

Thesis title	Journal app development
Thesis type	Project report
Course name	Mobile Software Engineering II
Course of study	Bachelor of Science Computer Science; Semester 6
Date	16 July 2024
Author	Sergei Ulvis
Matriculation number	32113004
Tutor	Christian Remfert

Table of Contents

Introduction	3
GitHub resource	3
Concept	3
Architectural approach and top-level components interaction	3
Key characteristics of MVVM in project:	5
Wireframes	5
Login application info and screenshots	6
Components of the application	6
Implementation of core functions	7
Critical evaluation and possible improvements	15
Lessons learned	15
Bibliography	16

Introduction

This document presents the design, approaches used and final functionality of Android mobile application “ToDo-s Journal App”. The application is built using Flutter (Google framework for crafting natively compiled applications for mobile, web, and desktop from a single codebase), Material design as it’s built-in in Flutter, Google Firebase as backend for text and photos storage, and MobX package as state management.

My choice is based on opinion that Flutter is ideal for cross-platform development, allowing to write a single codebase that runs on iOS, Android, web, and desktop. It offers fast development with hot reload, high performance due to native ARM code compilation, and beautiful, customizable UIs with a comprehensive widget library. Backed by Google, Flutter ensures strong support, a growing ecosystem, and reduced development costs, making it a future-proof choice.

GitHub resource

Full project code is in public GitHub repository available by link:

https://github.com/Wolfram-180/iu_dlbsemse02_task3_journal_app

Concept

Main features of “ToDo-s Journal App” application are:

- user registration, log in/out and account deletion possibilities
- short texts (journal entries) can be saved and edited
- allow photo to be selected from phone`s memory, and easily assigned (and removed) to journal entry
- full-screen photo pop-up mode
- “done” mark to be used to mark journal entry as done / not done
- journal entries are sorted initially by done/not done attribute, than by creation date-time
- emojis could be used in entry text

Architectural approach and top-level components interaction

The architectural approach used is MVVM (Model-View-ViewModel), which is a common pattern in Flutter applications.

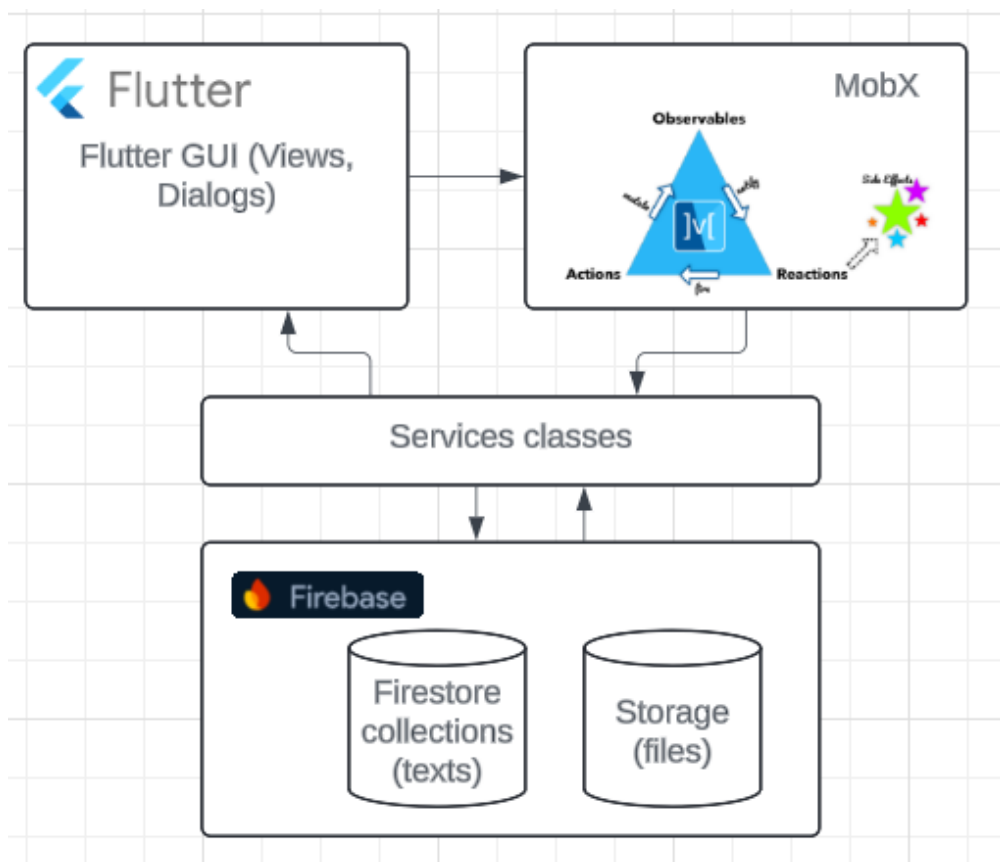
Model: represents the data layer and business logic of the application. This includes files in the state directory like `entry.dart`, `app_state.dart`, and their generated counterparts (`entry.g.dart`,

app_state.g.dart). These files define the structure of data and handle the state management, containing methods marked with @action, which allows them to modify MobX state.

View: the UI components of the application, which are responsible for presenting the data to the user and capturing user input. This layer is represented by the views directory, containing files like entries_list_view.dart, login_view.dart, and register_view.dart.

ViewModel: acts as an intermediary between the View and the Model. It processes the user input from the View, interacts with the Model, and updates the View accordingly. This functionality is distributed among various services and state management classes:

- services directory: files like auth_service.dart, entries_service.dart, and image_actions_service.dart handle business logic and data fetching; separation of services to abstract classes and their implementations allows to quickly replace implementations if needed to replace Firebase backend with some other backend kind.
- state directory: the app_state.dart file and related generated files manage the application's state and provide the necessary data to the views.
- dialogs directory: dialogs system built on common showGenericDialog (generic_dialog.dart) widget to avoid several screens / dialogs repetition.



Key characteristics of MVVM in project:

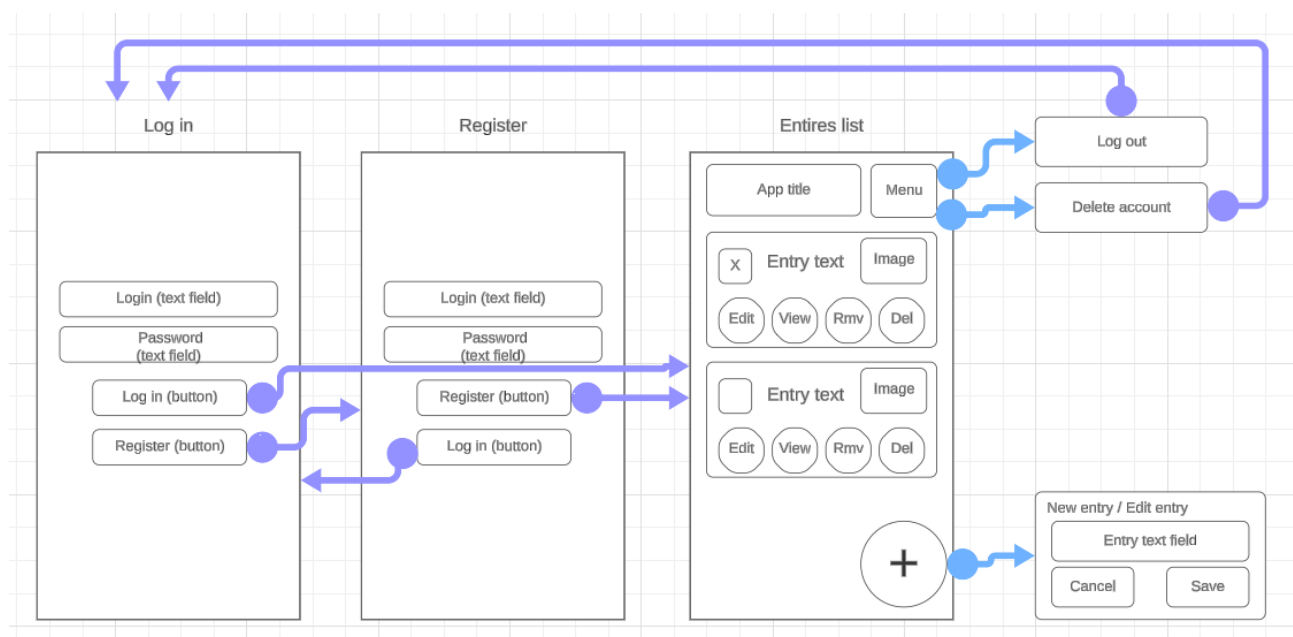
- data binding: the MobX state management solution used (e.g., via `app_state.dart`) allows the View to automatically reflect changes in the Model. This is a hallmark of MVVM, enabling a reactive UI.
- separation of concerns: the clear division of responsibilities between Views, Models, and ViewModels helps maintain a clean and manageable codebase.
- testability: by decoupling the View from the Model, MVVM enhances testability. ViewModel could be tested independently of the UI, which is critical for maintaining robust applications.

Wireframes

Following wireframes prepared in Lucidchart and available (shared) by links below.

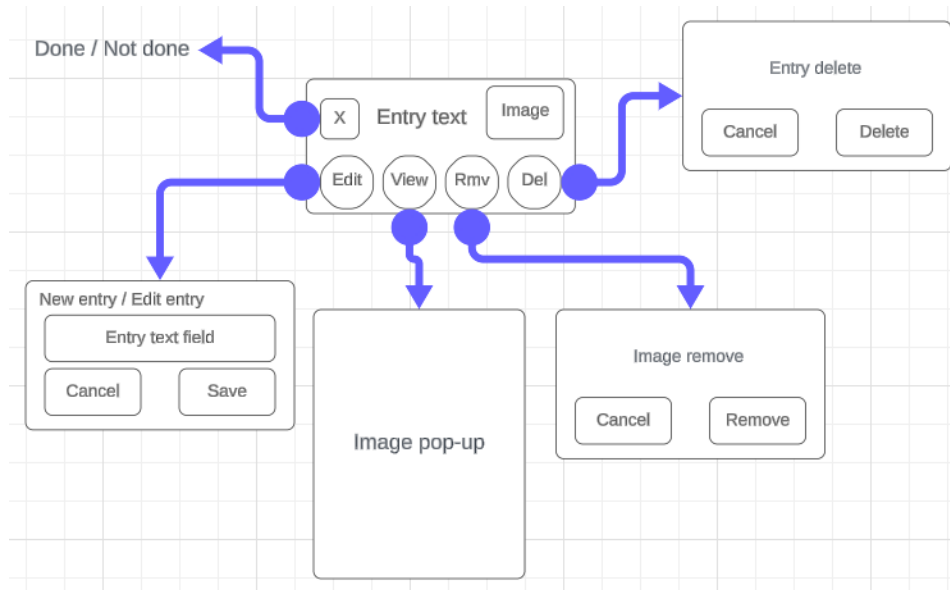
Screens routing: there are Login, Register and Main screens in application, with several pop-ups and dialogs.

<https://lucid.app/lucidchart/848675b7-79f1-4564-94e1-a63f8c5ed836/view>



Wireframe of Entry widget actions:

<https://lucid.app/lucidchart/ac21250f-54f7-4671-b560-7e8c7bcfaecb/view>



Login application info and screenshots

Login: sergei.ulvis@gmail.com

Password: dabracababra

Screenshots saved in “screenshots” folder.

Components of the application

The application consists of several key components organized into directories:

- main.dart: the entry point of the application
- firebase_options.dart: configures Firebase for the app (file is in .gitignore and not pushed to GitHub to avoid API keys publishing, will be provided in full text view in following pages)
- auth_service.dart: manages user authentication; contains base abstract class AuthService and overriding Firebase-aligned methods
- auth_error.dart: contains map of Firebase-aligned authentication errors, base abstract class AuthError (with properties dialogTitle, dialogText used in error dialogs) and errors-map aligned classes extending AuthError
- dialogs folder: contains custom dialog widgets using generic_dialog.dart as base widget, providing specific title, content, options – and receiving response for specified situations
- extensions folder: IfDebugging extension on String class for debugging purposes, allows in-place String fulfillment with some pre-defined values especially for debugging mode

- loading folder: contains LoadingScreen widget based on Singleton pattern (factory method returns previously created _shared instance), used in MobX state management approach in main.dart as Reaction : ReactionBuilder is part of the core MobX state control, fires whenever the dependent observables change, i.e. showing LoadingScreen as overlay in case AppState changed to isLoading
- services folder: contains services for data handling and business logic; to support code clearance and separation of concerns – base abstract classes used with further Firebase-aligned implementation, which allows to quickly replace implementations if needed without deep code refactoring
- state folder: main app classes using MobX Store to contain Observables values and Actions to modify Observables and as result to change state triggering reactive UI changes; partial “....g.dart” files are generated using “flutter pub run build_runner” command residing in watcher.cmd file (placed in project root)
- views folder: contains the UI widgets (parts of screens) of the application
- test folder: used in unit tests, “mocks” folder contains services mocks imitating real services, utils.dart contains test-aligned supporting utilities, and app_state_test.dart contains tests (detailed description in Testing approach section below).

Implementation of core functions

Navigation between screens: implemented using the Observer widget from the MobX package, which reacts to changes in the application state. The AppState class holds the state of the current screen, represented by an enum AppScreen. The Observer widget listens for changes in the currentScreen property of the AppState. Whenever currentScreen changes, the Observer widget rebuilds the UI to display the appropriate screen. The builder function inside the Observer widget uses a switch statement to return the corresponding screen widget based on the value of currentScreen. This ensures that the UI dynamically updates to reflect the current state.

Initial observable value:

```

29  /// in MobX, observable defines a trackable field that stores the state
30  ///
31  /// initial value of currentScreen is AppScreen.login
32  /// as interaction starts from user`s login
33  @observable
34  AppScreen currentScreen = AppScreen.login;

```

Action changing observable value to provided parameter:

```
/// in MobX, action marks a method as an action that will modify the observable properties
/// if method is not marked as action - observable modification from non-"action" method
/// will raise runtime exception
///
/// goTo action change currentScreen observable to provided AppScreen
/// to support routing
@action
void goTo(AppScreen screen) {
  currentScreen = screen;
}
```

Button with onPressed calling goTo() action with new AppScreen value:

```
main.dart  register_view.dart  login_view.dart
lib > views > register_view.dart > RegisterView > build
8   class RegisterView extends HookWidget {
12   Widget build(BuildContext context) {
66     const SizedBox(
67       height: 20,
68     ), // SizedBox
69     ElevatedButton(
70       child: const Text('Already registered? Log in here!'),
71       onPressed: () {
72         context.read<AppState>().goTo(
73           AppScreen.login,
74         );
75       },
76     ), // ElevatedButton
```

Observer reacting on new AppScreen value:

```
main.dart
lib > main.dart > ...
40   class App extends StatelessWidget {
44     Widget build(BuildContext context) {
82
83     /// MobX Observer used to route between screens
84     /// depending on AppState.currentScreen getter
85     child: Observer(
86       name: 'CurrentScreen',
87       builder: (context) {
88         switch (context.read<AppState>().currentScreen) {
89           case AppScreen.login:
90             return const LoginView();
91           case AppScreen.register:
92             return const RegisterView();
93           case AppScreen.journalEntries:
94             return const EntriesView();
95         }
96       },
97     ), // Observer
```

This approach leverages the reactive capabilities of MobX to manage screen navigation efficiently. By using the Observer widget, the application ensures that the UI remains in sync with the state, providing a seamless user experience. This method is both scalable and maintainable, allowing for easy updates and additions to the navigation logic.

Handling user interactions: involves capturing user inputs, processing them, and updating the UI accordingly. In the project user interactions are managed using MobX for state management and various Flutter widgets for capturing inputs. User inputs are captured using Flutter's dialogs and input widgets such as TextFormField, ElevatedButton, and others. The AppState class contains observable properties and actions to manage user interactions. When a user interacts with the UI, these properties are updated, triggering reactive updates in the UI. The UI reacts to state changes using the Observer widget. When the state changes (e.g., user inputs email or password, or the login process starts), the Observer widget rebuilds the relevant parts of the UI.

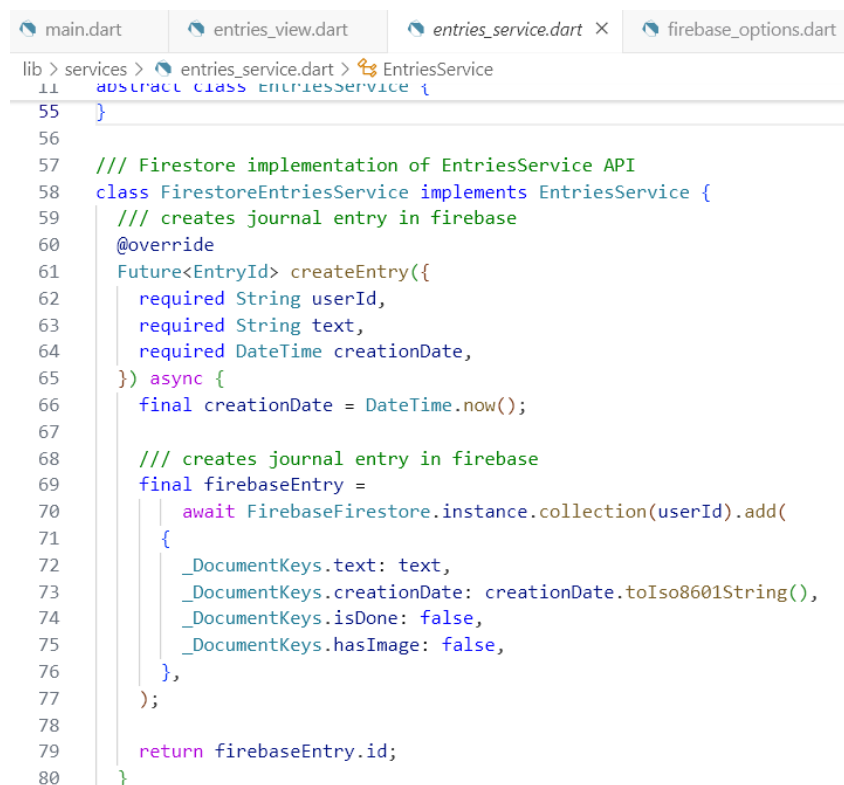
As example, new Entry creation process: new entryText received from pop-up dialog and createEntry action started with entryText as parameter:

```

19 Future<void> _addEntry() async {
20   // capturing context locally to avoid issues with async gaps
21   final localContext = context;
22   final entryText = await showTextFormFieldDialog(
23     context: localContext,
24     title: 'What is entry about?',
25     hintText: 'Enter your journal entry here',
26     optionsBuilder: () => {
27       TextFieldDialogButtonType.cancel: 'Cancel',
28       TextFieldDialogButtonType.confirm: 'Save',
29     },
30   );
31   if (!mounted) return;
32   if (entryText == null) {
33     return;
34   }
35   context.read<AppState>().createEntry(entryText);
36 }

```

Data storing, updating and retrieval: journal entry model described in state/entry.dart file; text data (journal entries) are stored in Firestore using a model class and service to handle CRUD operations:



The screenshot shows an IDE with four tabs: main.dart, entries_view.dart, entries_service.dart (active), and firebase_options.dart. The active tab displays the following Dart code:

```

lib > services > entries_service.dart > EntriesService
11 abstract class EntriesService {
55 }
56
57 /// Firestore implementation of EntriesService API
58 class FirestoreEntriesService implements EntriesService {
59   /// creates journal entry in firebase
60   @override
61   Future<EntryId> createEntry({
62     required String userId,
63     required String text,
64     required DateTime creationDate,
65   }) async {
66     final creationDate = DateTime.now();
67
68     /// creates journal entry in firebase
69     final firebaseEntry =
70       await FirebaseFirestore.instance.collection(userId).add(
71         {
72           _DocumentKeys.text: text,
73           _DocumentKeys.creationDate: creationDate.toIso8601String(),
74           _DocumentKeys.isDone: false,
75           _DocumentKeys.hasImage: false,
76         },
77       );
78
79     return firebaseEntry.id;
80   }

```

Images are stored in Firebase Storage using a service to handle file uploads and retrieve download URLs:



```
lib > services > image_actions_service.dart > FirebaseImageService
9  abstract class ImageService {
20 }
21
22 /// upload image to Firebase
23 class FirebaseImageService implements ImageService {
24   @override
25   Future<ImageID?> uploadImageToRemote({
26     required String filePath,
27     required String userId,
28     required String imageId,
29   }) {
30     final file = File(
31       filePath,
32     );
33
34     /// journal entry image upload to FirebaseStorage
35     return FirebaseStorage.instance
36       .ref(userId)
37       .child(imageId)
38       .putFile(file)
39       .then<ImageID?>((_) => imageId)
40       .catchError((_) => null);
41   }
}
```

Access to external services

The app integrates with Firebase for backend services. Configuration is handled in `firebase_options.dart`, full file listing provided here to avoid publishing in GitHub as API keys contained. File is generated using FlutterFire CLI utility (not required to build project).

For project to be built successfully - `firebase_options.dart` to be located in “lib” folder, on same level as “main.dart”:



```
IU_DLBCSEMSE02_TASK3_JOURNAL_APP
> .dart_tool
> .vscode
> android
> build
> doc
< lib
  > auth
  > dialogs
  > extensions
  > loading
  > services
  > state
  > views
  firebase_options.dart
  main.dart
  > test
```

Full listing of file firebase_options.dart:

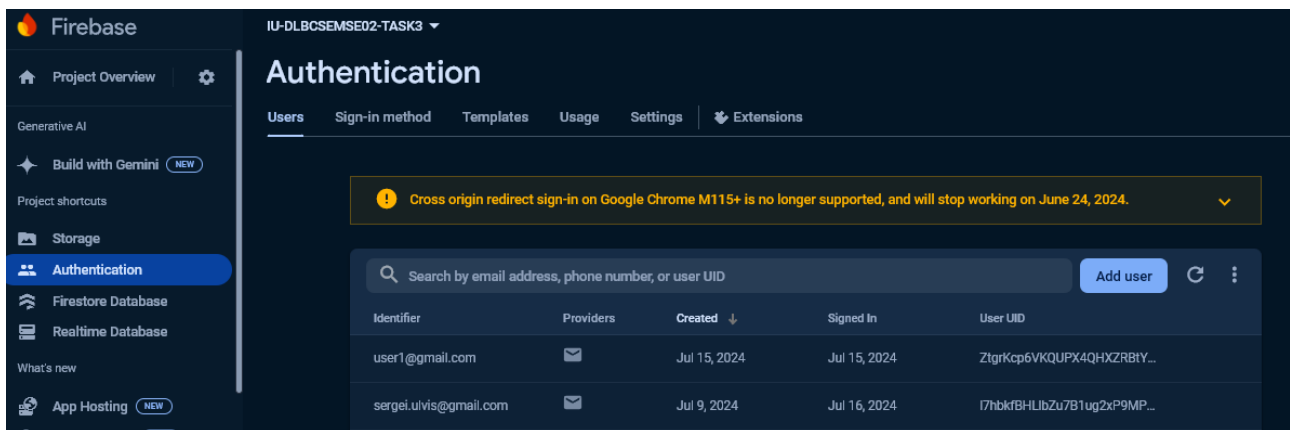
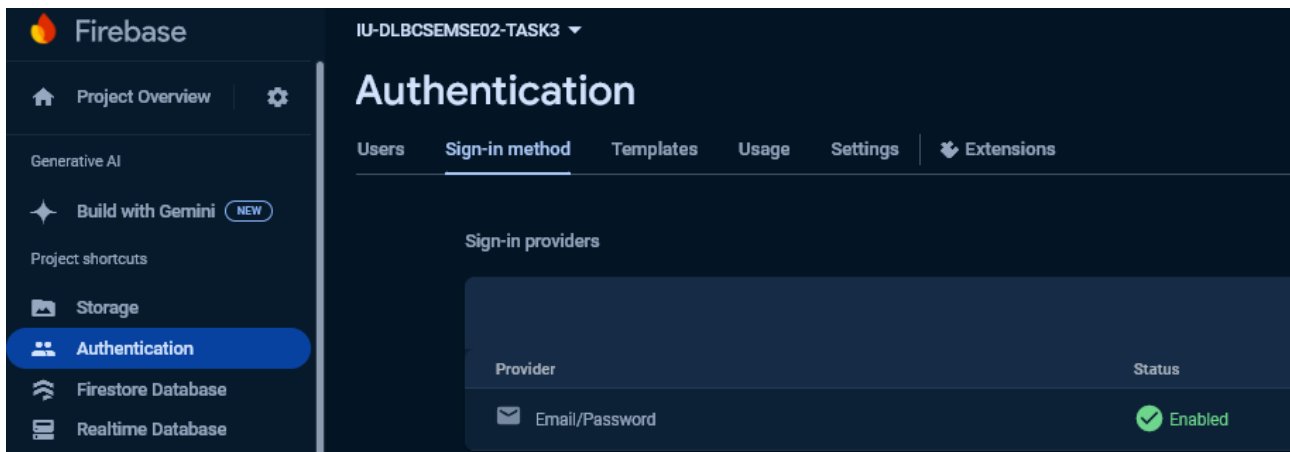
```
import 'package:firebase_core/firebase_core.dart' show FirebaseOptions;
import 'package:flutter/foundation.dart'
  show defaultTargetPlatform, kIsWeb, TargetPlatform;

class DefaultFirebaseOptions {
  static FirebaseOptions get currentPlatform {
    if (kIsWeb) {
      return web;
    }
    switch (defaultTargetPlatform) {
      case TargetPlatform.android:
        return android;
      case TargetPlatform.iOS:
        throw UnsupportedError(
          'DefaultFirebaseOptions have not been configured for ios - '
          'you can reconfigure this by running the FlutterFire CLI again.',
        );
      case TargetPlatform.macOS:
        throw UnsupportedError(
          'DefaultFirebaseOptions have not been configured for macos - '
          'you can reconfigure this by running the FlutterFire CLI again.',
        );
      case TargetPlatform.windows:
        throw UnsupportedError(
          'DefaultFirebaseOptions have not been configured for windows - '
          'you can reconfigure this by running the FlutterFire CLI again.',
        );
      case TargetPlatform.linux:
        throw UnsupportedError(
          'DefaultFirebaseOptions have not been configured for linux - '
          'you can reconfigure this by running the FlutterFire CLI again.',
        );
      default:
        throw UnsupportedError(
          'DefaultFirebaseOptions are not supported for this platform.',
        );
    }
  }
}

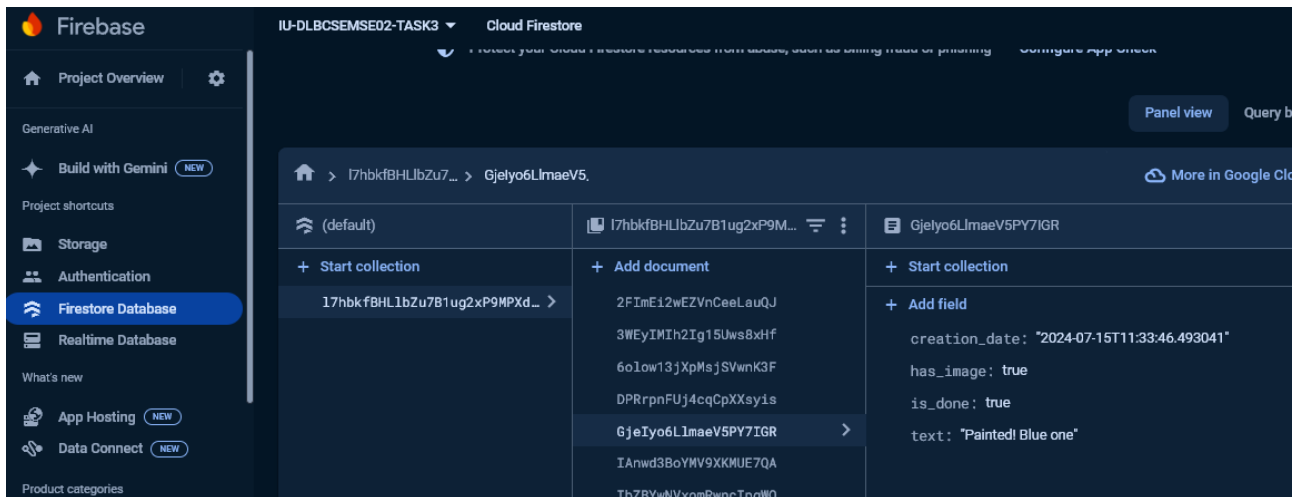
static const FirebaseOptions web = FirebaseOptions(
  apiKey: 'AIzaSyDHjZolH54QdsEEqpd9_akgiMl7yaa7joY',
  appId: '1:889423538419:web:5922b709516f9d8b142f86',
  messagingSenderId: '889423538419',
  projectId: 'w180-mobx-rem',
  authDomain: 'w180-mobx-rem.firebaseio.com',
  storageBucket: 'w180-mobx-rem.appspot.com',
);
```

```
static const FirebaseOptions android = FirebaseOptions(
    apiKey: 'AIzaSyAF3t8pxYYRv_k2_n0ybV7C1CayAkRjcj8',
    appId: '1:889423538419:android:9aebd12112aacede142f86',
    messagingSenderId: '889423538419',
    projectId: 'w180-mobx-rem',
    storageBucket: 'w180-mobx-rem.appspot.com',
);
}
```

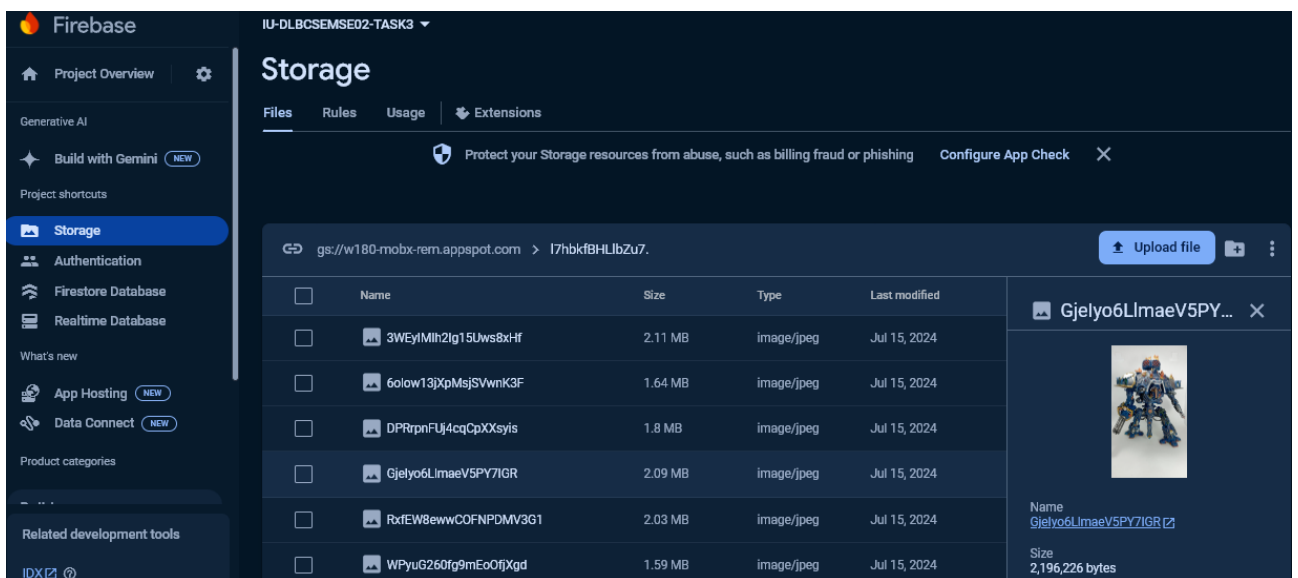
Email authentication enabled in Firebase, however without confirmation link being sent, so any email/password pair may be used to register and log in application.



Firestore database used for Entry texts/done attribute storage:



And Firebase Storage used for files storing, respectively:



Firestore interface does not show the link between entry document in Firestore and file object in Storage, however it exists, i.e. image getter in `entries_service.dart` receives image data based on

```
/// image object getter from FirebaseStorage
@override
Future<Uint8List?> getEntryImage({
  required EntryId entryId,
  required String userId,
}) async {
  try {
    final ref = FirebaseStorage.instance.ref(userId).child(entryId);
    final data = await ref.getData();
    return data;
  } catch (_) {
    return null;
  }
}
```

user and entry IDs:

Testing approach

Testing approach includes unit tests and the use of mock services to isolate and test individual components. “mocks” folder contains services mocks imitating real services, utils.dart contains test-aligned supporting utilities, and app_state_test.dart contains tests.

In Dart, every test is the “test()” function which creates a new test case with the given description (the string) and body, which contains actual test code. In test code, initially set up made to set the initial conditions or inputs for test, than tested action performed, than expect() function used to verify that the result is equal to expected, if the result is not as expected - the test will fail. Tests are started by “flutter test” command in terminal.

```
/// testing entries creation
Run | Debug
test(
  'creating entries',
  () async {
    await appState.initialize();
    const text = 'text';

    /// creating new entry and returning operation result
    final didCreate = await appState.createEntry(
      text,
    );

    /// checking operation result == true (entry created)
    didCreate.expectTrue();

    /// checking entries list length incremented by 1
    expect(
      appState.entries.length,
      mockEntry.length + 1,
    );

    /// checking new entry text equal to test text
    final testEntry = appState.entries.firstWhere(
      (element) => element.id == mockEntryId,
    );
    expect(
      testEntry.text,
      text,
    );

    /// checking new entry isDone == false
    testEntry.isDone.expectFalse();
  },
);
```

Tests successfully passed:

```
PS H:\YandexDisk\flutter_projects\iu_dlbsemse02_task3_journal_app> flutter test
00:16 +9: All tests passed!
```

Project documentation

Commentaries are provided in code and code documentation created using Dart documentation generator (<https://pub.dev/packages/dartdoc>), using “dart doc” command.

Opening index.html available in iu_dlbsemse02_task3_journal_app\doc\api\ will show HTML-based tree containing project documentation in structured view:

iu_dlbsemse02_task...
package

Libraries

LIBRARIES

state\app_state

auth\auth_error

services\auth_service

dialogs\delete_account_...

dialogs\delete_entry_dial...

views\entries_list_view

services\entries_service

views\entries_view

state\entry

firebase_options

dialogs\generic_dialog

extensions\if_debugging

services\image_actions_...

loading\loading_screen

loading\loading_screen_...

views\login_view

dialogs\logout_dialog

state\app_state

auth\auth_error

services\auth_service

dialogs\delete_account_dialog

dialogs\delete_entry_dialog

views\entries_list_view

services\entries_service

views\entries_view

state\entry

firebase_options

dialogs\generic_dialog

extensions\if_debugging

services\image_actions_service

Critical evaluation and possible improvements

The app currently meets the basic functionality requirements, including user authentication, navigation, and data handling. However, several improvements can be made:

- enhanced user interface: Implementing more interactive and responsive UI elements
- error handling: more robust error handling mechanisms for network requests and user inputs.
- additional entry fields may be added
- email verification process may be established on Firebase side to enhance security.

Lessons learned

- effective state management with MobX: utilizing MobX state management solution can significantly streamline data flow and UI updates comparing with Flutter vanilla setState() approach
- MobX boilerplate code generation with build_runner tool
- modular architecture: organizing code into clear, modular components improves maintainability and scalability, with separation of view, services and underlying implementation
- Firebase usage as backend with authentication and text/binary data storing
- automated testing and project documentation generation.

Bibliography

Google Developers (2024). Design for Android. <https://developer.android.com/design>

t2informatik (2024): Wireframing. <https://t2informatik.de/en/smartpedia/wireframing/?noredirect=en-US>

GitHub (2024). GitHub Docs. Repositories. <https://docs.github.com/en/repositories>

MobX.dart (2024). MobX for Dart / Flutter. <https://mobx.netlify.app>

Pub.dev (2024). MobX package. <https://pub.dev/packages/mobx>

Medium.com (2024). Flutter Dart Documentation. <https://medium.com/codex/flutter-dart-documentation-791371ff2e0f>

Medium.com (2024). The Complete Guide to Flutter App Testing with Examples. <https://medium.com/@dihsar/the-complete-guide-to-flutter-app-testing-with-examples-4ed7b8188bce>

Firebase Documentation (2024). Add Firebase to your Flutter app. <https://firebase.google.com/docs/flutter/setup?platform=android>