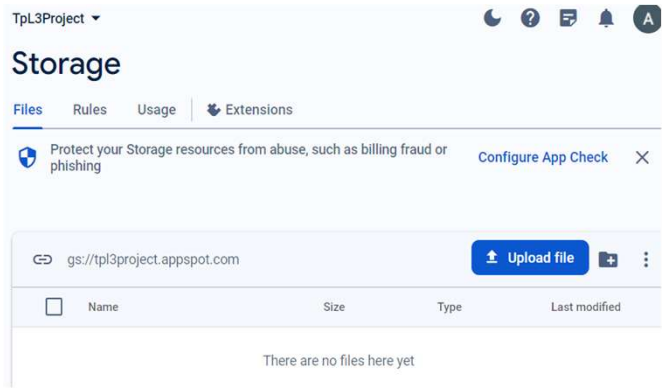
Remote storage techniques: Cloud storage

- **Implementation:** create the Storage from Firebase Console.



After storage creation, don't forget to import the fl_cloud_storage package from the Flutter project.





Remote storage techniques: Cloud storage

- **Implementation:** upload and get a file requires three steps:

```
import 'package:firebase_storage/firebase_storage.dart';
// 1- Get the file reference (storage location + file name)
 final refStorage = FirebaseStorage.instance.ref ('fileName.jpg');

// 2- Upload the file
await refStorage.putFile (myFile!);

// 3- Get the download URL
String myUrl = await refStorage.getDownloadURL ( );

// Example of using the uploaded image
Image.network(myUrl, width : 100, height: 100,);
```

Ideally choose a dynamic name based on the path of the file

```
// Store the file in a folder (images) rather than in the root.
final refStorage = FirebaseStorage.instance.ref ('images/fileName.jpg');
// Or using child()
final refStorage = FirebaseStorage.instance.ref ('images').child('fileName.jpg');
```

Remote storage techniques: Cloud storage

- **Implementation:** add data with a file (image).

```
CollectionReference myPersons = FirebaseFirestore.instance.collection('persons');

// Add data
await myPersons.add({"name": personName, "age": personAge, "profileImg": imgPath,});

// Get data
QuerySnapshot result = await myPersons.get( );

// Display the image of the document at index i
Image.network(result.docs[i]['profileImg'], width : 100, height: 100,);
```

- Delete the file (image) referenced by a data.

```
// Delete the image of the document at index i
FirebaseStorage.instance.refFromURL(result.docs[i]['profileImg']).delete();
```

Conclusion

- **Data persistence** is essential for mobile applications.
- **Local storage** with **Hive**, suitable for offline and lightweight data management.
- **Remote (cloud) storage** using **Firebase**, a scalable Backend-as-a-Service solution.
- Firebase provides a complete ecosystem covering:
  - ✓ **Authentication** (Firebase Auth) for secure user management.
  - ✓ **Firestore** (NoSQL database) for structured cloud data storage.
  - ✓ **Firebase Storage** for handling and storing media files.
- Combining both approaches enables efficient and scalable applications .