AIM: To connect Flutter UI with firebase database

Theory:

Flutter Connectivity with Firebase: A Comprehensive Guide

Introduction

In today's mobile-centric world, ensuring seamless internet connectivity is crucial for any application. Flutter, a popular framework for cross-platform development, offers robust tools for managing network connections, while Firebase, Google's cloud platform, provides powerful backend services. This note explores how to integrate connectivity management with Firebase in your Flutter app, ensuring a resilient and user-friendly experience even in fluctuating network conditions.

Understanding Network Connectivity in Flutter

1. **Flutter's Connectivity Plugin:**

    - Use the `connectivity` plugin to ascertain connectivity status (WiFi, mobile data, none).
    - Install the plugin: `flutter pub add connectivity`
    - Import and use the `Connectivity` class:

2. Dart

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

Future<void> main() async {
  var connectivityResult = await Connectivity().checkConnectivity();
  if (connectivityResult == ConnectivityResult.mobile) {
    // Use mobile data
  } else if (connectivityResult == ConnectivityResult.wifi) {
    // Use WiFi
  } else {
    // No internet connection
```

```
  }
}
```

3.
4.
5. **Network Change Monitoring:**

    - Subscribe to the `onConnectivityChanged` stream to receive updates on changes:

6. Dart

```
Connectivity().onConnectivityChanged.listen((ConnectivityResult result) {
  if (result == ConnectivityResult.none) {
    // Show an offline message
  }
});
```

7.
8.

Connecting to Firebase

1. **Firebase Setup:**

    - Create a Firebase project and configure authentication and databases as needed.
    - Follow the official guides: https://firebase.google.com/docs/flutter/setup

2.
3. **Firebase Services and Connectivity:**

    - Consider which Firebase services require internet access (e.g., Firestore, Authentication).
    - Handle offline behavior:

        - Local storage (e.g., SharedPreferences, Hive) for caching data.
        - Queueing operations for when connectivity resumes (e.g., using WorkManager).
    -
4.

Best Practices and Considerations

1. **Informative Error Handling:**
    - Provide clear and actionable error messages when offline or experiencing connectivity issues.
    - Use progress indicators to show data loading or syncing progress.
2.
3. **Offline-First Design:**
    - Design your app to function even without internet access (e.g., showing cached data).
    - Prioritize essential features that work offline.
4.
5. **Data Management:**
    - Manage offline data synchronization effectively:
        - Use Firestore's offline persistence and `enablePersistence` feature.
        - Implement conflict resolution strategies for merging local and remote data.
    -
6.

7. **Testing and Monitoring:**
    - Thoroughly test your app's behavior in offline scenarios.
    - Use Firebase Crashlytics or other tools to monitor network-related errors.
8.

Advanced Techniques

1. **Connectivity Plugins:**
    - Explore advanced plugins like `internet_connection_checker` for more granular insights.
2.
3. **Network-Aware Widgets:**
    - Leverage network-aware widgets (e.g., `OfflineIndicator`) for visual feedback.
4.
5. **Custom Offline Caching:**
    - Implement custom caching strategies for specific data types.
6.
7. **Background Tasks:**
    - Use WorkManager or other libraries to perform network-related tasks in the background.
8.

Conclusion

Effectively managing network connectivity in Flutter with Firebase is essential for robust and user-friendly mobile apps. By incorporating the techniques and considerations discussed here, you can ensure that your app remains functional and provides a positive experience even in challenging network conditions.

Additional Notes:

- This guide provides a foundational understanding. Feel free to explore further based on your specific app's requirements.
- The Firebase SDK offers extensive documentation and examples for in-depth guidance.
- Continuously test and refine your connectivity logic for optimal performance and resilience.

Code:

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

import 'package:flutter/material.dart';

import 'package:flutter_application_1/pages/Home.dart';

import 'package:flutter_application_1/pages/bottomnav.dart';

import 'package:flutter_application_1/pages/onboard.dart';

import 'package:flutter_application_1/pages/signup.dart';

import 'package:flutter_application_1/widget/app_constant.dart';

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

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  Stripe.publishableKey = publishableKey;
  try {
    await Firebase.initializeApp(
        options: FirebaseOptions(
            apiKey: 'AIzaSyAbQE5bkqgk7E6PUWUfdjR6bWGX1G7_d68',
            appId: '1:955784621517:android:d83194633c4d1088ea2884',
            messagingSenderId: '955784621517',
            projectId: 'fooddeliveryapp-628c0'));
  } catch (e) {
    print('Error initializing Firebase: $e');
  }
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
```

```
  const MyApp({super.key});


  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        title: 'Flutter Demo',
        theme: ThemeData(
          colorScheme: ColorScheme.fromSeed(
              seedColor: Color.fromARGB(255, 185, 26, 108)),
          useMaterial3: true,
        ),


        home: Onboard());
  }
}
class MyHomePage extends StatefulWidget {
 const MyHomePage({super.key, required this.title});
```

```
  final String title;

  @override

  State<MyHomePage> createState() => _MyHomePageState();

}



class _MyHomePageState extends State<MyHomePage> {

  int _counter = 0;



  void _incrementCounter() {

    setState(() {

      // This call to setState tells the Flutter framework that something has

      // changed in this State, which causes it to rerun the build method below

      // so that the display can reflect the updated values. If we changed

      // _counter without calling setState(), then the build method would not be

      // called again, and so nothing would appear to happen.

      _counter++;

    });

  }
```

```
@override
Widget build(BuildContext context) {
  // This method is rerun every time setState is called, for instance as done
  // by the _incrementCounter method above.
  //
  // The Flutter framework has been optimized to make rerunning build methods
  // fast, so that you can just rebuild anything that needs updating rather
  // than having to individually change instances of widgets.
  return Scaffold(
    appBar: AppBar(


      ),
    body: Center(
      // Center is a layout widget. It takes a single child and positions it
      // in the middle of the parent.
      child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
```

```
        children: <Widget>[
          const Text(
            'You have pushed the button this many times:',
          ),
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headlineMedium,
          ),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  );
}
```

}

Screenshots: