

Flutter with Firebase Example App



+



Flutter

Table of contents

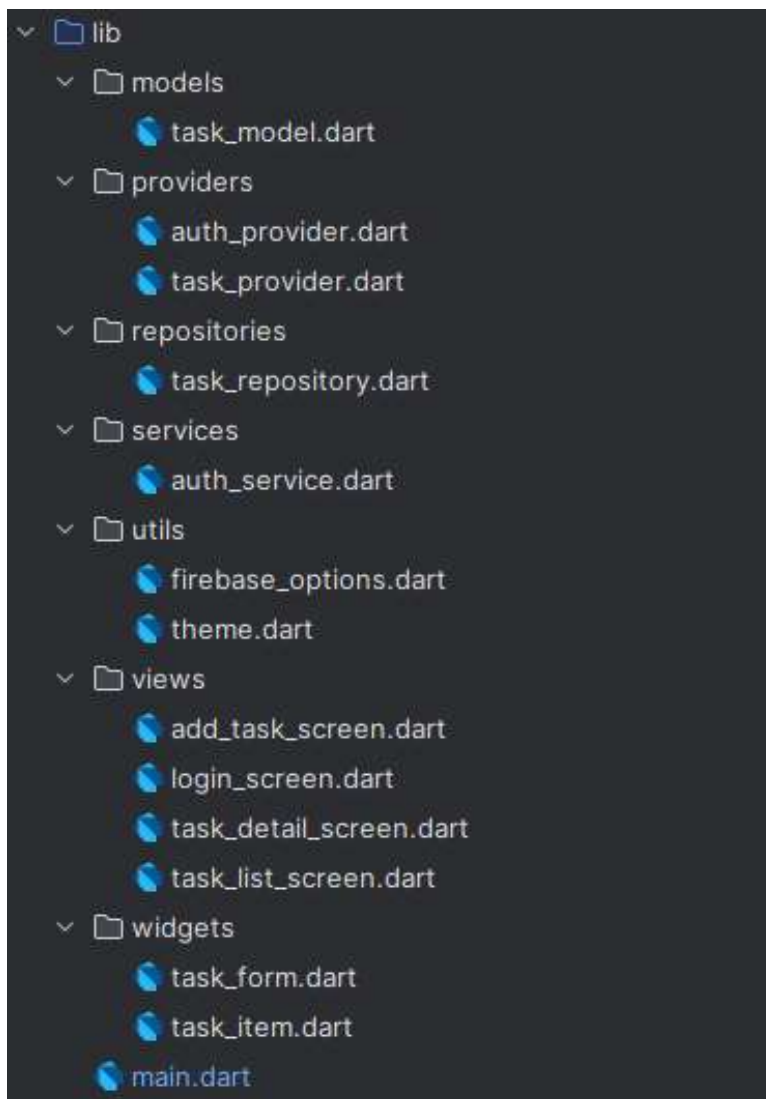
Introduction.....	2
Project structure and architecture.....	2
Important pieces of code	3
Use of Firebase Authentication	3
Use of Firebase Firestore Database.....	6
Initialize Firebase and Provider	9
Configure Firebase for the project	10
Install Firebase CLI	10
Connect the Flutter Project with Firebase	10
Test the connection.....	11
Firestore Database Rules.....	11
User Guide	12
Login / Register.....	12
Main Page.....	13
Add new / update Task.....	14

Introduction

The demo app was written with Flutter and uses Firebase's Firestore Database to store tasks and Firebase Authentication to verify the user. In the app, the user is prompted to log in initially. If no login is available, a new account can be created. Once logged in, the user sees the tasks they have created and can create, delete or update new tasks. The app was written to give an initial idea of how to use Flutter and Firebase.

Project structure and architecture

The App is structured according to the Model-View-ViewModel (MVVM) pattern to promote the separation of concerns and improve maintainability. The following image shows the project structure.



Important pieces of code

Use of Firebase Authentication

This chapter takes a closer look at the service class and the provider class of the authentication. These are used for login and registration. The provider is also responsible for managing the user's state.

auth_service.dart

The AuthService encapsulates all communication with Firebase Authentication. It contains methods for login, registration and logout as well as a stream that monitors changes to the authentication status.

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

class AuthService {
  final FirebaseAuth _auth = FirebaseAuth.instance;

  User? get currentUser => _auth.currentUser;

  Stream<User?> authStateChanges() => _auth.authStateChanges();

  Future<String?> signInWithEmailAndPassword(String email, String password)
  async {
    try {
      await _auth.signInWithEmailAndPassword(email: email, password:
password);
      return null;
    } on FirebaseAuthException catch (e) {
      return _handleAuthException(e);
    } catch (e) {
      return e.toString();
    }
  }

  Future<String?> registerWithEmailAndPassword(String email, String
password) async {
    try {
      await _auth.createUserWithEmailAndPassword(email: email, password:
password);
      return null;
    } on FirebaseAuthException catch (e) {
      return _handleAuthException(e);
    } catch (e) {
      return e.toString();
    }
  }

  Future<void> signOut() async {
    try {
      await _auth.signOut();
    } catch (e) {
      print(e.toString());
    }
  }

  String _handleAuthException(FirebaseAuthException e) {
    switch (e.code) {
      case 'user-not-found':
```

```

        return 'No user found for that email.';
    case 'wrong-password':
        return 'Wrong password provided for that user.';
    case 'weak-password':
        return 'The password provided is too weak.';
    case 'email-already-in-use':
        return 'The account already exists for that email.';
    default:
        return e.message ?? 'An unknown error occurred.';
    }
}
}

```

auth: An instance of FirebaseAuth that is used to perform authentication operations.

currentUser: A getter method that returns the currently logged in user.

authStateChanges: A stream that monitors changes to the authentication status. This is used to respond to logins and logouts.

signInWithEmailAndPassword: A method for signing in a user with email and password. If successful, zero is returned, otherwise an error message.

registerWithEmailAndPassword: A method for registering a new user with email and password. If successful, zero is returned, otherwise an error message.

signOut: A method for logging out the currently logged in user.

_handleAuthException: A private method for error handling that returns specific error messages based on the Firebase error code.

auth_provider.dart

The AuthProvider uses the AuthService to encapsulate the authentication logic and manage the state of the app. It notifies the UI components about changes to the authentication status.

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import '../services/auth_service.dart';

class AuthProvider with ChangeNotifier {
  final AuthService _authService = AuthService();
  User? _user;
  User? get user => _user;

  AuthProvider() {
    _authService.authStateChanges().listen((User? user) {
      _user = user;
      notifyListeners();
    });
  }

  Future<String?> signInWithEmailAndPassword(String email, String password)
  async {
    return await _authService.signInWithEmailAndPassword(email, password);
  }

  Future<String?> registerWithEmailAndPassword(String email, String
  password) async {
    return await _authService.registerWithEmailAndPassword(email,
    password);
  }

  Future<void> signOut() async {
    await _authService.signOut();
  }
}
```

_authService: An instance of AuthService that is used to perform authentication operations.

_user: The currently logged in user. This variable is updated when the authentication status changes.

user: A getter method that returns the currently logged in user.

Constructor: The authentication status is monitored in the constructor. If the status changes, _user is updated and notifyListeners() is called to notify all registered listeners.

signInWithEmailAndPassword: This method calls the corresponding method in the AuthService and returns the result.

registerWithEmailAndPassword: This method calls the corresponding method in the AuthService and returns the result.

signOut: This method calls the signOut method in the AuthService.

Use of Firebase Firestore Database

The service class and provider class for the Firestore database are now analysed in more detail. These classes perform the CRUD functions on the database and provide a list of tasks.

task_repository.dart

The TaskRepository encapsulates all communication with Firebase Firestore. It contains methods for retrieving, adding, updating and deleting tasks.

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

import '../models/task_model.dart';

class TaskRepository {
  final FirebaseFirestore _db = FirebaseFirestore.instance;

  Future<List<Task>> fetchTasks(String userId) async {
    final snapshot = await
    _db.collection('users').doc(userId).collection('tasks').get();
    return snapshot.docs.map((doc) => Task.fromMap(doc.data(),
    doc.id)).toList();
  }

  Future<void> addTask(Task task) async {
    final docRef = await
    _db.collection('users').doc(user.uid).collection('tasks').add(task.toMap());
    ;
    task.id = docRef.id;
  }

  Future<void> updateTask(Task task) async {
    await
    _db.collection('users').doc(user.uid).collection('tasks').doc(task.id).update(task.toMap());
  }

  Future<void> deleteTask(String taskId) async {
    await
    _db.collection('users').doc(user.uid).collection('tasks').doc(taskId).delete();
  }
}
```

_db: An instance of FirebaseFirestore that is used to perform Firestore operations.

fetchTasks: This method retrieves all tasks of a specific user. It takes the user ID as a parameter, performs a query in Firestore and returns a list of task objects.

addTask: This method adds a new task for the user currently logged in. It receives a task object, adds it to Firestore and updates the ID of the task with the document ID generated by Firestore.

updateTask: This method updates an existing task. It takes a task object, searches for the corresponding document in Firestore and updates it.

deleteTask: This method deletes a task. It takes the task ID as a parameter, searches for the corresponding document in Firestore and deletes it.

task_provider.dart

The TaskProvider uses the TaskService to encapsulate the business logic of task management and to manage the status of the tasks. It notifies the UI components about changes to the task status.

```
import 'package:flutter/material.dart';
import '../models/task_model.dart';
import '../services/task_repository.dart';
import 'auth_provider.dart'

class TaskProvider with ChangeNotifier {
  final TaskRepository _repository;
  late AuthProvider? _authProvider;

  List<Task> _tasks = [];
  bool _isLoading = false;

  List<Task> get tasks => _tasks;
  bool get isLoading => _isLoading;

  TaskProvider(this._repository);

  void updateAuthProvider(AuthProvider authProvider) {
    _authProvider = authProvider;
    _refreshTasks();
  }

  Future<void> _refreshTasks() async {
    final user = _authProvider?.user;
    if (user == null) {
      _tasks = [];
      notifyListeners();
      return;
    }

    _isLoading = true;
    notifyListeners();

    _tasks = await _repository.fetchTasks(user.uid);

    _isLoading = false;
    notifyListeners();
  }

  Future<void> addTask(Task task) async {
    final user = _authProvider?.user;
    if (user == null) return;
    await _repository.addTask(task, user.uid);
    _tasks.add(task);
    notifyListeners();
  }

  Future<void> updateTask(Task task) async {
    final user = _authProvider?.user;
    if (user == null) return;
    await _repository.updateTask(task, user.uid);
    final index = _tasks.indexWhere((t) => t.id == task.id);
    if (index != -1) {
      _tasks[index] = task;
      notifyListeners();
    }
  }
}
```



```

    }

    Future<void> deleteTask(String taskId) async {
        final user = _authProvider?.user;
        if (user == null) return;
        await _repository.deleteTask(taskId, user.uid);
        _tasks.removeWhere((t) => t.id == taskId);
        notifyListeners();
    }
}

```

_repository: An instance of TaskRepository that handles all Firestore operations for tasks.

_authProvider: A reference to AuthProvider, used to access the current authenticated user.

_tasks: A private list of Task objects representing the user's current tasks.

_isLoading: A boolean that tracks whether task data is currently being loaded.

tasks: A getter that exposes the current list of tasks.

isLoading: A getter that exposes the current loading state.

Constructor: The constructor initializes the repository but does not immediately load tasks. The updateAuthProvider method must be called afterward.

updateAuthProvider: Sets the AuthProvider instance and immediately refreshes the task list for the authenticated user.

_refreshTasks: Private method that checks if a user is logged in, loads their tasks from Firestore, and updates _tasks and _isLoading, notifying listeners accordingly.

addTask: Adds a new task for the current user in Firestore, updates the local list, and notifies listeners.

updateTask: Updates an existing task for the current user in Firestore and updates the local list accordingly.

deleteTask: Deletes a task in Firestore for the current user and removes it from the local list.

Initialize Firebase and Provider

main.dart

```
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:provider/provider.dart';
import 'package:task_manager/utils/theme.dart';
import 'providers/auth_provider.dart';
import 'providers/task_provider.dart';
import 'views/login_screen.dart';
import 'views/task_list_screen.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => AuthProvider()),
        ChangeNotifierProxyProvider<AuthProvider, TaskProvider>(
          create: (_) => TaskProvider(TaskRepository()),
          update: (_, authProvider, taskProvider) =>
            taskProvider!..updateAuthProvider(authProvider),
        ),
      ],
      child: Consumer<AuthProvider>(
        builder: (context, auth, _) {
          return MaterialApp(
            title: 'Task Manager App',
            theme: buildThemeData(),
            home: auth.user != null ? TaskListScreen() : LoginScreen(),
          );
        },
      ),
    );
  }
}
```

MyApp: The root widget of the app, implemented as a StatelessWidget.

MultiProvider: Used to inject multiple providers into the widget tree:

- AuthProvider: Manages authentication state and user login/logout.
- TaskProvider: Manages task data. Depends on AuthProvider via ChangeNotifierProxyProvider.

ChangeNotifierProxyProvider: Updates TaskProvider whenever AuthProvider changes, ensuring it always has the current user context.

Consumer<AuthProvider>: Reactively rebuilds the app based on the authentication state.

MaterialApp: The main application widget that sets the app title, theme, and determines which screen (login or task list) to display.

home: Displays either LoginScreen or TaskListScreen, depending on whether the user is logged in.

Configure Firebase for the project

Before starting, ensure Flutter is installed on your device. First, create a new Flutter project using the following command:

```
flutter create your_app_name
```

Next, navigate to the Firebase Console and create a new project. Enter a name and disable Google Analytics. To add Android, iOS, and Web support, use the Firebase CLI.

<https://firebase.google.com/docs/cli>

Install Firebase CLI

Open a command prompt and install the CLI with npm:

```
npm install -g firebase-tools
```

After installing the Firebase CLI, log in to your Firebase account:

```
firebase login
```

To test the connection to Firebase, list all your projects with the following command:

```
firebase projects:list
```

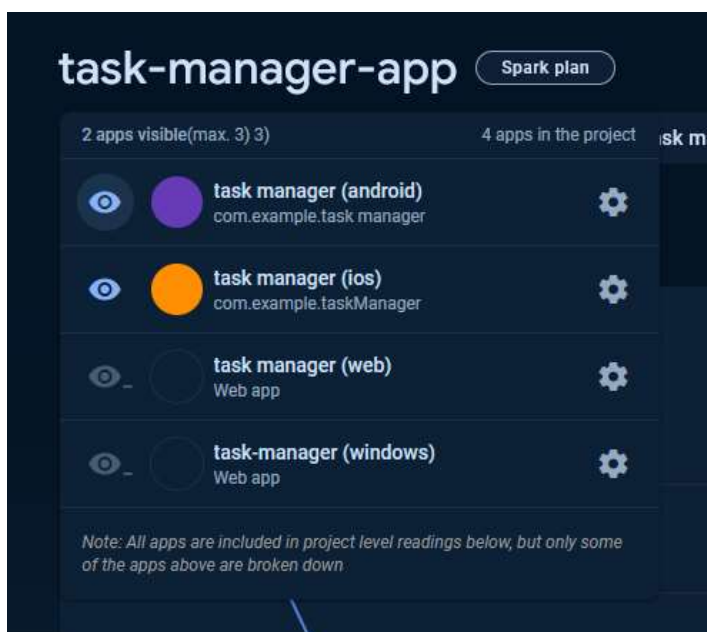
Connect the Flutter Project with Firebase

Switch to the root folder of your Flutter project and configure Firebase:

```
flutterfire configure
```

A list of all your Firebase projects will appear. Select your project and press Enter. Then choose which platform(s) you want to support.

After the command completes, you will see a `firebase_options.dart` file, indicating that all configurations for the platforms are done. When you reopen the Firebase Console, the connected apps will be visible.



Test the connection

To test if the connection is working, modify the `main` function in your `main.dart` file as follows:

```
void main() async {  
  WidgetsFlutterBinding.ensureInitialized();  
  await Firebase.initializeApp();  
  runApp(MyApp());  
}
```

Add `firebase_core` to your project to run this:

flutter pub add firebase_core

Now, run your project. If no errors occur, the connection to Firebase was successful.

Firestore Database Rules

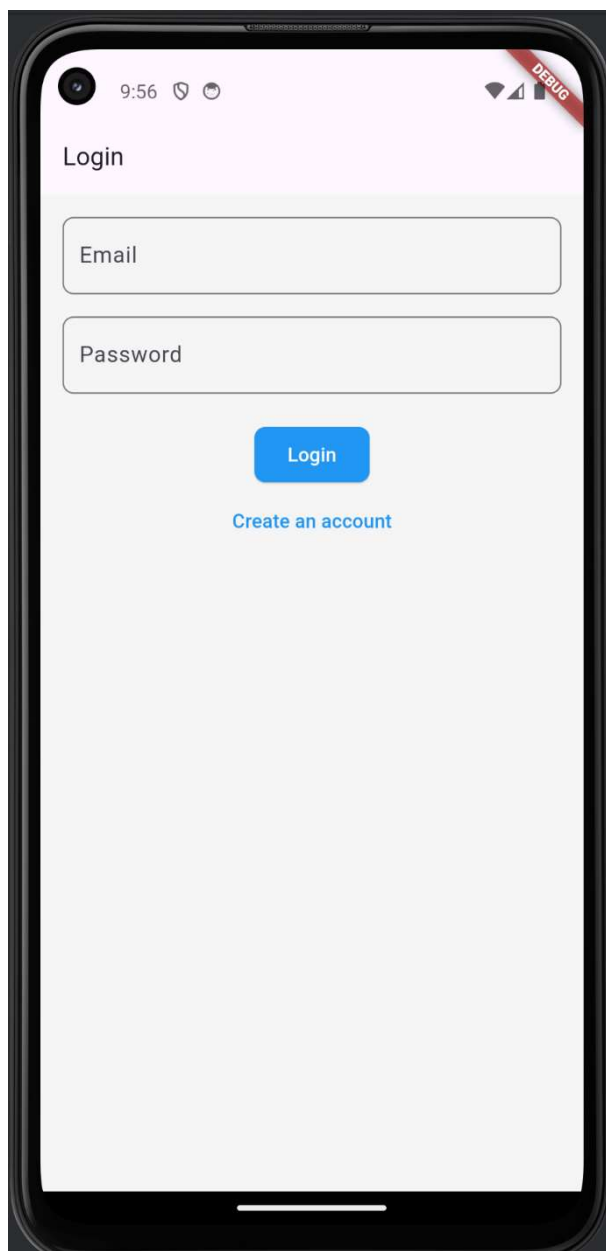
Rules can be defined to control and restrict access to the database in Firestore Database. The following image shows an example of what these rules could look like. These example rules regulate access to the document with the `userId` and to the tasks belonging to the user. The user only has read and write access if he is authenticated.

```
1  rules_version = '2';  
2  
3  service cloud.firestore {  
4    match /databases/{database}/documents {  
5      // Match user document  
6      match /users/{userId} {  
7        allow read, write: if request.auth != null && request.auth.uid == userId;  
8  
9        // Match tasks subcollection  
10       match /tasks/{taskId} {  
11         allow read, write: if request.auth != null && request.auth.uid == userId;  
12       }  
13     }  
14   }  
15 }  
16
```

User Guide

Login / Register

When the app is started, you will be taken to the login page. If you have already created a login, you can log in with your email address and password. Otherwise, you can click on Create an account and create a new account. You must use a valid email address to create a new account. Your chosen password must be at least 6 characters long.



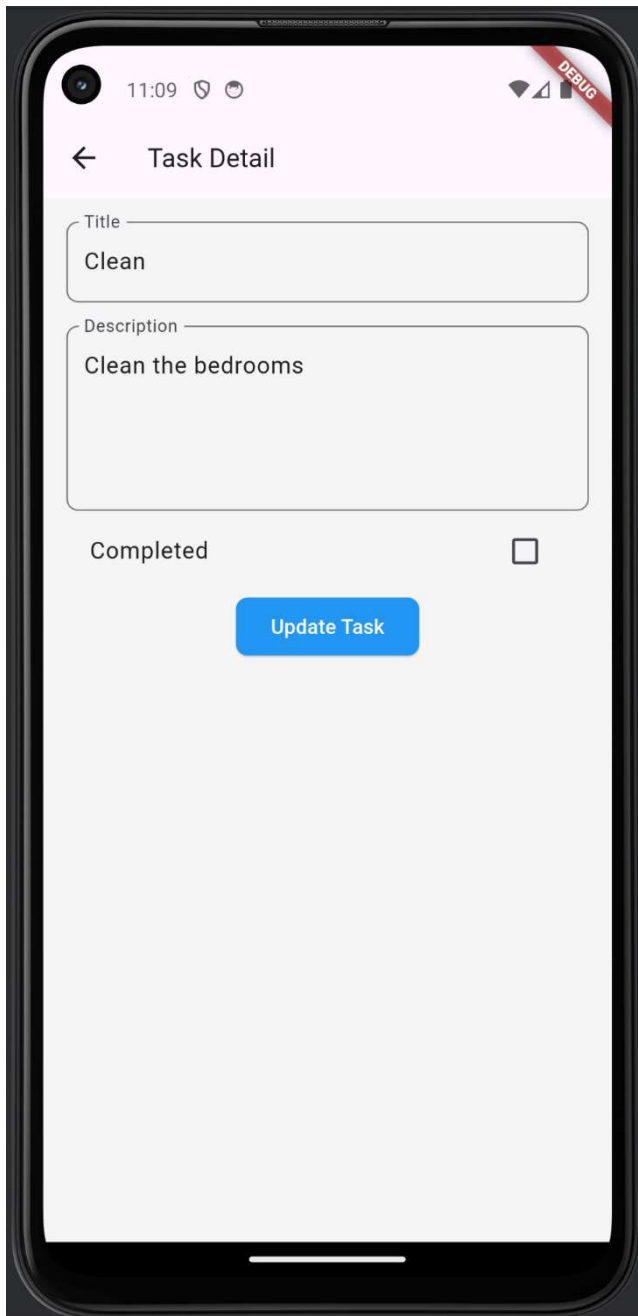
Main Page

On the main page you can see all the tasks you have created. You can delete a task by clicking on the rubbish bin symbol next to it. This is followed by a dialogue window in which you must confirm the deletion process. By clicking on a task, you will be redirected to a page where you can edit the task. In the top right-hand corner, there is a button with which you can log out. You will be redirected to the login / register page. In the bottom right corner, there is a button with a plus sign, by clicking on this button you will be redirected to a page where you can create a new task.



Add new / update Task

If you click on the plus button or on a task, you will be forwarded to a form. Here you can enter the title and a description. When you update a task, the previous values are already in the input fields. There is also the option of clicking on a small box if the task has been completed. Click on the blue button to return to the main page.



The screenshot shows a mobile application interface for editing a task. At the top, the status bar displays the time 11:09 and various icons. The app's header is pink and contains a back arrow and the title 'Task Detail'. The form itself is light gray and contains two input fields: 'Title' with the text 'Clean' and 'Description' with the text 'Clean the bedrooms'. Below these fields is a 'Completed' checkbox, which is currently unchecked. At the bottom of the form is a blue button labeled 'Update Task'. A red 'DEBUG' banner is visible in the top right corner of the app screen.