

# **Documentation: Flutter Application with Firebase Integration**

## *A Demonstration of a Restaurant Management App with App State Monitoring*

By : Ahmed Mohammed Mourad

Eng/ Ayman Hegazy

### **I. Introduction**

The Restaurant Management App is designed to streamline restaurant operations through a digital platform. Built using Flutter for a cross-platform experience, it integrates Firebase to manage authentication, real-time data, and app monitoring effectively. The purpose of this application is to enhance operational efficiency and provide a user-friendly interface for restaurant staff and customers.

### **II. Technology Stack**

**Flutter Framework:** Utilized for developing cross-platform mobile applications.

**Firebase Services:**

- Authentication
- Firestore Database
- Cloud Messaging
- Analytics and Monitoring

### **III. Application Features**

- **User Roles:** Separate functionalities for admins, staff, and customers.
- **Menu Management:** Add, edit, and organize menu items.
- **Order Tracking:** Real-time updates on order status.
- **App State Monitoring:** Firebase integration to monitor and analyze app performance.

#### IV. Architecture

- **App Architecture:** Follows the MVVM (Model-View-ViewModel) design pattern.
- **Firebase Integration:** Centralized backend service for real-time synchronization.
- **State Management:** Utilizes Provider or Riverpod for state management.

Include a diagram of the architecture.

#### V. Firebase Integration

**Authentication:** Secure login and registration process for users.

**Firestore Database:** Efficient storage and retrieval of menu and order data.

**Cloud Messaging:** Notifications for order updates.

**Analytics:** Monitoring app usage and crash analytics.

#### VI. User Interface

- **Key UI Components:**
  - Login Screen
  - Dashboard
  - Menu Management Screen
  - Order Details Screen

Include screenshots or mockups of key UI components.

#### VII. Testing and Debugging

- **Testing Methodologies:** Unit testing, integration testing, and user acceptance testing.
- **Debugging Challenges:** Addressed issues with real-time updates and Firebase integration.

## VIII. Reusable Components

### *Error Code Translation Function*

#### **Purpose**

The function aims to enhance user experience by translating technical error codes into clear, readable messages that help users understand what went wrong and guide them on how to resolve the issue.

#### **Key Features**

- **Input:** Accepts an error code as a string (errorCode).
- **Logic:**
  - Uses a switch statement to match the input error code against predefined cases.
  - Each case corresponds to a specific error, such as invalid email format or incorrect password.
  - Returns a descriptive error message tailored to the specific issue.
- **Output:** Returns a user-friendly string message corresponding to the error code.
- **Default Case:** Includes a fallback message for unknown or unspecified error codes.

#### **Example Error Messages**

- 'invalid-email': Indicates an incorrectly formatted email, with a message like "The email address is not valid."
- 'user-disabled': Alerts the user that the account is disabled and suggests contacting support.
- 'wrong-password': Provides feedback for incorrect password entries and encourages retrying.
- 'email-already-in-use': Guides users to log in if their email is already registered.
- 'network-request-failed': Advises users to check their internet connection if a network error occurs.

#### **Benefits**

- **User Guidance:** Helps users understand errors without technical jargon.
- **Consistency:** Ensures error messages are standardized across the application.
- **Ease of Debugging:** Facilitates faster resolution by linking user actions to specific error codes.

This function is especially useful in applications with authentication workflows, such as login and registration processes, where clear error handling is crucial for a smooth user experience.

```
auth_error_messages.dart x
Dart SDK is not configured

1 String getReadableAuthError(String errorCode) {
2   switch (errorCode) {
3     case 'invalid-email':
4       return 'The email address is not valid';
5     case 'user-disabled':
6       return 'This account has been disabled. Please contact support';
7     case 'user-not-found':
8       return 'No account exists with this email. Please check your email or sign up';
9     case 'wrong-password':
10      return 'The password you entered is incorrect. Please try again';
11     case 'email-already-in-use':
12      return 'This email is already registered. Please try logging in instead';
13     case 'operation-not-allowed':
14      return 'This operation is not allowed. Please contact support';
15     case 'weak-password':
16      return 'Your password is too weak. Please use at least 6 characters with a mix of letters and n
17     case 'too-many-requests':
18      return 'Access temporarily blocked due to many failed attempts. Please try again later';
19     case 'network-request-failed':
20      return 'Unable to connect. Please check your internet connection and try again';
21     case 'invalid-credential':
22      return 'Invalid email or password. Please check your credentials and try again';
23     case 'invalid-verification-code':
24      return 'Invalid verification code. Please try again';
25     case 'invalid-verification-id':
26      return 'Invalid verification. Please restart the verification process';
27     case 'requires-recent-login':
28      return 'This operation is sensitive and requires recent authentication. Please log in again';
29     default:
30      return 'An error occurred. Please check your information and try again';
```

## ***CustomButton Class Documentation***

### **Purpose**

The CustomButton class is a reusable, customizable button widget that builds upon Flutter's ElevatedButton. It provides a consistent design and simplifies button implementation in the application.

### **Key Features**

#### **Class Declaration:**

- Inherits from StatelessWidget, indicating that the button's appearance and behavior do not change dynamically.
- Accepts two required parameters:
  - onPressed: A callback function triggered when the button is pressed.
  - child: A widget displayed as the button's content, such as text or an icon.

#### **Constructor:**

- Includes named parameters with the required keyword for onPressed and child to ensure they are provided when the widget is used.
- Uses super.key to pass the widget's key to its parent class for identification.

#### **build Method:**

- Defines the visual appearance of the button using ElevatedButton.
- Customizes the button's style with the ElevatedButton.styleFrom method:
  - backgroundColor: Dynamically uses the primary color from the current theme.
  - foregroundColor: Sets the text or icon color to white for readability.
  - minimumSize: Ensures the button spans the full width of its container and has a fixed height of 56 pixels.
  - shape: Gives the button rounded corners with a border radius of 12 pixels.
  - elevation: Sets the elevation to 0 for a flat, modern design.
- The child widget defines the button's inner content.

custom\_button.dart ×

Dart SDK is not configured

```
1  import 'package:flutter/material.dart';
2
3  class CustomButton extends StatelessWidget {
4    final VoidCallback? onPressed;
5    final Widget child;
6
7    const CustomButton({
8      super.key,
9      required this.onPressed,
10     required this.child,
11   });
12
13   @override
14   Widget build(BuildContext context) {
15     return ElevatedButton(
16       onPressed: onPressed,
17       style: ElevatedButton.styleFrom(
18         backgroundColor: Theme.of(context).colorScheme.primary,
19         foregroundColor: Colors.white,
20         minimumSize: const Size(double.infinity, 56),
21         shape: RoundedRectangleBorder(
22           borderRadius: BorderRadius.circular(12),
23         ),
24         elevation: 0,
25       ),
26       child: child,
27     );
28   }
29 }
```

## **CustomTextField Widget**

### **Purpose**

The CustomTextField widget is a reusable text input field designed to handle various use cases, including password fields, text validation, and different keyboard input types. It provides additional functionality such as toggleable visibility for obscured text, making it ideal for secure or dynamic input scenarios.

### **Key Features**

#### **Class Declaration:**

- **CustomTextField** extends StatefulWidget, making it dynamic and capable of managing internal states (e.g., toggling text visibility).
- It accepts several customizable properties:
  - controller: A TextEditingController to manage and retrieve the text input.
  - label: A string that displays as the field's label.
  - obscureText: A boolean that determines if the input text should be hidden (e.g., for passwords).
  - keyboardType: Specifies the type of keyboard (e.g., numeric, email, etc.).
  - validator: A function to validate the text input when used in forms.

#### **Internal State:**

- The widget maintains a private `_obscureText` variable to toggle text visibility for obscured fields.

#### **Lifecycle Method:**

- **initState:** Initializes `_obscureText` with the value of `widget.obscureText`.

#### **Build Method:**

- The build method uses `TextFormField` to provide a customizable text input field with dynamic behavior and styling.

- **Controller Binding:** The TextFormField is linked to the provided controller.
- **Text Obscurity:** The obscureText property is dynamically set based on the \_obscureText state.
- **Validation:** Custom validation logic can be added through the validator function.

## Decoration:

- Displays a label using widget.label.
- Adds a suffix icon (eye icon) for password fields to toggle visibility.

```
custom_text_field.dart ×  
Dart SDK is not configured  
1 import 'package:flutter/material.dart';  
2   
3 class CustomTextField extends StatefulWidget {  
4   final TextEditingController controller;  
5   final String label;  
6   final bool obscureText;  
7   final TextInputType? keyboardType;  
8   final String? Function(String?)? validator;  
9   
10  const CustomTextField({  
11    super.key,  
12    required this.controller,  
13    required this.label,  
14    this.obscureText = false,  
15    this.keyboardType,  
16    this.validator,  
17  });  
18   
19  @override  
20  State<CustomTextField> createState() => _CustomTextFieldState();  
21 }  
22   
23 class _CustomTextFieldState extends State<CustomTextField> {  
24   late bool _obscureText;  
25   
26   @override  
27   void initState() {  
28     super.initState();  
29     _obscureText = widget.obscureText;  
30   }
```



## *DefaultFirebaseOptions Class*

The file defines the `DefaultFirebaseOptions` class, which contains platform-specific Firebase configurations. It enables seamless initialization of Firebase in a Flutter app based on the platform on which the app runs (e.g., Android, iOS).

### **Key Features**

#### **Class Definition:**

- **DefaultFirebaseOptions:** A utility class that holds Firebase configuration details for different platforms (e.g., Android, iOS, etc.).

#### **Platform-Based Configuration:**

- **currentPlatform:** A getter that determines the target platform at runtime (using `kIsWeb` or `defaultTargetPlatform`) and returns the corresponding `FirebaseOptions` instance.
- Throws an `UnsupportedError` for platforms not yet configured in the file (e.g., iOS, macOS, web, etc.).

#### **Android Configuration:**

Contains Firebase configuration details specific to Android:

- **apiKey:** Authentication key for Firebase API access.
- **appId:** Unique identifier for the Firebase app.
- **messagingSenderId:** Identifier for Firebase Cloud Messaging.
- **projectId:** Identifier for the Firebase project.
- **databaseURL:** URL for the Firebase Realtime Database.
- **storageBucket:** URL for Firebase Cloud Storage.

#### **Unsupported Platforms:**

For platforms like iOS, macOS, Linux, Windows, and web, the code currently throws an `UnsupportedError`. To enable these platforms, reconfiguration using the FlutterFire CLI is required.

```
1 > ../../
3 > import ...
6
7 /// Default [FirebaseOptions] for use with your Firebase apps.
8 ///
9 /// Example:
10 /// ```dart
11 /// import 'firebase_options.dart';
12 /// // ...
13 /// await Firebase.initializeApp(
14 ///   options: DefaultFirebaseOptions.currentPlatform,
15 /// );
16 /// ```
17 class DefaultFirebaseOptions {
18   static FirebaseOptions get currentPlatform {
19     if (kIsWeb) {
20       throw UnsupportedError(
21         'DefaultFirebaseOptions have not been configured for web - '
22         'you can reconfigure this by running the FlutterFire CLI again.',
23       );
24     }
25     switch (defaultTargetPlatform) {
26       case TargetPlatform.android:
27         return android;
28       case TargetPlatform.iOS:
29         throw UnsupportedError(
30           'DefaultFirebaseOptions have not been configured for ios - '
31           'you can reconfigure this by running the FlutterFire CLI again.',
32         );
33       case TargetPlatform.macOS:
```

## *Kitchen Management System*

The application serves as a kitchen management system that integrates Firebase for authentication and database functionalities. It includes key components such as the main entry point (KitchenApp), fallback error handling (ErrorApp), and a theme configuration to maintain a consistent user interface design.

### Key Components

#### Firestore Initialization

- **Purpose:** Ensures Firestore services are initialized before launching the app.
- **Configuration:** Uses `DefaultFirestoreOptions.currentPlatform` to handle platform-specific configurations.
- **Error Handling:** If initialization fails, the app launches the ErrorApp with a retry option to attempt re-initialization.

#### Routing

- **App Routes:** Defined within the MaterialApp widget.
- **Authentication State Management:** Utilizes StreamBuilder to listen for Firestore authentication state changes:
  - If the user is logged in, navigates to the KitchenOrdersScreen.
  - If the user is not authenticated, redirects to the LoginScreen.

#### Theming

- **Custom Theme:** The app uses the ThemeData class to ensure consistent design across the app.
  - **Color Scheme:**
    - Primary color: Dark blue (0xFF2B2D42).
    - Secondary color: Light grey-blue (0xFF8D99AE).
    - Tertiary color: Black.
  - **Card Theme:**
    - Elevation: 4.
    - Margin: Horizontal 16px, vertical 8px.
    - Rounded corners: 16px radius.
  - **Input Decoration Theme:**
    - Custom borders for text fields with different styles for focused, enabled, and error states.
    - Padding: Symmetric horizontal and vertical padding.
  - **Button Themes:**

- Elevated buttons: Flat, rounded corners with vertical padding.
- Text buttons: Padding and rounded corners.

## Error Handling

- **ErrorApp:** A fallback UI displayed when Firebase initialization fails.
  - Displays an error icon and message.
  - Includes a "Retry" button that re-runs the main method to attempt reinitialization.

## Screens

- **LoginScreen:** A placeholder for the user login interface.
- **KitchenOrdersScreen:** Displays orders for the kitchen staff, shown when the user is authenticated.

## Benefits

- **Error Resilience:** Provides fallback handling to inform users of any initialization issues.
- **Dynamic Routing:** Automatically navigates users based on their authentication state.
- **Consistent UI:** Applies a cohesive theme, offering a polished appearance throughout the app.
- **Firebase Integration:** Simplifies backend operations by utilizing Firebase for authentication and other services.

## Improvements

- **Platform Configurations:** Add support for additional platforms, such as iOS and web.
- **Enhanced Error Handling:** Expand error handling in the ErrorApp to log errors or notify support teams.
- **Localization:** Implement text localization for multi-language support.

main.dart ×

Dart SDK is not configured

```
1  > import ...
7
8  void main() async {
9    try {
10     WidgetsFlutterBinding.ensureInitialized();
11     await Firebase.initializeApp(
12       options: DefaultFirebaseOptions.currentPlatform,
13     );
14     runApp(const KitchenApp());
15   } catch (e) {
16     print('Error initializing Firebase: $e');
17     runApp(const ErrorApp()); // Fallback app in case of initialization error
18   }
19 }
20
21 class KitchenApp extends StatelessWidget {
22   const KitchenApp({super.key});
23
24   @override
25   Widget build(BuildContext context) {
26     return MaterialApp(
27       debugShowCheckedModeBanner: false,
28       initialRoute: '/',
29       routes: {
30         '/': (context) => StreamBuilder<User?>(
31           stream: FirebaseAuth.instance.authStateChanges(),
32           builder: (context, snapshot) {
33             if (snapshot.connectionState == ConnectionState.waiting) {
34               return const Center(child: CircularProgressIndicator());
35             }
36           }
37         )
38       }
39     );
40   }
41 }
```

## *Flutter StatefulWidget for Kitchen Order Management*

The **KitchenOrdersScreen** is a **Flutter StatefulWidget** designed for managing and displaying kitchen orders in a restaurant system. It integrates with **Firestore Realtime Database** for real-time order updates and includes functionality for marking orders as complete and signing out.

### **Purpose**

This screen is designed to:

- Fetch and display active kitchen orders in real-time.
- Allow kitchen staff to mark orders as completed.
- Provide a user-friendly UI for order management with expandable order details.

### **Key Features**

#### **Real-Time Order Updates**

- Utilizes **Firestore Realtime Database** to fetch and listen for changes in the orders node.
- Orders are filtered to include only those marked as enabled and are sorted by time (newest first).
- Updates the UI dynamically whenever the database changes.

#### **Order Completion**

- Includes a method (**\_completeOrder**) to update the database, marking the order as disabled when completed.
- Identifies orders by **tableNumber** and updates the corresponding entry in the database.

#### **Sign-Out Functionality**

- Provides a **sign-out button** in the app bar.
- Displays a confirmation dialog to avoid accidental sign-outs.
- Redirects the user to the **login screen** upon successful sign-out.

#### **Expandable Order Details**

- Each order is displayed as an expandable card.

- The expansion reveals details such as:
  - Ordered items and their quantities.
  - Total price.
- Includes a "**Complete Order**" button for marking the order as processed.

## Error Handling

- Gracefully handles **null values** and unexpected data formats in the database.
- Displays a "**No active orders**" message when there are no orders.

## Theming

- The screen adopts a **custom theme** from the app's global theme, ensuring consistency in design.
- Uses **rounded corners** and **subtle colors** for a modern, professional appearance.

## Workflow

### Initialization

- On widget creation, **\_listenToOrders** is called to start listening for order changes in the database.

### Fetching Orders

- Listens to **onValue** events on the **orders node**.
- Filters and processes active orders (**enabled == true**).
- Sorts orders by timestamp (descending).

### UI Rendering

- Displays a list of orders using **ListView.builder**.
- Each order is rendered as a card with an **ExpansionTile** for detailed view.

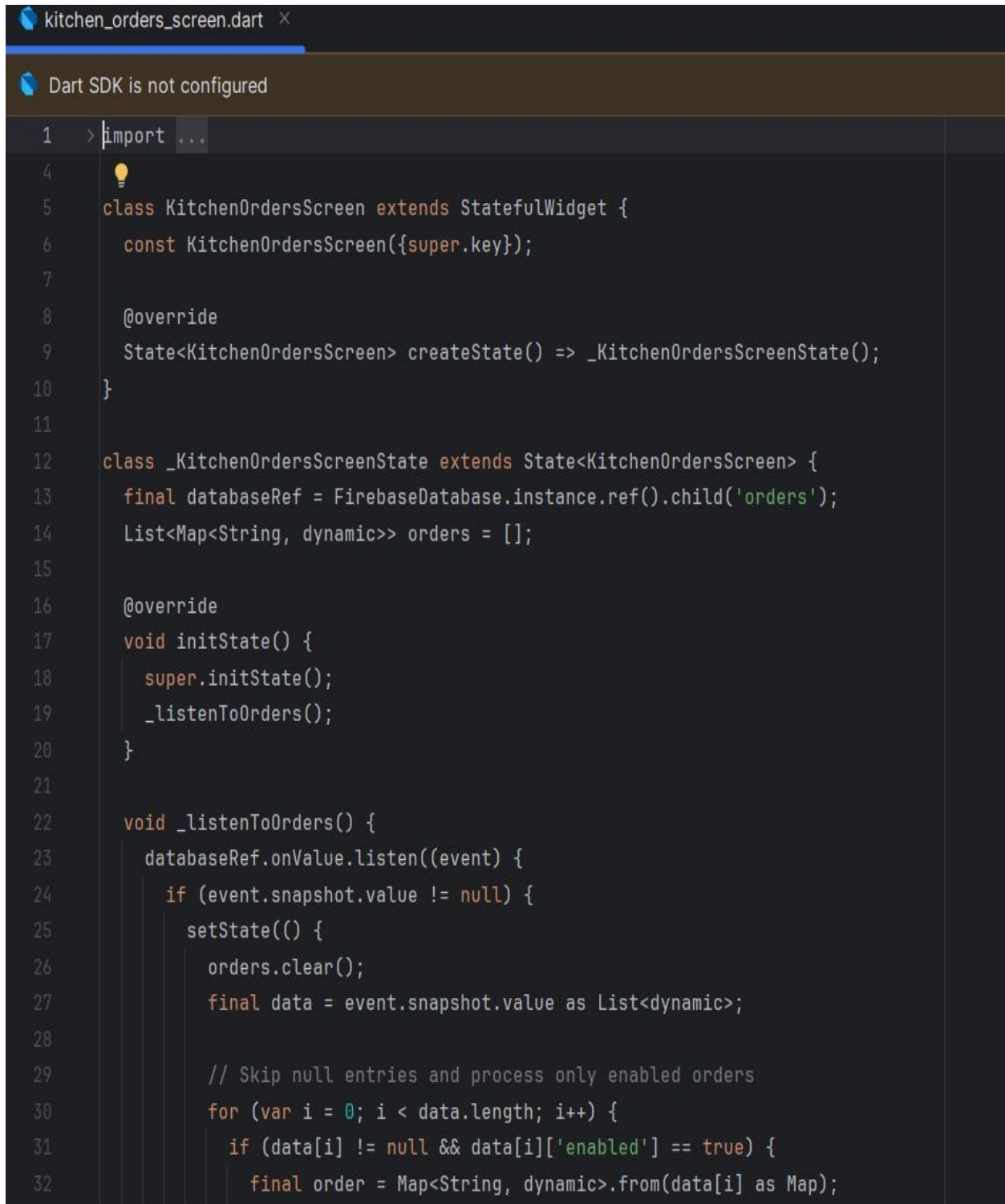
### Completing an Order

- Updates the corresponding order's **enabled** field to **false** in the database.

### Sign-Out

- Logs out the user using **FirebaseAuth.signOut**.

- Navigates back to the **login screen**.



```
kitchen_orders_screen.dart x
Dart SDK is not configured

1  > import ...
4  ⚡
5  class KitchenOrdersScreen extends StatefulWidget {
6    const KitchenOrdersScreen({super.key});
7
8    @override
9    State<KitchenOrdersScreen> createState() => _KitchenOrdersScreenState();
10 }
11
12 class _KitchenOrdersScreenState extends State<KitchenOrdersScreen> {
13   final databaseRef = FirebaseDatabase.instance.ref().child('orders');
14   List<Map<String, dynamic>> orders = [];
15
16   @override
17   void initState() {
18     super.initState();
19     _listenToOrders();
20   }
21
22   void _listenToOrders() {
23     databaseRef.onValue.listen((event) {
24       if (event.snapshot.value != null) {
25         setState(() {
26           orders.clear();
27           final data = event.snapshot.value as List<dynamic>;
28
29           // Skip null entries and process only enabled orders
30           for (var i = 0; i < data.length; i++) {
31             if (data[i] != null && data[i]['enabled'] == true) {
32               final order = Map<String, dynamic>.from(data[i] as Map);
```



## ***Flutter StatefulWidget for Firebase Authentication***

The LoginScreen is a Flutter StatefulWidget designed to manage user login with Firebase Authentication. It integrates email verification checks, error handling, and a smooth user interface with custom components. Below is a detailed breakdown of its functionality and implementation:

### **Purpose**

- Provide a login interface for users.
- Authenticate users using Firebase Authentication.
- Ensure email verification before granting access.
- Navigate to the main app (e.g., kitchen orders screen) after successful login.

### **Key Features**

#### **Firebase Authentication:**

- Authenticates users using `signInWithEmailAndPassword`.
- Validates email verification before granting access.

#### **Email Verification Handling:**

- Prompts the user to verify their email if it has not already been verified.
- Allows users to resend the verification email via a dialog.

#### **Customizable Error Messaging:**

- Maps Firebase Authentication errors to user-friendly messages using `getReadableAuthError`.

#### **Reusable UI Components:**

- Uses `CustomTextField` for input fields and `CustomButton` for the login button.
- Ensures consistent theming across the app.

#### **Responsive UI:**

- Adapts to different screen sizes using `SafeArea` and `SingleChildScrollView`.

#### **Error Feedback:**

- Displays a styled error message when login fails.

- Highlights issues like missing email/password or Firebase errors.

## Workflow

### Input Validation:

- Ensures email and password fields are not empty using validators.

## Login Attempt:

- Initiates login via Firebase Authentication.
- Displays a loading indicator during the login process.

### Email Verification Check:

- If the email is not verified, signs out the user and shows a dialog with options to resend the verification email.

## Navigation:

- Redirects to the KitchenOrdersScreen upon successful login and email verification.

### Error Handling:

- Catches Firebase-specific errors and unexpected issues.
- Updates the UI with a user-friendly error message.

```
login_screen.dart
Dart SDK is not configured

1  > import ...
8
9  ~ class LoginScreen extends StatefulWidget {
10     const LoginScreen({super.key});
11
12     @override
13     State<LoginScreen> createState() => _LoginScreenState();
14 }
15
16 ~ class _LoginScreenState extends State<LoginScreen> {
17     final _formKey = GlobalKey<FormState>();
18     final _emailController = TextEditingController();
19     final _passwordController = TextEditingController();
20     bool _isLoading = false;
21     String? _errorMessage;
22
23     Future<void> _login() async {
24         if (!_formKey.currentState!.validate()) return;
25
26         ~ setState(() {
27             _isLoading = true;
28             _errorMessage = null;
29         });
30
31         try {
32             // Sign in user
33             UserCredential userCredential =
34             ~ await FirebaseAuth.instance.signInWithEmailAndPassword(
35                 email: _emailController.text.trim(),
36                 password: _passwordController.text
```

## ***User Account Creation with Firebase Authentication***

The SignupScreen is a Flutter StatefulWidget designed for creating a new user account using Firebase Authentication. It ensures proper input validation, provides clear user feedback, and handles email verification. Below is a detailed breakdown of its implementation:

### **Purpose**

- **Enable user registration:** Allows users to create accounts using email and password.
- **Input validation:** Verifies that the input fields are filled and passwords match.
- **Email verification:** Sends a verification email after successful signup and guides users to verify their email before logging in.

### **Key Features**

#### **Firebase Integration:**

- Utilizes createUserWithEmailAndPassword to register users.
- Sends email verification links through Firebase.

#### **Form Validation:**

- Ensures the email field is not empty.
- Verifies that the password has at least 6 characters.
- Confirms that both password fields match.

#### **Error Feedback:**

- Displays user-friendly error messages for Firebase Authentication errors via getReadableAuthError.
- Handles unexpected errors gracefully and informs the user.

#### **Reusable UI Components:**

- Uses CustomTextField and CustomButton to maintain a consistent design throughout the app.

#### **User Guidance:**

- Informs users about the email verification process.

- Redirects users to the login screen after signup completion.

### **Responsive Design:**

- Adapts seamlessly to different screen sizes using SafeArea and SingleChildScrollView.

### **Workflow**

#### **Input Validation:**

- Verifies that the email and password fields are filled.
- Checks that the password meets the minimum length requirement and matches the confirmation password.

#### **Signup Process:**

- Creates a new account using the provided email and password.
- Sends a verification email to the user's inbox.
- Displays a success dialog, prompting users to verify their email.

#### **Error Handling:**

- Catches Firebase-specific errors such as "email already in use" and displays relevant error messages.

#### **Navigation:**

- Redirects the user to the login screen upon successful signup.

### **Improvements**

#### **Enhanced Password Validation:**

- Add checks for a mix of uppercase and lowercase letters, numbers, and special characters to strengthen password security.

#### **Accessibility:**

- Provide semantic labels for input fields and buttons to enhance support for screen readers.

#### **User Experience:**

- Display tooltips or hints next to the password field to indicate password requirements.

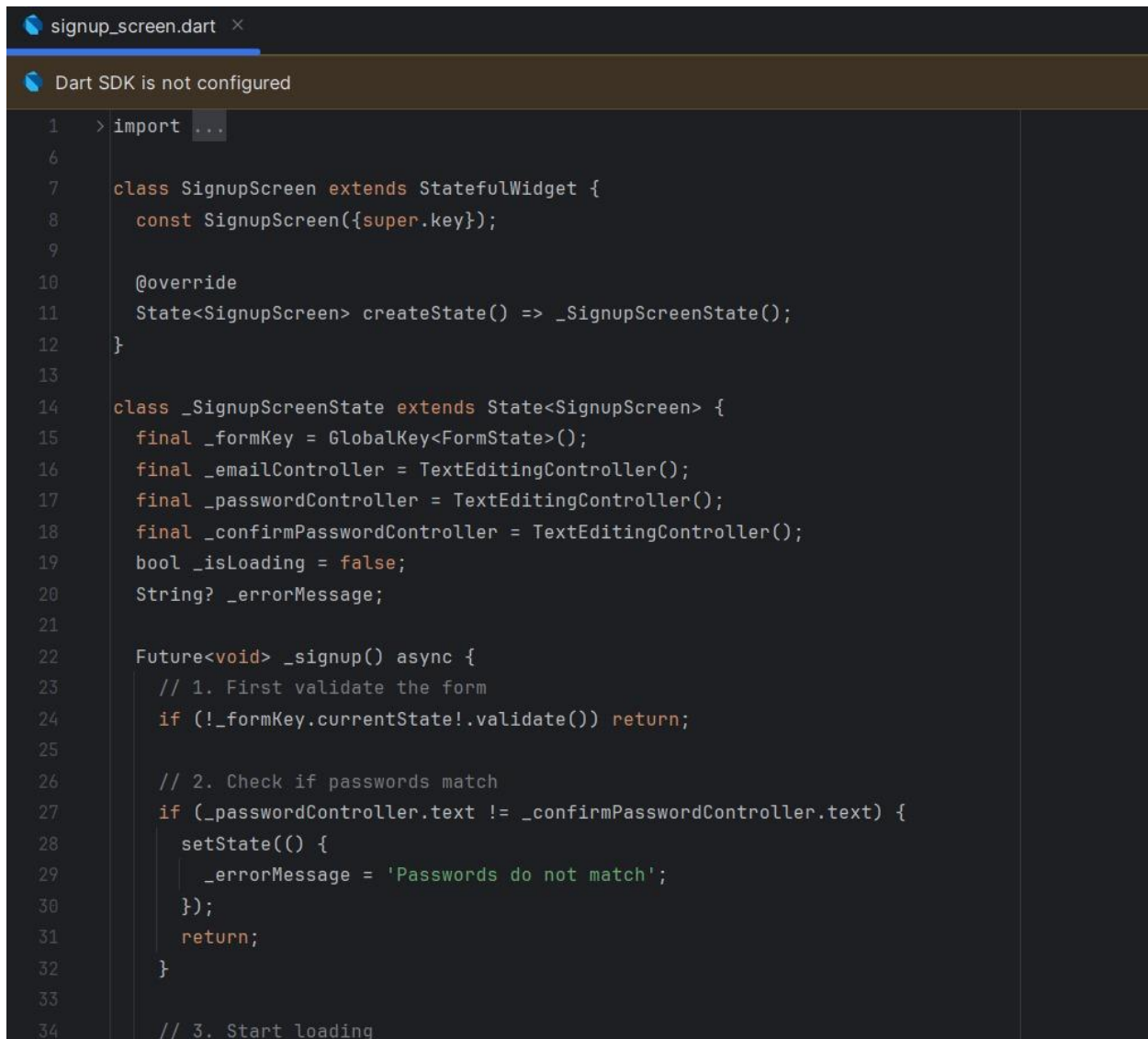
- Include a link to the terms and conditions during the signup process.

### Internationalization:

- Implement localization for error messages and UI texts to support multiple languages.

### Logging:

- Add error logging mechanisms to capture unexpected issues, aiding in debugging and improvement.



```
1  > import ...
6
7  class SignupScreen extends StatefulWidget {
8    const SignupScreen({super.key});
9
10   @override
11   State<SignupScreen> createState() => _SignupScreenState();
12 }
13
14 class _SignupScreenState extends State<SignupScreen> {
15   final _formKey = GlobalKey<FormState>();
16   final _emailController = TextEditingController();
17   final _passwordController = TextEditingController();
18   final _confirmPasswordController = TextEditingController();
19   bool _isLoading = false;
20   String? _errorMessage;
21
22   Future<void> _signup() async {
23     // 1. First validate the form
24     if (!_formKey.currentState!.validate()) return;
25
26     // 2. Check if passwords match
27     if (_passwordController.text != _confirmPasswordController.text) {
28       setState(() {
29         _errorMessage = 'Passwords do not match';
30       });
31       return;
32     }
33
34     // 3. Start loading
```

## *Firebase Configuration for Flutter: Platform-Specific Setup*

### Purpose

This file is a platform-specific configuration for Firebase in a Flutter project, generated using the FlutterFire CLI. It provides the necessary FirebaseOptions for initializing Firebase on various platforms. The file is well-structured but currently supports only Android.

### Key Features

#### 1. Dynamic Platform Detection

- The file uses `defaultTargetPlatform` to identify the current platform and provides the appropriate `FirebaseOptions` based on the detected platform.

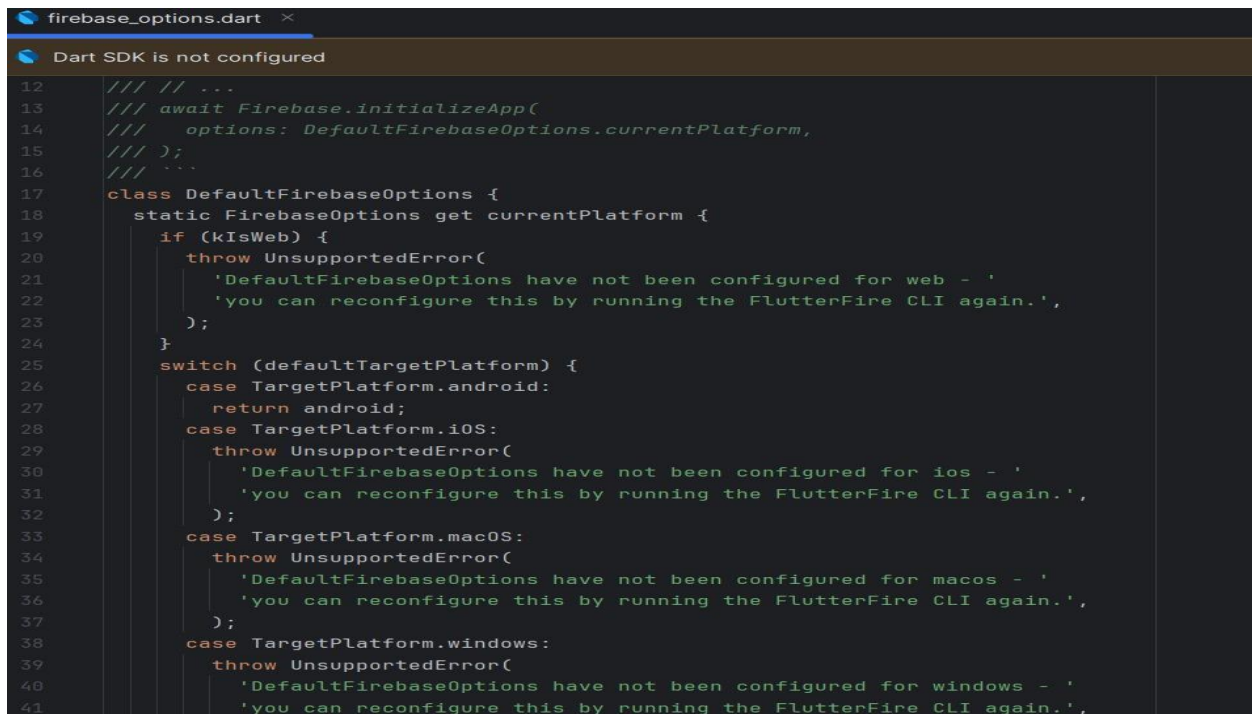
#### 2. Error Handling

- The file throws an `UnsupportedError` for platforms that are not configured, ensuring the app does not proceed without a proper Firebase setup.

#### 3. Android Configuration

- Contains the Firebase configuration specific to the Android platform.

### Android Firebase Configuration



```
12  /// // ...
13  /// await Firebase.initializeApp(
14  ///   options: DefaultFirebaseOptions.currentPlatform,
15  /// );
16  /// ...
17  class DefaultFirebaseOptions {
18    static FirebaseOptions get currentPlatform {
19      if (kIsWeb) {
20        throw UnsupportedError(
21          'DefaultFirebaseOptions have not been configured for web - '
22          'you can reconfigure this by running the FlutterFire CLI again.',
23        );
24      }
25      switch (defaultTargetPlatform) {
26        case TargetPlatform.android:
27          return android;
28        case TargetPlatform.iOS:
29          throw UnsupportedError(
30            'DefaultFirebaseOptions have not been configured for ios - '
31            'you can reconfigure this by running the FlutterFire CLI again.',
32          );
33        case TargetPlatform.macos:
34          throw UnsupportedError(
35            'DefaultFirebaseOptions have not been configured for macos - '
36            'you can reconfigure this by running the FlutterFire CLI again.',
37          );
38        case TargetPlatform.windows:
39          throw UnsupportedError(
40            'DefaultFirebaseOptions have not been configured for windows - '
41            'you can reconfigure this by running the FlutterFire CLI again.',
```

- **apiKey**: Authentication key for Firebase services.
- **appId**: Identifies the Firebase app.
- **messagingSenderId**: Used for Firebase Cloud Messaging (FCM).
- **projectId**: Firebase project identifier.
- **databaseURL**: Realtime Database endpoint.
- **storageBucket**: Location for storing files in Firebase Storage.

## Unsupported Platforms

Currently, this configuration throws an `UnsupportedError` for the following platforms:

- Web
- iOS
- macOS
- Windows
- Linux

## *Food Menu for a Table-Based Restaurant App in Flutter*

This **Home widget** implements a functional and user-friendly food menu for a table-based restaurant app in Flutter. It features search functionality, a categorized menu, and the ability to add or remove items from an order. Below is a breakdown of its key components, along with suggestions for further improvement.

## Key Features

### Dynamic Navigation

- The menu categories (All, Fast Food, Pizza, Drinks, Desserts) are dynamically filtered and displayed using a `PageView`, which is controlled by a `BottomNavigationBar`.

### Search Functionality

- Filters food items in the current category based on user input in the search bar, allowing for quick item selection.

### Dynamic Item Selection

- Allows users to increment or decrement item quantities.

- Displays the count of each selected item directly in the UI for a seamless user experience.

### **Floating Action Button (FAB)**

- Shows the total number of selected items.
- Navigates to an order review page (OrderPage) when clicked, providing a summary of the selected items.

### **Error Handling**

- Handles edge cases, such as reducing an item's quantity to zero, which automatically removes it from the selection.

### **Error Handling for Missing Assets**

- Add placeholder images for missing assets to prevent runtime errors:

### **Improvement Suggestions**

#### **Optimize Search and Filter Logic**

- Move the filtering logic into a helper function to improve readability and reusability across the widget.

#### **Theming Consistency**

- Use a centralized ThemeData for styling across all widgets, ensuring consistent design and easy theme adjustments.

#### **Responsiveness**

- Use MediaQuery for dynamically adjusting the grid layout based on screen size, ensuring the app looks great on all devices.

#### **Modularize Code**

- Extract components like the search bar, food card, and navigation bar into separate widgets. This will improve the readability and maintainability of the code.

#### **Handle Duplicate Item Names**

- Ensure uniqueness in the selection by using IDs instead of item names as keys in `_selectedItems` to avoid potential duplication issues.

#### **Enhance Accessibility**



- Add semantic labels for screen readers to improve accessibility for users with disabilities.

## Potential Additions

### Price and Quantity Summary

- Display the total price alongside the item count in the FAB, giving users a quick view of their order total.

### Animations

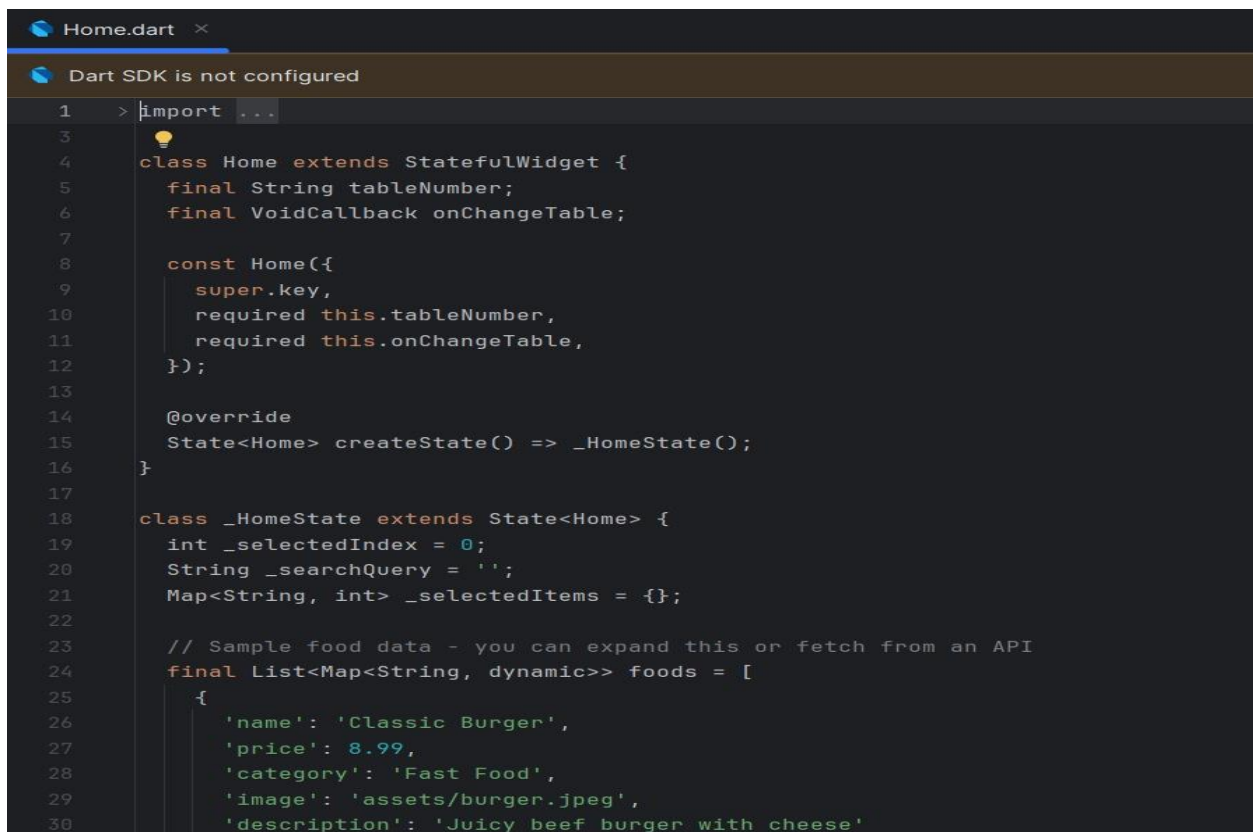
- Add smooth transitions when switching pages or categories to enhance the user experience with fluid and intuitive navigation.

### Database Integration

- Integrate with a backend API or Firebase to fetch food items dynamically, allowing the app to easily update the menu and handle inventory changes.

### Localization

- Support multiple languages for the UI text and food descriptions, enabling the app to serve a global audience.



```
Home.dart x
Dart SDK is not configured
1  > import ...
3
4  class Home extends StatefulWidget {
5      final String tableNumber;
6      final VoidCallback onChangeTable;
7
8      const Home({
9          super.key,
10         required this.tableNumber,
11         required this.onChangeTable,
12     });
13
14     @override
15     State<Home> createState() => _HomeState();
16 }
17
18 class _HomeState extends State<Home> {
19     int _selectedIndex = 0;
20     String _searchQuery = '';
21     Map<String, int> _selectedItems = {};
22
23     // Sample food data - you can expand this or fetch from an API
24     final List<Map<String, dynamic>> foods = [
25         {
26             'name': 'Classic Burger',
27             'price': 8.99,
28             'category': 'Fast Food',
29             'image': 'assets/burger.jpeg',
30             'description': 'Juicy beef burger with cheese'
```

## ***Firebase Initialization and App State Management***

### **Firebase Initialization**

The app begins with Firebase initialization to ensure proper setup for both Android and iOS platforms using the `Firebase.initializeApp()` method. Platform-specific configurations are handled by the `DefaultFirebaseOptions` class, providing the necessary Firebase options.

### **Real-Time State Management with Firebase**

The `AppStateWrapper` widget contains a `StreamBuilder` that listens for changes in the `appState` node of Firebase Realtime Database. This node includes the `isBlocked` field, which determines whether the app should display the blocking screen or allow users to interact with the main content.

- If the `isBlocked` flag is true, the app displays a `BlockingScreen` widget with a customizable message.
- If the `isBlocked` flag is false or not set, the app proceeds to display the main page (`TableSelectionPage`).

### **StreamBuilder for Real-Time Updates**

The `StreamBuilder` listens to updates in the Firebase database, ensuring the app responds to real-time changes. When the database entry for `isBlocked` is modified, the app automatically reflects this change without requiring a manual refresh or restart.

### **UI Design and Theme**

The app utilizes Material Design 3 to create a modern, responsive user interface. The `MaterialApp` widget provides the overall structure, while theme settings define the app's appearance.

- The background color is set to a light shade (`0xFFFF5F5F5`).
- Custom color schemes are applied using a seed color (`0xFF2B2D42`).
- UI elements such as the `AppBar`, buttons, and text fields are styled to ensure a cohesive visual experience throughout the app.

### **Error Handling and Fallback**

The app includes robust error handling to ensure that the user experience remains intact:

- If the appState is not available in Firebase or if there's an issue accessing the data, the app defaults to displaying the TableSelectionPage rather than breaking the user experience.
- The BlockingScreen also handles cases where the blockMessage is missing by showing a default message indicating that the app is disabled.

## **App State Control via Firebase**

The isBlocked flag in Firebase provides a simple mechanism for remotely blocking the app. This feature is useful for maintenance or temporary restrictions. Firebase's real-time nature ensures that once the app is unblocked, users regain access immediately without any downtime.

The Flutter app utilizes Firebase Realtime Database for real-time state management, effectively controlling the availability of app features. The use of StreamBuilder ensures dynamic updates, keeping the app responsive and adaptive. With a Material Design 3-based UI, the app offers a modern, user-friendly experience, while Firebase enables remote control over user access and app state.

## **Potential Improvements**

### **Improved Error Handling**

Enhancing the app's error handling could further increase reliability, particularly for scenarios where the Firebase connection fails or data is unavailable.

### **Performance Optimization**

To improve performance, especially as the app scales, implementing caching mechanisms or pagination for Firebase data could be beneficial.

### **Security Enhancements**

Refining Firebase security rules is crucial to ensure that only authorized users can modify the appState, preventing unauthorized access to critical app features.

order\_page.dart ×

Dart SDK is not configured

```
1  > import ...
5
6  class OrderPage extends StatefulWidget {
7      final Map<String, int> selectedItems;
8      final List<Map<String, dynamic>> foods;
9      final String tableNumber;
10
11     const OrderPage({
12         super.key,
13         required this.selectedItems,
14         required this.foods,
15         required this.tableNumber,
16     });
17
18     @override
19     State<OrderPage> createState() => _OrderPageState();
20 }
21
22 class _OrderPageState extends State<OrderPage> {
23     final databaseRef = FirebaseDatabase.instance.ref();
24     bool isLoading = false;
25
26     double get totalPrice {
27         double total = 0;
28         for (var food in widget.foods) {
29             if (widget.selectedItems.containsKey(food['name'])) {
30                 total += food['price'] * widget.selectedItems[food['name']]!;
31             }
32         }
33         return total;
```

## ***OrderPage Functionality in Flutter App***

This section describes the OrderPage in a Flutter app, where users can review and place their food orders. The page integrates Firebase to upload order details, including the selected items, their quantities, and the total price. Below is a breakdown of the key components and functionality of the page.

### **Key Components**

#### **OrderPage Class:**

- A **stateful widget** that receives three parameters:
  - selectedItems: A map of food names and quantities.
  - foods: A list of available foods with details.
  - tableNumber: The table number associated with the order.

#### **State Management:**

- The `_OrderPageState` class is responsible for managing the state of the order page.
- The total price of the selected items is calculated using the `totalPrice` getter.
- A loading state (`isLoading`) is used to manage the order submission process and prevent multiple submissions.

#### **Firebase Integration:**

- `databaseRef` connects to **Firestore Realtime Database**, where order data (including selected items, total price, and order time) is uploaded.
- The `_uploadOrder` function handles uploading the order to Firebase and updates the UI with a success or error message depending on the result.

#### **Order Item List:**

- A `ListView.builder` is used to display the list of selected items.
  - Each item includes its name, quantity, individual price, and total price.
  - Each item is displayed inside a **Card** widget with an image, name, and price details.

#### **Order Summary:**

- At the bottom of the screen, the total order price is displayed along with an "Order" button.
  - When pressed, the button triggers the `_uploadOrder` function.
  - If an order is being processed (`isLoading` is true), the button shows a loading indicator to prevent further actions.

### App Navigation:

- After a successful order submission, the app displays a confirmation message and navigates the user back to the **TableSelectionPage**, resetting the app for new orders.

### Flow of the App

#### Displaying the Order:

- The `OrderPage` receives the selected food items, calculates the total price, and displays each item with its price, quantity, and a thumbnail image.

#### Order Submission:

- When the user presses the "Order" button, the app sends the order details to Firebase, including the current date and time.
- If the order is successfully uploaded, a success message is displayed; otherwise, an error message is shown.

#### UI Design:

- The UI is styled with a **dark theme** (0xFF121212), and each order item is presented in a card that includes food images and price details.
- The "Order" button is styled to match the app's theme and is disabled during the loading state.

#### Error Handling

If an error occurs during the order submission (e.g., a network issue), an **error message** is displayed using a **SnackBar** to inform the user of the issue.

## ***TableSelectionPage Widget Documentation***

This code defines a TableSelectionPage widget, where users can select a table number to place their order in a restaurant. It is part of a Flutter app that helps users choose a table and then navigate to the next screen to start placing an order.

### **Key Components**

#### **TableSelectionPage Class:**

- **StatefulWidget:** A StatefulWidget that allows the user to select a table. It uses a GridView.builder to display the available tables as buttons.
- **State Management:** The widget maintains two states:
  - **selectedTable:** Stores the current selected table number.
  - **isLoading:** A boolean to track the loading state during table selection.

#### **Table List:**

- **Table Generation:** The tables list generates table numbers from 1 to 20. Each table is represented by a button in a grid.
- **Button Interaction:** When a user presses a table button, the button's style changes to indicate the selected table.

#### **Table Selection and Navigation:**

- **\_selectTable Method:** This method is invoked when a table is selected. It simulates a loading state (using Future.delayed). After the loading, the app navigates to the Home page.
- **Passing Data:** The Home page receives the selectedTable as a parameter, ensuring the chosen table is passed to the next screen.

#### **Loading Indicator:**

- **Loading State:** If the user is waiting for the table selection process to complete (when isLoading == true), a semi-transparent overlay with a CircularProgressIndicator appears on the screen to indicate that the action is being processed.

#### **UI Design:**

- **Table Grid:** The UI displays a grid of tables, where each table is represented by an ElevatedButton.

- **Button Colors:** The buttons have different colors based on whether the table is selected or not. The selected table button uses a secondary color from the theme.
- **Grid Layout:** The grid layout is managed with a `SliverGridDelegateWithFixedCrossAxisCount` to create a 2-column grid with spacing between the items.

### Navigation:

- **Navigation to Home:** Once a table is selected, the `Navigator.pushReplacement` method is used to navigate to the Home screen, passing the selected table number and a callback function (`onChangeTable`) that allows the user to go back to the `TableSelectionPage` if needed.

### Flow of the App

1. **Displaying Table Options:** The `TableSelectionPage` displays a list of tables (numbered from 1 to 20) in a grid layout. Each table is represented by a button.
2. **User Selection:** When the user presses a table button, it triggers the `_selectTable` function. The function sets the `selectedTable` variable and simulates a loading process using `Future.delayed`.
3. **Navigation to Home Page:** After the simulated loading time, the app navigates to the Home page, passing the selected table number along with a callback function to allow changing the table if needed.
4. **Loading Overlay:** During the loading process (while waiting for the table to be selected), the UI shows a `CircularProgressIndicator` in a semi-transparent overlay to inform the user that their action is being processed.

### *Flutter Widget Test: Counter App Functionality*

This is a simple Flutter widget test used to verify the basic functionality of a counter app. The test checks whether the counter value is incremented when the user taps a button.

### Key Components of the Code:

#### Test Setup:

- The test uses the `flutter_test` package, which is specifically used for writing unit and widget tests in Flutter.



- `testWidgets` is used to define the widget test, where the first argument is the description of the test, and the second argument is the test function.

### **Test Steps:**

#### **1. Building the App:**

This line builds the `KitchenApp` widget and triggers the first frame to load the app.

#### **2. Initial State Verification:**

The test checks that initially the text 0 is present on the screen and that the text 1 is not found.

#### **3. Simulating a Tap Gesture:**

This simulates the user tapping on the '+' icon (presumably a button that increments the counter). `tester.pump()` triggers a rebuild of the widget, allowing the app to update after the tap.

#### **4. Post-Tap Verification:**

The test checks that after the tap, the text 0 is no longer displayed, and the text 1 is now visible, indicating that the counter has been incremented.

### **Purpose of the Test:**

- **Widget Interaction:** The test simulates a user interaction (tapping the '+' icon) and checks the changes in the UI after the interaction.
- **Verifying State Change:** It ensures that the state change (the increment of the counter) is reflected correctly in the widget tree, confirming that the button's functionality is working as expected.

## **IX. Conclusion**

The Restaurant Management App provides an efficient, user-friendly solution for managing restaurant operations. By leveraging Flutter for cross-platform compatibility and integrating Firebase for authentication, real-time data, and app monitoring, the app enhances both the operational efficiency of restaurant staff and the overall customer experience. This digital platform simplifies key processes, ensuring a seamless flow of information and improving day-to-day tasks, making it a valuable tool for modern restaurant management.