Lab session 2: UI Design and Creating Flutter Apps

Task-3 Create an authentication method with Flutter and Dart

In today's digital landscape, user authentication plays a vital role in securing mobile applications. While traditional login forms and authentication methods have been widely used, implementing biometric authentication has become increasingly popular. That's why, we decided to implement Flutter Local Authentication using Biometrics that provides a convenient and secure way for users to access their apps using unique biometric data, such as Face ID or Touch ID/Fingerprint.

Biometric authentication leverages the built-in biometric sensors available on modern mobile devices, such as facial recognition or fingerprint scanning. It provides a more secure authentication method by utilizing unique biometric features that are difficult to replicate or forge.

In this documentation, we will walk through the necessary steps to integrate biometric authentication using Flutter and Dart. We will cover the installation of dependencies, configuration of permissions, implementation of the authentication logic, and integration of biometric authentication into the app.

Getting Started

To integrate local authentication into your Flutter app, we can use a plugin called "local_auth." This plugin uses platform APIs to access the device's hardware securely, ensuring that no private information is leaked.

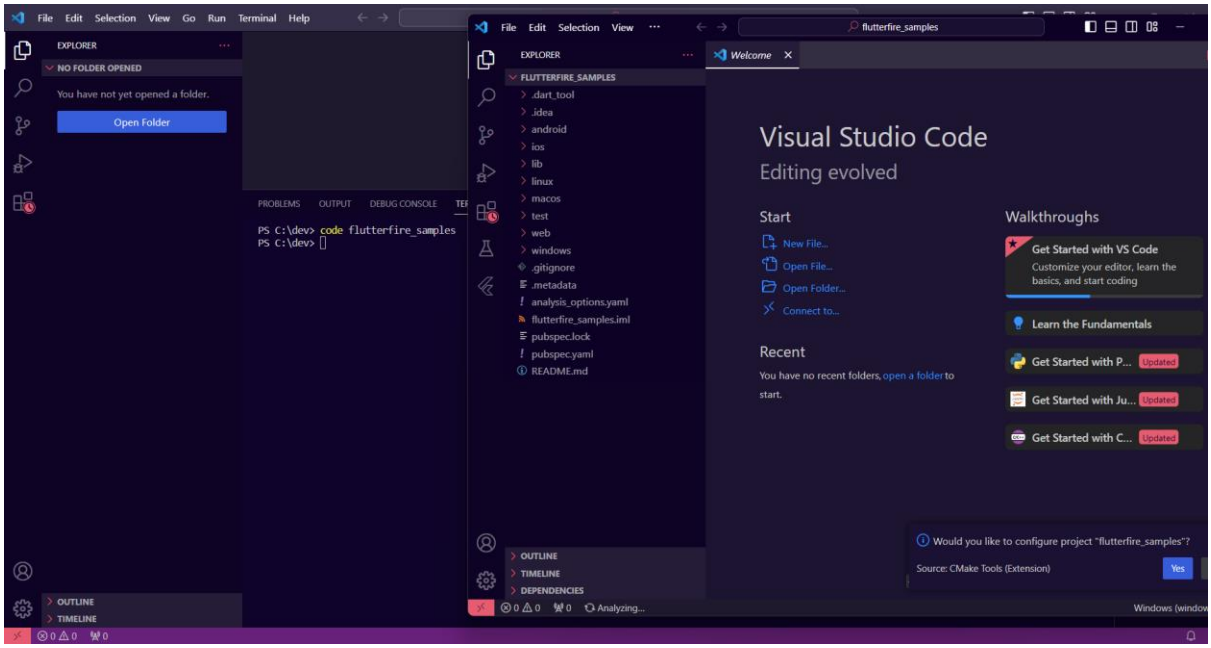
Create project

Step 1: Create a Flutter project by using the following command:

```
PS C:\Users\fyfal> cd ../../dev
PS C:\dev> flutter create flutterfire_samples
Command exited with code 128: git fetch --tags
Standard error: fatal: unable to access 'https://github.com/flutter/flutter.git/': OpenSSL SSL_read: SSL_ERROR_SYSCALL, errno 10054


Creating project flutterfire_samples...
Resolving dependencies in flutterfire_samples... (2.0s)
Got dependencies in flutterfire_samples.
Wrote 129 files.

All done!
You can find general documentation for Flutter at: https://docs.flutter.dev/
Detailed API documentation is available at: https://api.flutter.dev/
If you prefer video documentation, consider: https://www.youtube.com/c/flutterdev

In order to run your application, type:

  $ cd flutterfire_samples
  $ flutter run

Your application code is in flutterfire_samples\lib\main.dart.
```

The standard error is due to a poor internet connection.

Step 2: Open the project in Vs code using the command "code flutterfire_samples". This command will open a new Vs Code window with the project open.



Step 3: Migrate the project to null safety

Null safety is a feature in Dart that helps developers write more robust and predictable code by preventing null reference errors. It helps catch potential issues early in the development process, reducing the likelihood of null reference errors and improving the overall stability of your Dart applications, including those developed with Flutter.

Flutter 2.0 has support for null safety in stable channel, but in order to use it inside the app we have to run a command for migrating the project to null safety.

Before running the migration command, we will check if all the current project dependencies support null safety by using:

```
PS C:\Dev\flutterfire_samples> dart pub outdated --mode=null-safety

    The Dart tool uses Google Analytics to report feature usage statistics
    and to send basic crash reports. This data is used to help improve the
    Dart platform and tools over time.

    To disable reporting of analytics, run:

      dart --disable-analytics


The `--mode=null-safety` option is no longer supported.
Consider using the Dart 2.19 sdk to migrate to null safety.
```

The message suggests that instead of using the deprecated --mode=null-safety option, we should consider using the Dart 2.19 SDK to migrate the project to null safety. This is a recommendation to adopt the newer and more comprehensive null safety features introduced in Dart 2.19.

Since the environment sdk is up to date :

```
EXPLORER                    pubspec.yaml
FLUTTERFIRE_SAMPLES         pubspec.yaml
  .dart_tool
  .idea
  .vscode
  android
  ios
  lib
  linux
  macos
  test
  web
  windows
  .gitignore
  .metadata
  analysis_options.yaml
  flutterfire_samples.iml
  pubspec.lock
  pubspec.yaml
  README.md

4   # pub.dev using `flutter pub publish`. This is preferred for private packages.
5   publish_to: 'none' # Remove this line if you wish to publish to pub.dev
6
7   # The following defines the version and build number for your application.
8   # A version number is three numbers separated by dots, like 1.2.43
9   # followed by an optional build number separated by a +.
10  # Both the version and the builder number may be overridden in flutter
11  # build by specifying --build-name and --build-number, respectively.
12  # In Android, build-name is used as versionName while build-number used as versionCode.
13  # Read more about Android versioning at https://developer.android.com/studio/publish/versioning
14  # In iOS, build-name is used as CFBundleShortVersionString while build-number is used as CFBundleVersion.
15  # Read more about iOS versioning at
16  # https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CoreFoundationKeys.html
17  # In Windows, build-name is used as the major, minor, and patch parts
18  # of the product and file versions while build-number is used as the build suffix.
19  version: 1.0.0+1
20
21  environment:
22    sdk: '>=3.2.3 <4.0.0'
23
24  # Dependencies specify other packages that your package needs in order to work.
25  # To automatically upgrade your package dependencies to the latest versions
```

We can proceed to the migration with the command “dart migrate”:

```
PS C:\Dev\flutterfire_samples> dart migrate
Could not find a command named "migrate".

Usage: dart <command|dart-file> [arguments]

Global options:
-v, --verbose               Show additional command output.
    --version               Print the Dart SDK version.
    --enable-analytics      Enable analytics.
    --disable-analytics     Disable analytics.
    --suppress-analytics    Disallow analytics for this `dart *` run without changing the analytics configuration.
-h, --help                  Print this usage information.

Available commands:
  analyze    Analyze Dart code in a directory.
```

After some researches, we found out that the migrate command is no longer available in the latest version of dart.

```
PS C:\Dev\flutterfire_samples> dart --version
Dart SDK version: 3.2.3 (stable) (Tue Dec 5 17:58:33 2023 +0000) on "windows_x64"
```

Dart 2.19 is the final release that supports null-safety migration, including the dart migrate tool. Dart 3 has built-in sound null safety so there is no need to migrate.

Import packages

Now, we need to import the plugin by adding the following line to the project’s *pubspec.yaml* file:

```
dependencies:
  flutter:
    sdk: flutter

  local_auth: ^2.1.7
```

Working on the UI

The tutorial we are following has started from an existing project so we have chosen to adopt the same approach to save time and build upon an established foundation.

Step 1: We first cloned the project and fix all the dependencies issues.

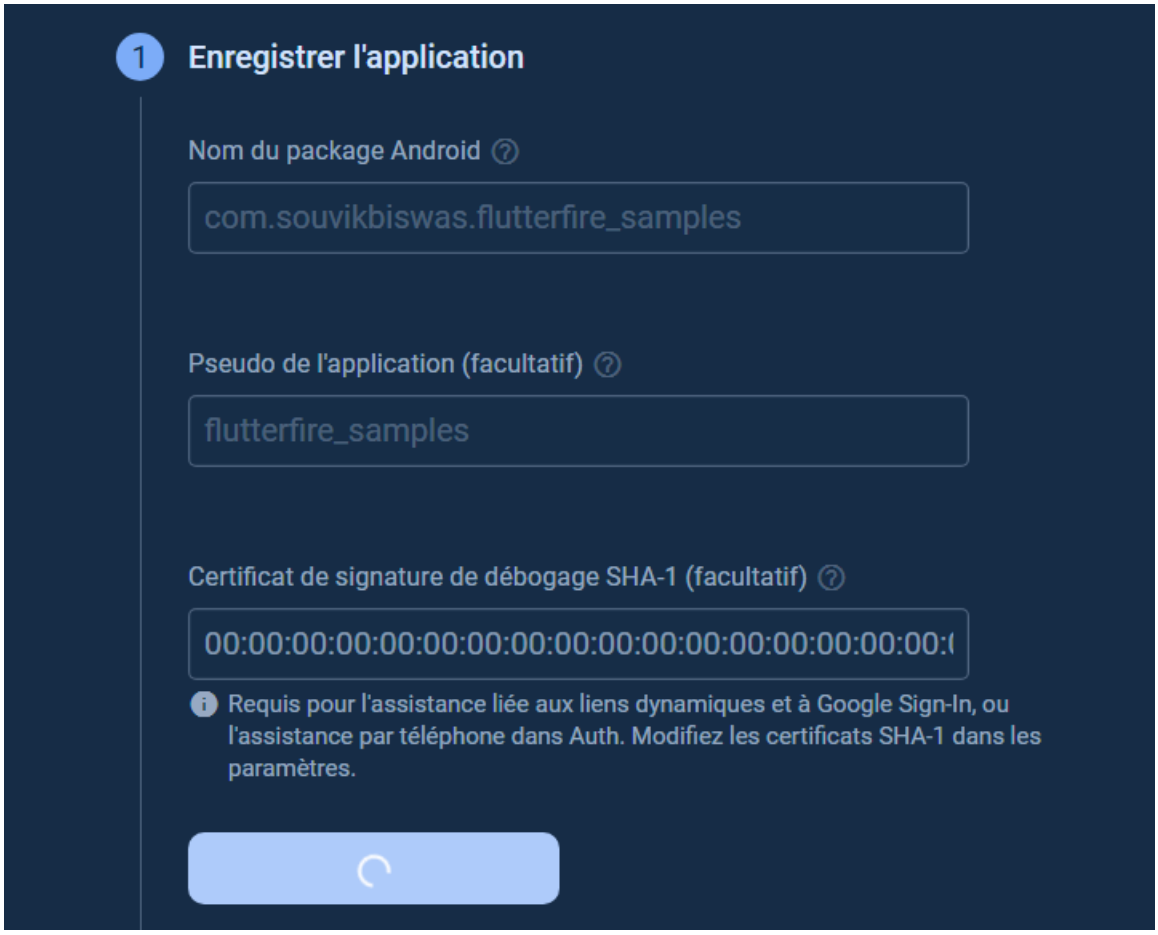
Step 2: Next, we created a new Firebase project from the console.

Step 3: Configure the Firebase for each platform

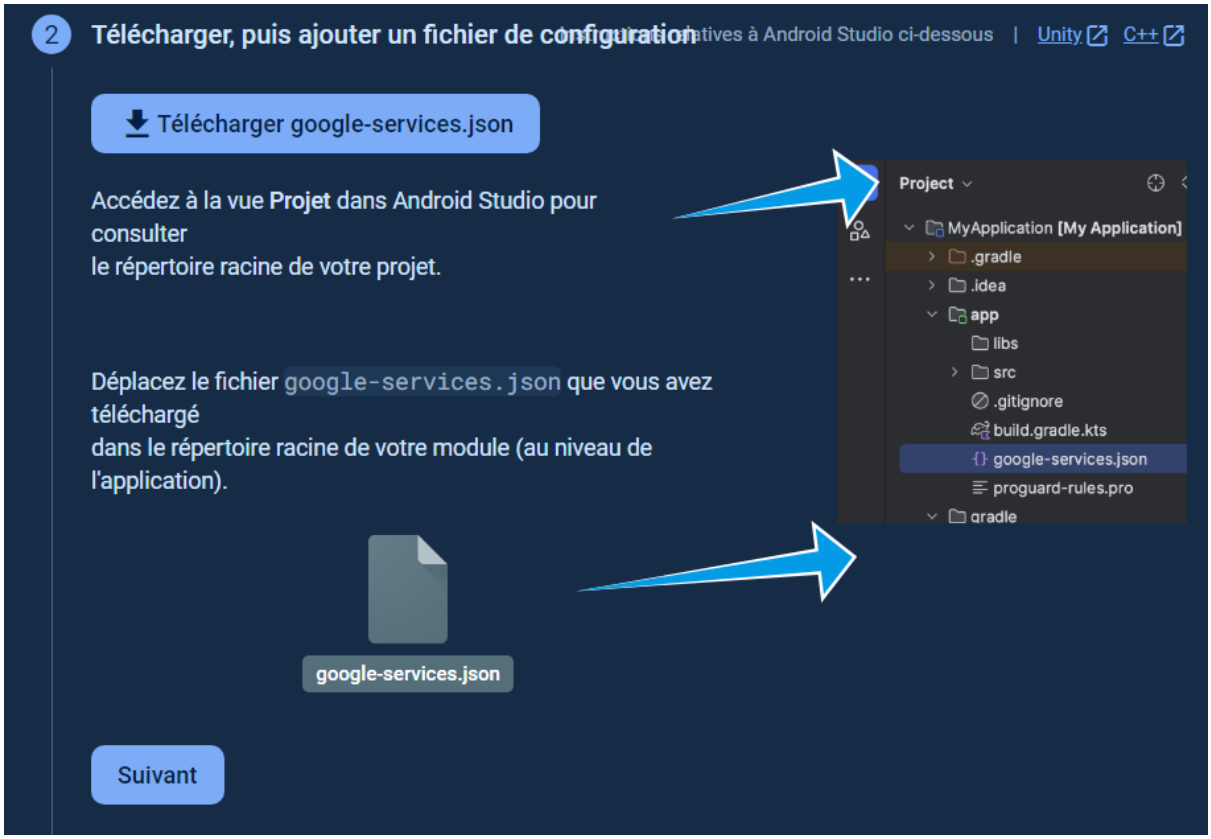
We need to set up Firebase for each platform (Android and iOS) to enable the Firebase services used in the Flutter app.

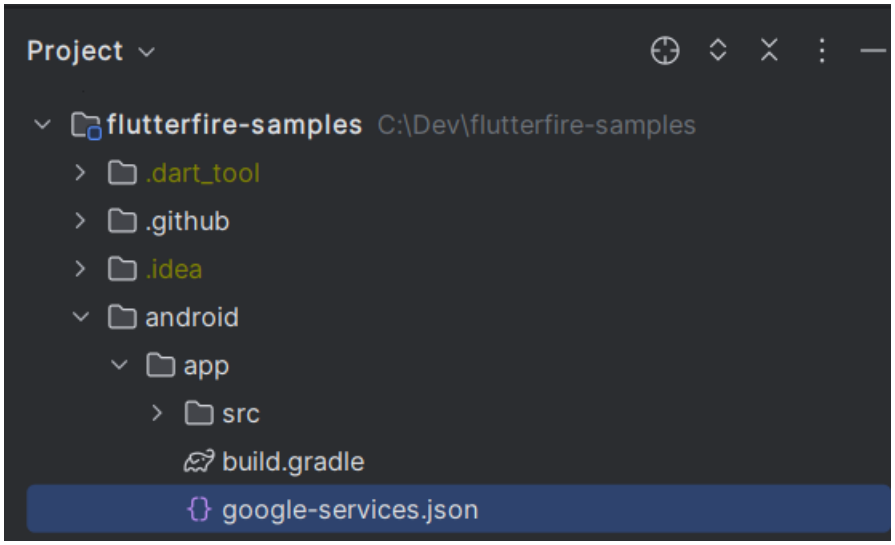
Android:

- In the Firebase Console, click on "Add app" and select the Android platform.
- Follow the instructions to register the app by providing the package name (usually found in the android/app/build.gradle file of the Flutter project) and the app nickname (optional).



- Download the google-services.json file provided by Firebase.
- Move the google-services.json file to the android/app directory of the Flutter project.





- Open the android/build.gradle file in the Flutter project and add the following code at the end of the file:

```
dependencies {
// ...
classpath 'com.google.gms:google-services:4.3.10'
}
```

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    classpath 'com.google.gms:google-services:4.3.10'
}
```

- Open the android/app/build.gradle file in the Flutter project and add the following code at the end of the file, just before the dependencies block:

  apply plugin: 'com.google.gms.google-services'

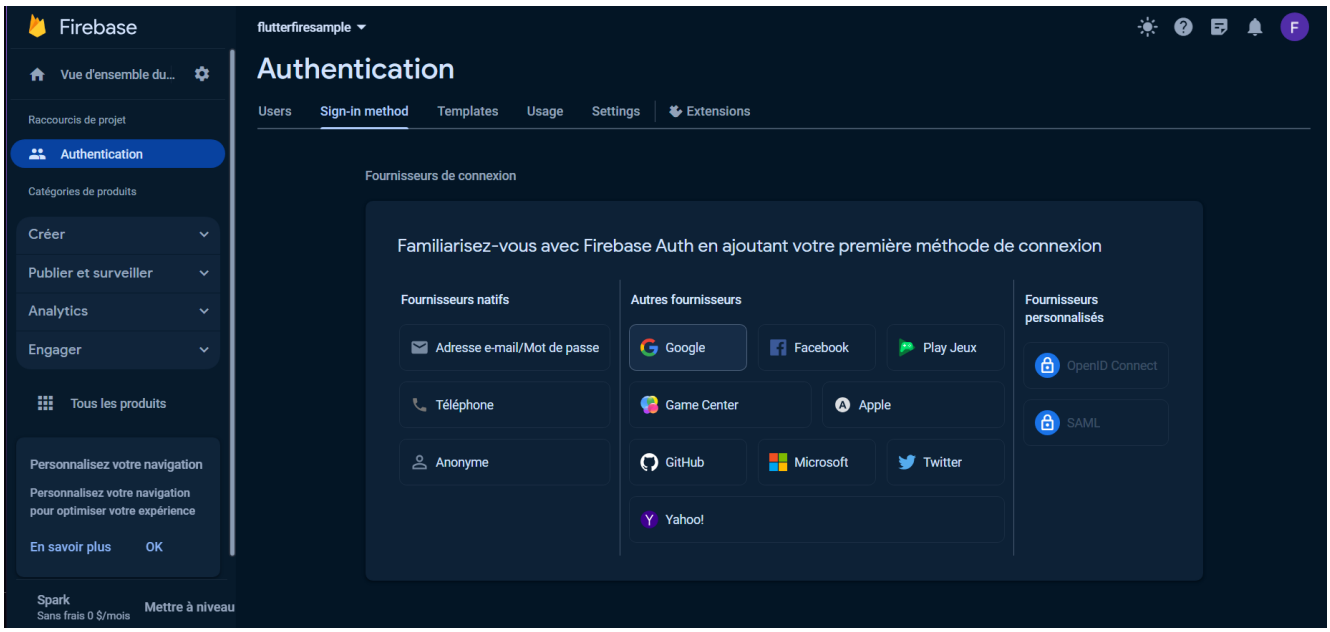
- Sync the project with the Gradle files by clicking on the "Sync Now" button in Android Studio or running the **flutter pub get** command in the terminal.

iOS:

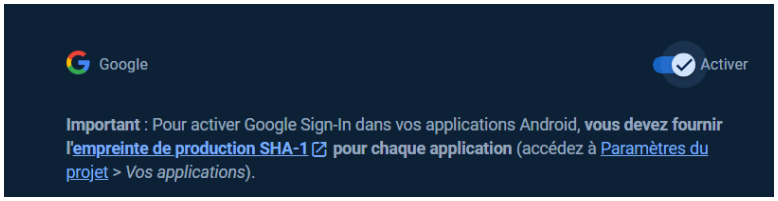
- In the Firebase Console, click on "Add app" and select the iOS platform.
- Follow the instructions to register the app by providing the iOS bundle ID (usually found in the ios/Runner.xcodeproj/project.pbxproj file of your Flutter project) and the app nickname (optional).
- Download the GoogleService-Info.plist file provided by Firebase.
- Move the GoogleService-Info.plist file to the ios/Runner directory of the Flutter project.
- Open your Flutter project in Xcode by running the command open ios/Runner.xcworkspace in the terminal.
- In Xcode, select the "Runner" target, go to the "Signing & Capabilities" tab, and make sure the Apple Developer account is selected and the "Automatically manage signing" option is enabled.
- Run the app on an iOS device or simulator to complete the Firebase configuration for iOS.

Step 4: Set up Google Sign-In

- In the Firebase console, go to the "Authentication" section.



- Enable the "Google" sign-in provider.



- Make sure to provide the appropriate OAuth client ID and secret for Android and iOS platforms, which is obtained from the Google Cloud Console (https://console.cloud.google.com/).
- Configure the necessary scopes and other settings for Google Sign-In according to your requirements.

Step 5: Run the app to test it

Implement local authentication

Currently the project contains two layouts:

- SignInScreen
- UserInfoScreen

This is a simple project for implementing Firebase authentication & Google Sign In to your Flutter app. We will be adding a SecretVaultScreen to this app, which will require biometric authentication to access.

Step 1 : Let's get started by adding a button labeled Access secret vault to the UserInfoScreen that will route to the SecretVaultScreen.

user_info_screen.dart

lib > screens > user_info_screen.dart > _UserInfoScreenState > build

```
122                 color: CustomColors.firebaseGrey.withOpacity(0.8),
123                 fontSize: 14,
124                 letterSpacing: 0.2), // TextStyle
125             ), // Text
126             SizedBox(height: 16.0),
127             ElevatedButton(
128               style: ButtonStyle(
129                 backgroundColor: MaterialStateProperty.all(
130                   CustomColors.firebaseOrange,
131                 ),
132                 shape: MaterialStateProperty.all(
133                   RoundedRectangleBorder(
134                     borderRadius: BorderRadius.circular(10),
135                   ), // RoundedRectangleBorder
136                 ),
137               ), // ButtonStyle
138               onPressed: () async {
139                 bool isAuthenticated =
140                     await Authentication.authenticateWithBiometrics();
141
142                 if (isAuthenticated) {
143                   Navigator.of(context).push(
144                     MaterialPageRoute(
145                       builder: (context) => SecretVaultScreen(),
146                     ), // MaterialPageRoute
147                   );
148                 } else {
149                   ScaffoldMessenger.of(context).showSnackBar(
150                     Authentication.customSnackBar(
151                       content: 'Error authenticating using Biometrics.',
152                     ),
153                   );
154                 }
155               },
156               child: Padding(
157                 padding: EdgeInsets.only(top: 10.0, bottom: 10.0),
158                 child: Text(
```

Step 2: The SecretVaultScreen will contain an icon, a text, and a button (for going back to the previous screen).

secret_vault_screen.dart

lib > screens > secret_vault_screen.dart > ...

```dart
import 'package:flutter/material.dart';
import 'package:flutterfire_samples/res/custom_colors.dart';
import 'package:flutterfire_samples/widgets/app_bar_title.dart';

class SecretVaultScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: CustomColors.firebaseNavy,
      appBar: AppBar(
        elevation: 0,
        leading: Container(),
        backgroundColor: CustomColors.firebaseNavy,
        title: AppBarTitle(),
      ), // AppBar
      body: SafeArea(
        child: Padding(
          padding: const EdgeInsets.only(
            left: 16.0,
            right: 16.0,
            bottom: 20.0,
          ), // EdgeInsets.only
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Row(),
              Icon(
                Icons.lock_open,
                size: 60,
                color: CustomColors.firebaseGrey,
              ), // Icon
              SizedBox(height: 24.0),
              Text(
                'You have successfully accessed the secret vault. Leaving the va
                style: TextStyle(
                  color: CustomColors.firebaseGrey.withOpacity(0.8),
```

Step 3: We need to authenticate the user in the UserInfoScreen before proceeding to the SecretVaultScreen.

We will define a new method in the Authentication class, present in the authentication.dart file, called authenticateWithBiometrics() where the entire logic of biometric authentication will be written.

authentication.dart 1

lib > utils > authentication.dart > ...

```dart
  static Future<bool> authenticateWithBiometrics() async {
    final LocalAuthentication localAuthentication = LocalAuthentication();
    bool isBiometricSupported = await localAuthentication.isDeviceSupported();
    bool canCheckBiometrics = await localAuthentication.canCheckBiometrics;

    bool isAuthenticated = false;

    if (isBiometricSupported && canCheckBiometrics) {
      List<BiometricType> biometricTypes =
          await localAuthentication.getAvailableBiometrics();
      print(biometricTypes);

      isAuthenticated = await localAuthentication.authenticate(
        localizedReason: 'Please complete the biometrics to proceed.',
        biometricOnly: true,
      );
    }

    return isAuthenticated;
  }
}
```

The authenticateWithBiometrics() method will return a boolean indicating whether the biometric authentication is successful.

Step 4: Now, we can update the onPressed() method of the Access secret vault button to use the biometric authentication.

If the authentication is successful then the user will navigate to the SecretVaultScreen, otherwise a SnackBar would be shown with an error message.

Setup for using biometrics

For Android:

Step 1: We need to add this permission to the AndroidManifest.xml file present in the directory android -> app -> src -> main:

```
authentication.dart 1    AndroidManifest.xml ×
android > app > src > main > AndroidManifest.xml
1  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2      package="com.souvikbiswas.flutterfire_samples">
3    <uses-permission android:name="android.permission.USE_FINGERPRINT"/>
```

Step 2: Next, we need to Update the MainActivity.kt file to use FlutterFragmentActivity instead of FlutterActivity:

```
authentication.dart 1    AndroidManifest.xml    MainActivity.kt ×
android > app > src > main > kotlin > com > souvikbiswas > flutterfire_samples > MainActivity.kt
1  package com.souvikbiswas.flutterfire_samples
2
3  import io.flutter.embedding.android.FlutterFragmentActivity
```

Testing the app on Simulator

We were facing difficulties running the app due to version incompatibilities.

Conclusion

In this tutorial, we explored biometric authentication in Dart and gained an understanding of local authentication. We learned several key concepts, including:

- Importing the local_auth package
- Creating an instance of the plugin
- Checking device support for biometric authentication
- Retrieving available biometric types
- Authenticating users using biometrics or pin/passcode

Moreover, as beginners in mobile app programming, we delved into various fundamental aspects of Dart, such as syntax, class creation, and file hierarchy. We also ventured into the realm of third-party tools like Firebase, discovering how to register and synchronize an app with it. Additionally, we gained familiarity with essential command-line operations like using flutter and dart commands, managing package imports, and gaining a basic understanding of working with multiple screens in a mobile app and defining UI elements.

One notable challenge encountered during this tutorial was version incompatibility. Given that you are following a tutorial published three years ago, updates and changes have likely occurred. An example of this is the built-in migration process and the disabled dart migrate command, which reflect the evolution of the language and tools over time.

By navigating these challenges and exploring the concepts presented, we have taken significant strides in understanding biometric authentication in Dart, as well as building a foundation in mobile app development practices and tools.

References

https://blog.codemagic.io/flutter-local-authentication-using-biometrics/

https://console.firebase.google.com/

https://pub.dev/packages/local_auth

https://console.cloud.google.com

https://dart.dev/null-safety#dart-3-and-null-safety