**MOBILE PROJECT**

**Main:**

at first this code responsible for Application setup Integrates with Firebase, handle login and screen transitions

**Most important Point in code:**

* Firebase Initialization

Firebase is initialized using project data such as apiKey, authDomain, and projectId.

The code attempts to initialize Firebase using try and catch:

If initialization succeeds, the app is launched using MyApp.

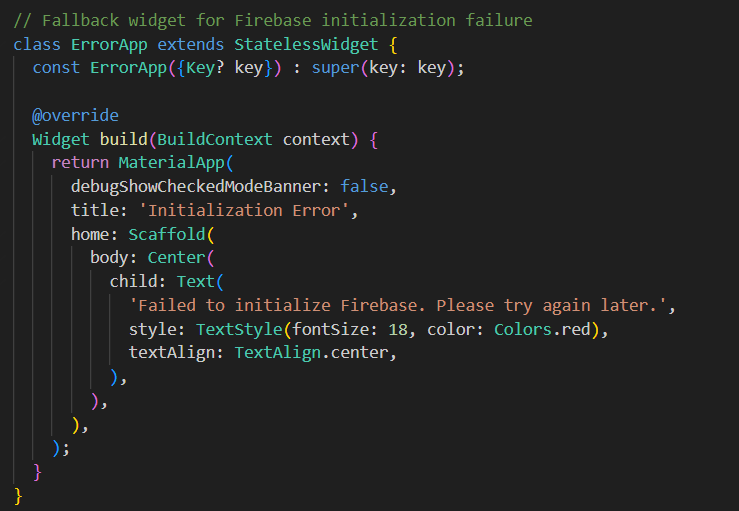
If initialization fails, an ErrorApp screen is displayed



* ErrorApp Widget

If there is a problem while setting up Firebase, a message is displayed to the user saying: "Failed to initialize Firebase. Please try again later.

This is done using a simple widget with text in the middle of the screen.



* MyApp Widget

This is the main application.

Uses ChangeNotifierProvider to provide login credentials via AuthService.

Relies on Routing to determine which screen to display based on the user's status:

If the user is logged in, HomeScreen is displayed.

If not logged in, LoginScreen is displayed.

* Routes

The code specifies the routes the application can go to:

/ (home): Specifies whether the user is logged in or not.

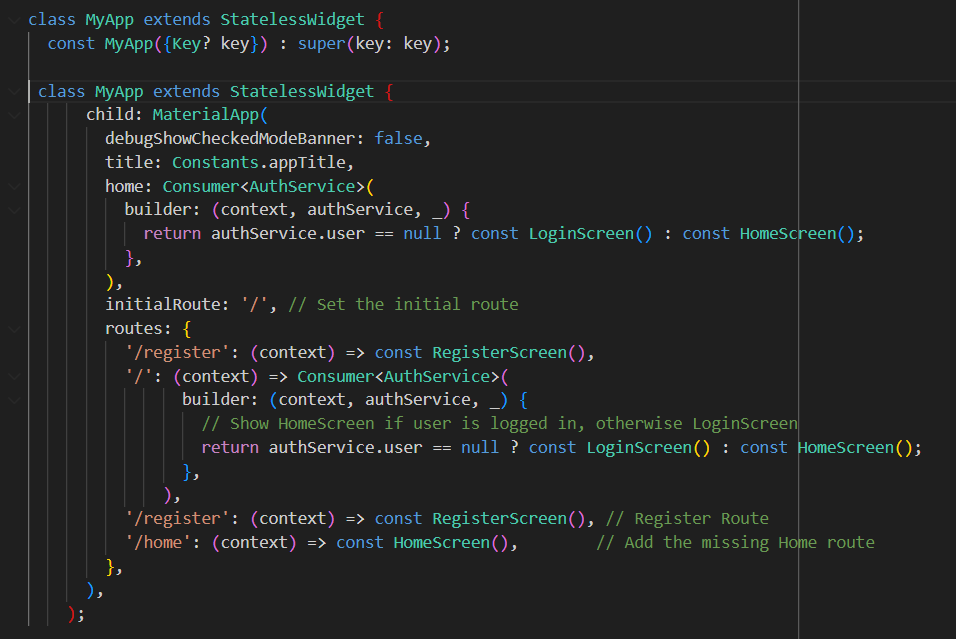
/register: Takes the user to the registration screen.

/home: Displays the home screen.

* Additional improvements:

Added the missing /home path.

Use Consumer<AuthService> to dynamically update screens based on user status



***Home Screen:***

The code represents a HomeScreen interface within a Flutter application. In this screen, tasks and related services are handled, in addition to managing logout.

**Most important Point in code:**

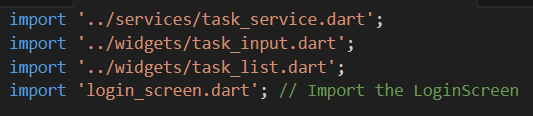
* Import services and widgets:

task\_service.dart: Contains services related to task management.

task\_input.dart: Widget for entering new tasks.

task\_list.dart: Widget displaying the list of tasks.

login\_screen.dart: Login interface.



* Bring services

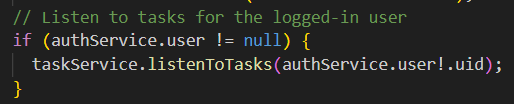
AuthService is used to manage the login state of the user.

TaskService is used to manage tasks (Fetch/Listen).



* Update tasks

If the user is logged in, the application listens for changes in the user's task list via listenToTasks.



* Home screen structure

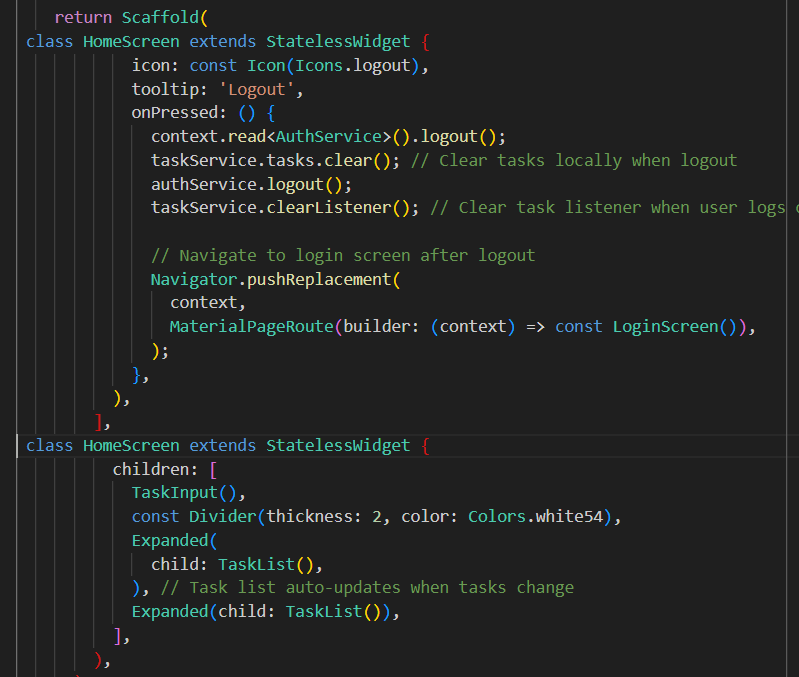
AppBar Contains a Logout button:

Calls logout in AuthService to end the session.

Clears local tasks via taskService.clearListener

TaskInput: Widget to add new tasks.

TaskList: Displays the list of tasks, and automatically updates when tasks are changed



**AuthService**:  
The code represents part of the AuthService for managing login, logout and account creation using Firebase Authentication in a Flutter app.

**Most important Point in code:**

* user property

Returns the current user (if logged in).

User data is obtained through Firebase Authentication via the currentUser property.



* login

Log in with email and password.

What happens inside:

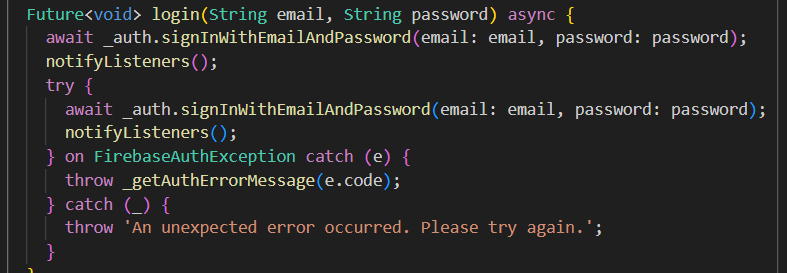
Uses Firebase's signInWithEmailAndPassword to log in.

If the login is successful, notifyListeners are called to notify the connected interfaces of the status update.

If an error occurs:

The error type is checked with FirebaseAuthException.

A clear error message is returned using the \_getAuthErrorMessage function



* register

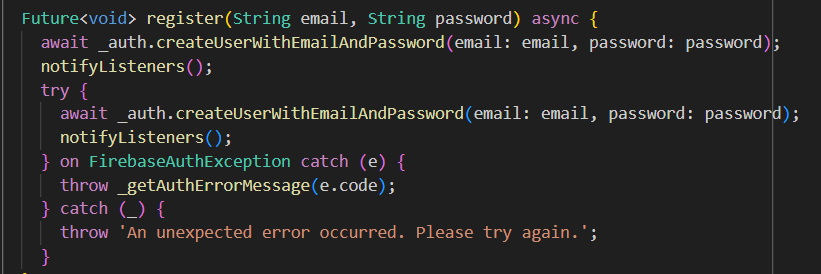
Create a new account using email and password.

What happens inside:

Uses createUserWithEmailAndPassword to create the account.

If the creation is successful, notifyListeners is called to update the status.

If an error occurs, it is handled the same way as for login



* logout

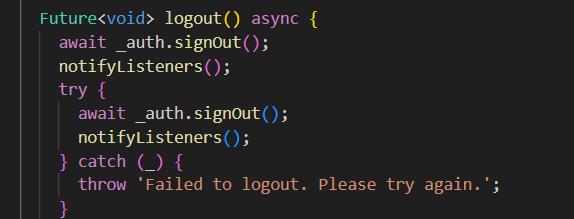
Logs out the current user.

What happens inside:

Uses signOut from Firebase to log out.

NotifyListeners are called to update the app state after logout.

If an error occurs, a custom error message is returned.



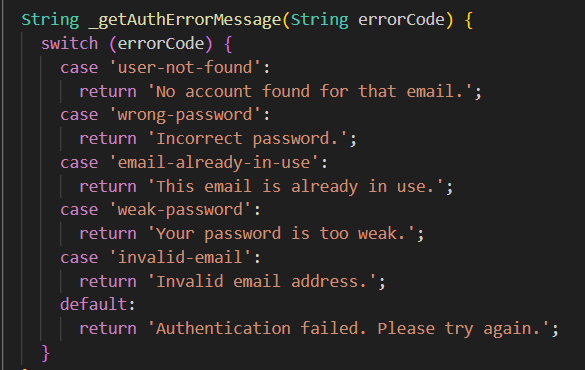
* getAuthErrorMessage

Converts the error code sent by Firebase into a message that is understandable and clear to the user.

How it works:

Uses a switch to match the error code with the appropriate message.

If the error code is not known, a generic error message is returned.



**What are the features of this code?**

Proper error management:

Error codes are converted into clear messages for the user.

Separation of responsibilities:

Each operation (login, logout, create account) has an independent function

Using notifyListeners:

Informs associated interfaces of any changes in authentication status

Task service:

The code represents a TaskService that relies on Firebase Firestore to store and retrieve tasks in a Flutter app. It also contains a Task model for managing task data.

**Most important Point in code:**

* TaskService

The code represents a TaskService that relies on Firebase Firestore to store and retrieve tasks in a Flutter app. It also contains a Task model for managing task data.

1. fetchTasks

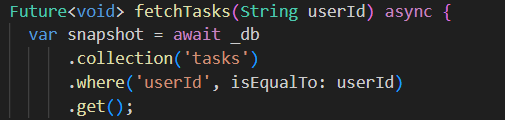
Fetch tasks from Firestore based on userId.

What happens inside:

Documents are fetched using a condition that specifies that the userId field matches the userId.

Documents are converted to Task objects using the Task.fromFirestore function.

The tasks list is updated locally and the associated interfaces are notified of the change using notifyListeners





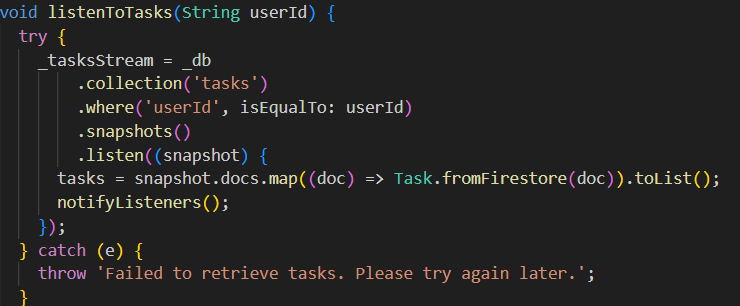
1. listenToTasks

Track any changes to the task list in real time.

What happens inside:

Firestore snapshots are used to get continuous updates when data changes.

The task list is updated on each change, and the associated interfaces are notified



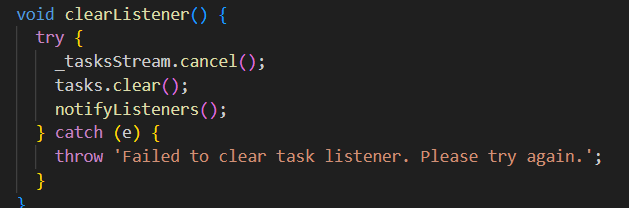
1. clearListener

Unsubscribe from Firestore updates when you log out.

What happens inside:

Unsubscribe using \_tasksStream.cancel().

The task list is cleared locally

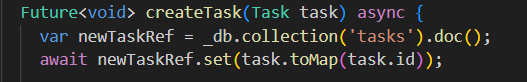


1. createTask

Create a new task and save it to Firestore.

What happens inside:

A new document is created using set and the task object is converted to a suitable format using toMap

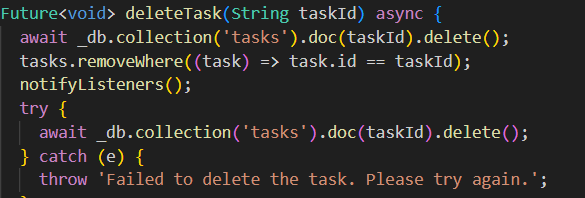


1. deleteTask

Delete a task from Firestore.

What happens inside:

The document is deleted based on the task ID.

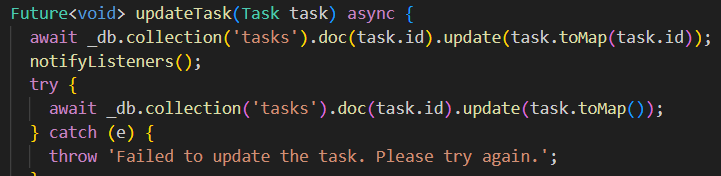


1. updateTask

Update an existing important detail.

What happens inside:

The document is updated using update and the data is converted to a suitable format using toMap.



* Task Model

id: Task ID.

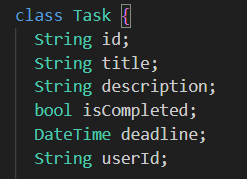
title: Task title.

description: Task description.

isCompleted: Task status (completed or not).

deadline: Deadline.

userId: User ID who created the task

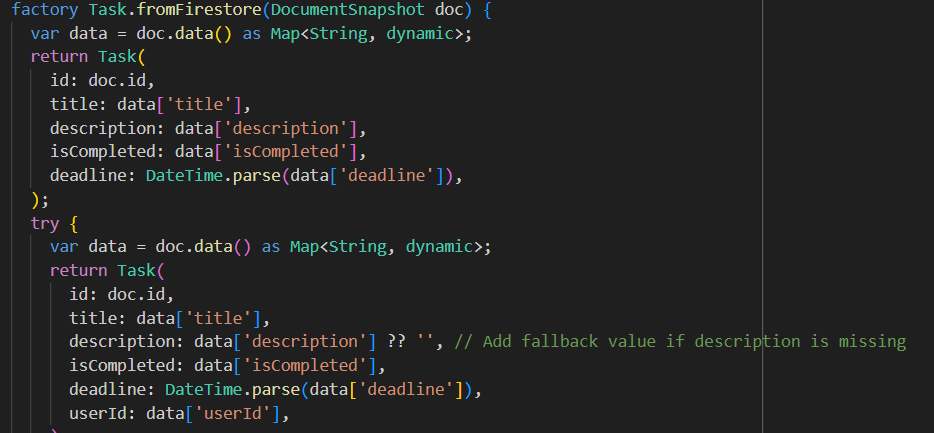


* fromFirestore

Converts data coming from Firestore into a Task object.

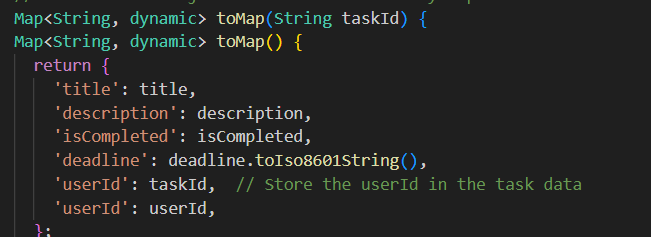
What happens inside:

The document is parsed into a Map and the Constructor is called to create the object



* toMap

Convert the task object to a format suitable for storing in Firestore.



Task input

The code shows a new task entry interface in a Flutter app using a Provider to manage state, interacting with AuthService (to manage authentication) and TaskService (to manage tasks) services.

**Most important Point in code:**

* TaskInput

Basic Structure:

TaskInput: A UI element of type StatefulWidget.

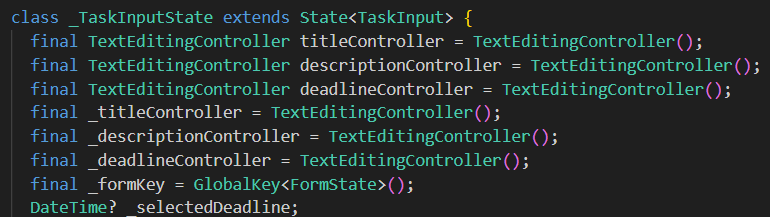
TaskInputState: A TaskInput state that manages the input and form state.

Variables:

\_formKey: A key for the form to validate the fields.

titleController, \_descriptionController, \_deadlineController: Text controls inside the input fields.

selectedDeadline: Stores the user-selected deadline



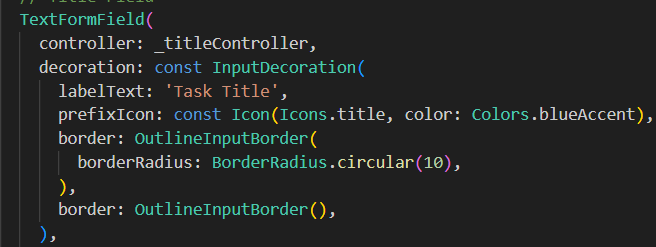
* Field entry interface

1. Title

Enter the task title.

Verification:

Make sure the field is not empty

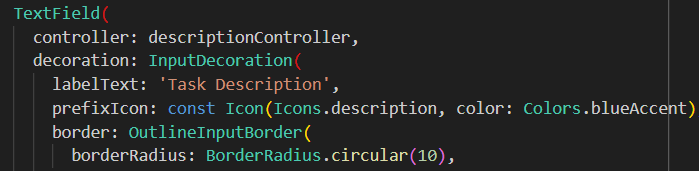


2. Description

Enter a task description.

Verification:

Make sure the field is not empty.



3. Deadline

Select a deadline using DatePicker.

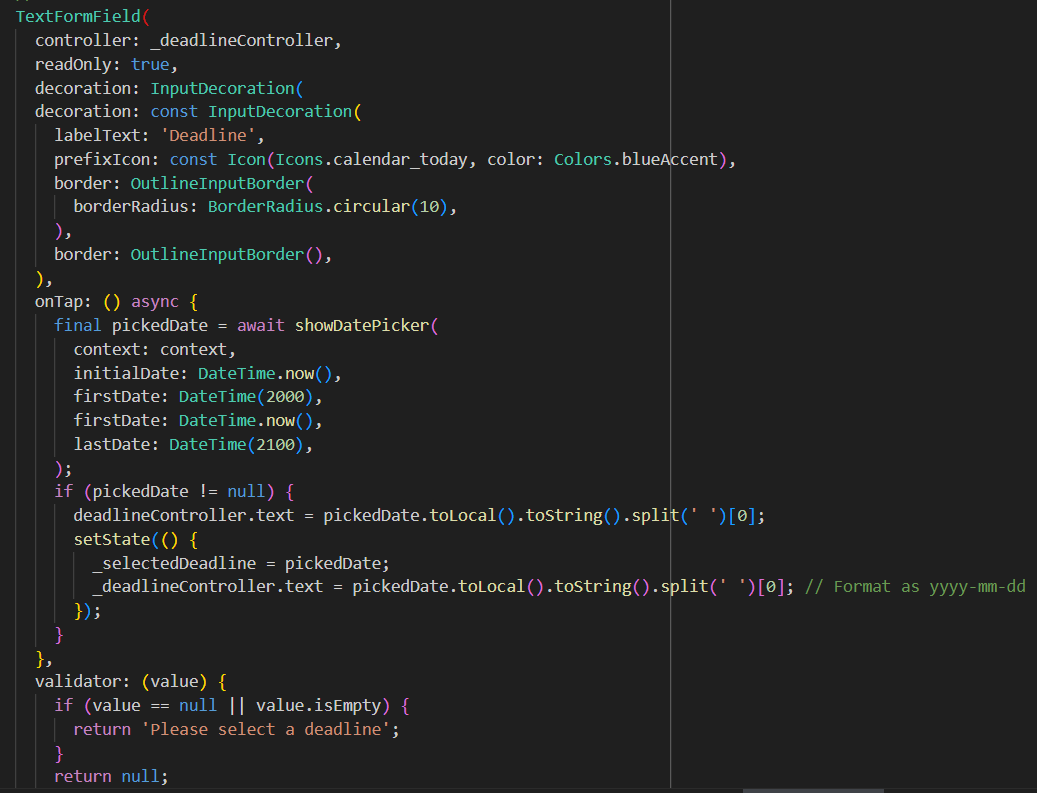
What happens inside:

When you click on the field, the date selection window is displayed.

The date is stored in the \_selectedDeadline variable.

Verification:

Making sure a deadline is selected



* Add task button

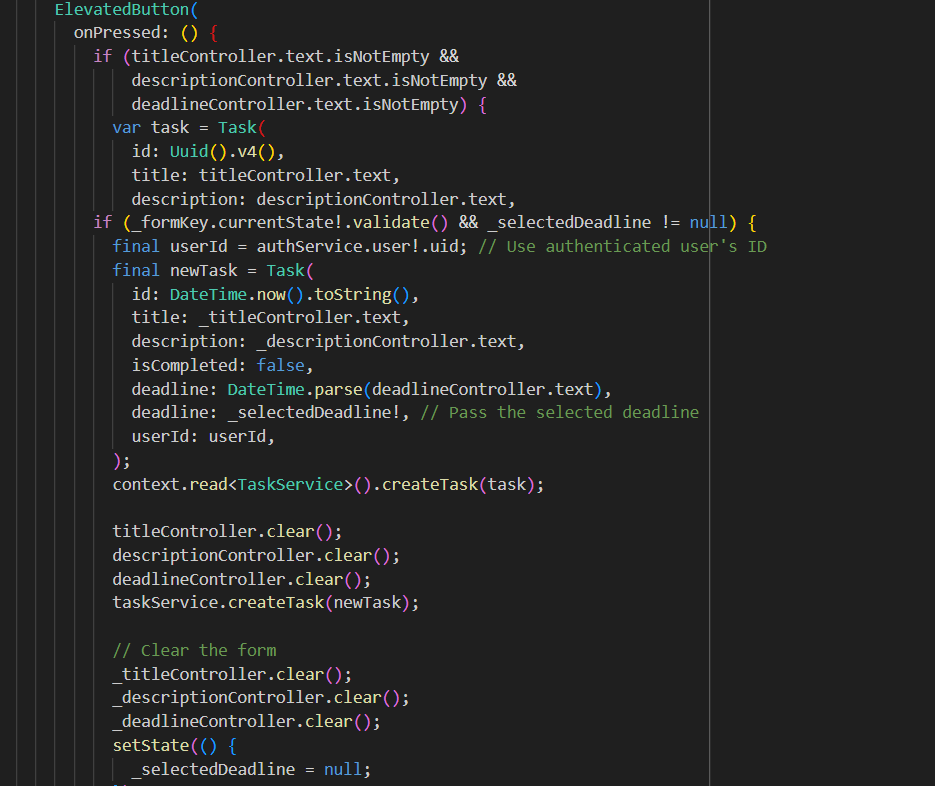
Validate fields and add task.

What happens inside:

Validate fields using \_formKey.

Call createTask from TaskService to create a new task.

Clear fields after adding task



Task list:

The above code is a TaskList widget that uses ListView.builder to display the list of tasks stored inside the TaskService. The widget relies on Provider to get data and manage state.

#### Provider

#### context.watch<TaskService>().tasks:

#### Gets the current list of tasks from the TaskService class.

#### Tasks are automatically updated when any data changes thanks to the Provider mechanism.

#### 

#### Display message if no tasks are available

#### Checks if the task list is empty. If empty, displays an alert message in the middle of the screen using Center.

#### 

#### Use ListView.builder to display a list of tasks.

#### ListView.builder: Used to create a scrollable list of items (tasks) that are dynamically generated.

#### itemCount: Number of items in the list (equal to the number of tasks).

#### itemBuilder: Implemented to create a UI element for each task.

#### 

#### Design each item in the list using Card.

#### UI element to design each task using shadow and spacing effect.

#### Spacing is customized using margin.

#### 

#### Task content inside ListTile

#### Displays the title of the task.

#### If the task is complete (isCompleted) the title is shown with a line (strikethrough).

#### subtitle: Displays the description and deadline using a Column to display the texts vertically.

#### trailing: Contains a delete button (IconButton).

#### When pressed, the task is deleted using the deleteTask function of the TaskService.

#### onTap: Used to update the completion status of the task (isCompleted).

#### The task is updated by calling updateTask.

#### How is the widget used?

#### 