

ALF Meets Flutter



START YOUR FLUTTER JOURNEY WITH ALF

ALF stands for Application Layers Framework and is designed to standardize the implementation of Flutter applications by promoting a clean architecture. It achieves this by organizing application logic into distinct layers hence the name. But there is more to the name...

If you find this framework useful, please kindly consider supporting this project by making a donation via my Ko-fi page (https://ko-fi.com/flutter_alf). Even a small gesture can be a great source of motivation.

TABLE OF CONTENT

1	INTRODUCTION
2	BUILDING THE FRAMEWORK
2.1	REQUIREMENTS
2.2	DESIGN
2.3	IMPLEMENTATION
2.4	SHOWCASE
3	INTERNATIONALIZATION
4	COMMON APPLICATION CLASSES
5	CENTRALIZED APPLICATION CONFIGURATIONS
6	CONSOLIDATED PACKAGE IMPORT
7	STRUCTURING YOUR FLUTTER PROJECT
8	NAVIGATION AND ROUTING
8.1	REQUIREMENTS AND EVALUATION
8.2	INTEGRATION WITH ALF
9	FIELD AND PAGE VALIDATION
10	NOTIFICATION AND MESSAGING
11	FINAL WORDS
	APPENDIX – HELP AND FAQ

1 INTRODUCTION

You may have noticed that many Flutter tutorials on the popular video streaming platform focus on building mobile applications with visually appealing UIs. These tutorials often feature coding sessions that guide you through widget layout techniques, ensuring a pleasant user experience. Some even recreate the UIs of well-known applications from scratch to demonstrate that Flutter can achieve similar results.

However, while these tutorials showcase impressive UI designs, they typically lack real business logic. In practice, a well-architected application is structured into multiple layers, such as the presentation layer, domain layer, and data layer, to ensure maintainability and scalability.

This is where the Application Layers Framework (ALF) comes in. ALF was developed to standardize the design and implementation of these layers, providing a clean, scalable, extendable, and maintainable architecture. By following this framework, developers can avoid the pitfalls of disorganized, spaghetti code and build robust applications with a solid foundation.

Using this framework will have the following benefits:

- Clear separation of concerns
- Code reusability
- Easier code maintenance
- Form different development teams responsible for different areas
 - UI front-end and UI navigation
 - Business logic interfacing with backend service providers
 - Backend service providers and underlying platforms

We will begin by defining the key requirements that the Application Layers Framework aims to address. Then, we will walk through the implementation of a small sample application to demonstrate the framework in action. Additionally, we will explore how other essential requirements, though not strictly part of the core application layers, can integrate with ALF, allowing for future extensibility.

While frameworks and architectural patterns are valuable, excessive complexity can make applications harder to understand and maintain, requiring highly skilled developers. ALF is designed to strike a balance between structure and simplicity, ensuring a standardized approach to application architecture while keeping implementation straightforward and accessible.

TARGET AUDIENCE

This framework is intended for developers who are already familiar with software development using architectural patterns and have experience building simple Flutter applications.

- **New to application architecture patterns?** You may want to read **Unleashing Creativity: Exploring Architecture Patterns in Flutter** (<https://medium.com/@samra.sajjad0001/unleashing-creativity-exploring-architecture-patterns-in-flutter-12b7465bc927>) for foundational insights.
- **Just starting with Flutter?** We recommend checking out Google's **Flutter code lab** (<https://codelabs.developers.google.com/codelabs/flutter-codelab-first>), a great starting point for learning Flutter basics. Once you complete that tutorial, you can set aside its approach to state management and application architecture, ALF will provide a structured alternative.

Much effort has gone into developing this framework, and we welcome constructive feedback to refine and enhance ALF, contributing to the broader Flutter community.

More on the name...

The name ALF was inspired by the classic ALF TV series from the 1980s, which featured an adorable extraterrestrial character. The show was undeniably great, and its acronym, Alien Life Form, felt like the perfect fit for this framework. You can find highlights from the series on the popular video streaming platform if you would like to revisit its charm.



2 BUILDING THE FRAMEWORK

If you are new to Flutter, it is helpful to start with basic tutorials that guide you through creating a simple proof-of-concept (POC) application. These tutorials typically demonstrate the fundamental steps of building a mobile app using Flutter.

The term “POC” is intentional because most introductory tutorials focus on just one or two screens interacting with each other, often mixing presentation, controller logic, and data within a single codebase. While this approach is useful for quick experimentation, it is not suitable for production-ready applications. POC applications are typically built in a “quick and dirty” manner to validate technologies but are discarded afterward. Their structure does not support scalability or the addition of new features that meet real-world user requirements.

Our goal is to create an application that is reusable and scalable, ensuring that as more features are added, the implementation remains consistent and easy to understand and maintain. To achieve this, it is crucial to architect the application properly and define clear application layers before writing any code.

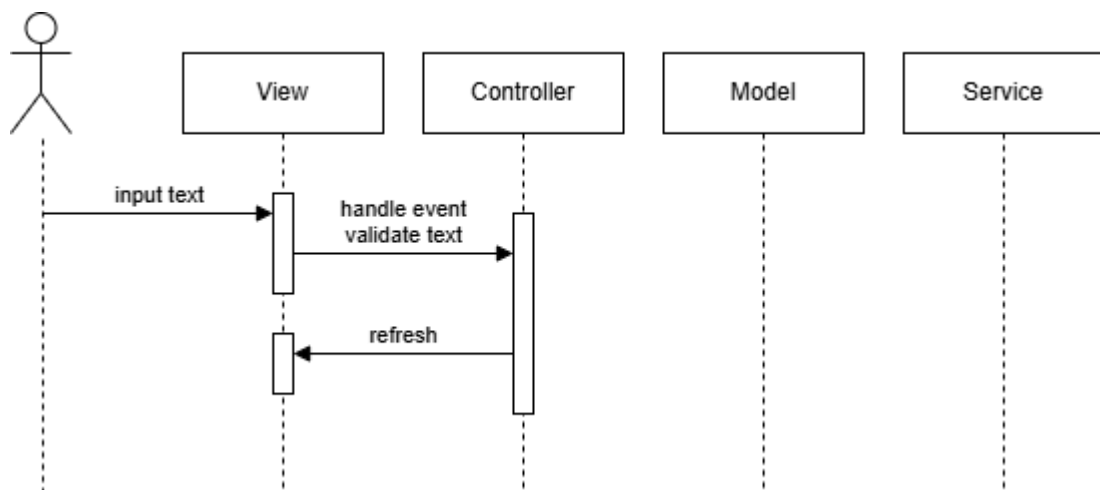
If you would prefer to see a working sample application first, you can skip ahead to chapter 2.4 (Showcase), and then return to these foundational concepts, from defining requirements to designing and implementing the framework.

2.1 REQUIREMENTS

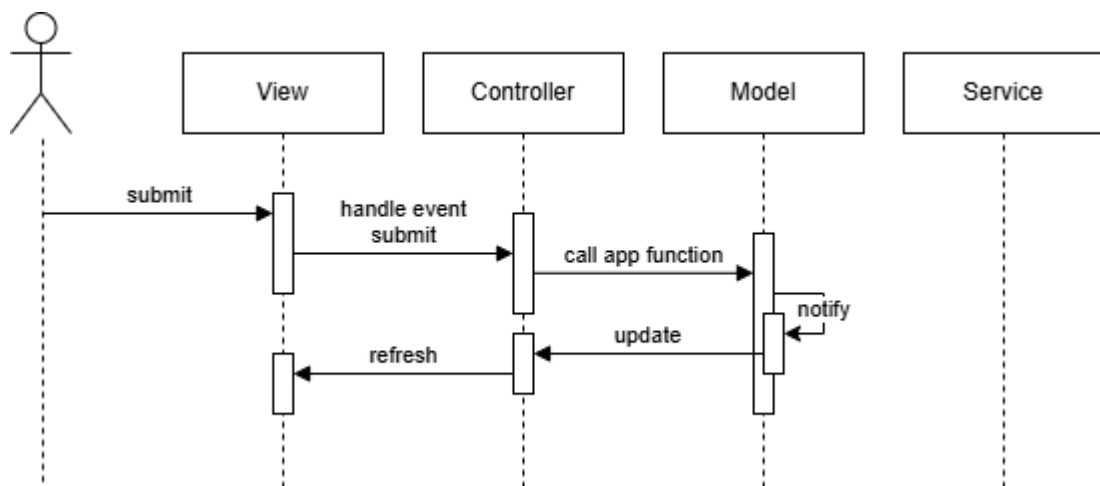
A fundamental architectural design pattern that should be implemented in all user-facing applications is the separation of logic for display, control, and data. This approach is commonly known as the Model-View-Controller (MVC) design pattern, which exists in several variations. However, the core principle remains the same which is separating application logic from presentation and control.

There are three common scenarios in which a user interacts with an application:

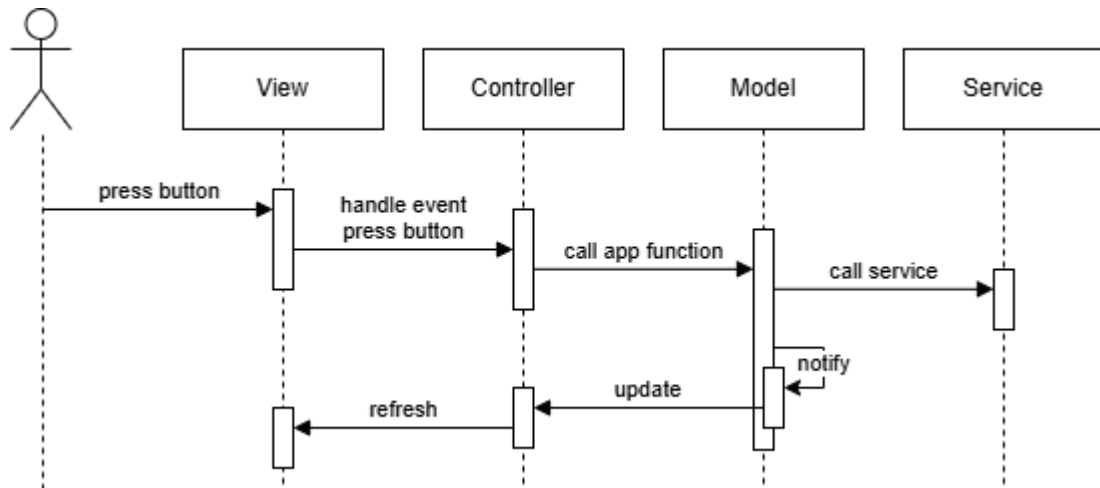
1. **User Input Validation** The page (or view) is presented to the user, prompting for input. As the user updates the input (e.g., an email address), the controller validates its correctness. The interaction flows back and forth between the view and the controller until a valid input is received and ready for submission (e.g., user login). At this stage, the model and service components are not yet involved. This interaction is illustrated in the sequence diagram below.



2. **Processing User Input and Updating the View** Once the input is validated, the page allows submission to the model instance, which executes the intended application function. The model instance exists throughout the lifecycle of the application or a specific module. After processing, the model generates new data, notifying the controller, which then instructs the view to refresh the page. This flow is depicted in the sequence diagram below.



3. **Model Interaction with Service Providers** The model instance persists across the application or a specific module, providing application functions that the controller can call. Additionally, it can leverage service providers to perform backend tasks (e.g., local/remote storage, API services). These service providers abstract backend interactions, ensuring that the application's interface remains unchanged even if the backend implementation changes. The sequence diagram below illustrates how the model interacts with service providers.



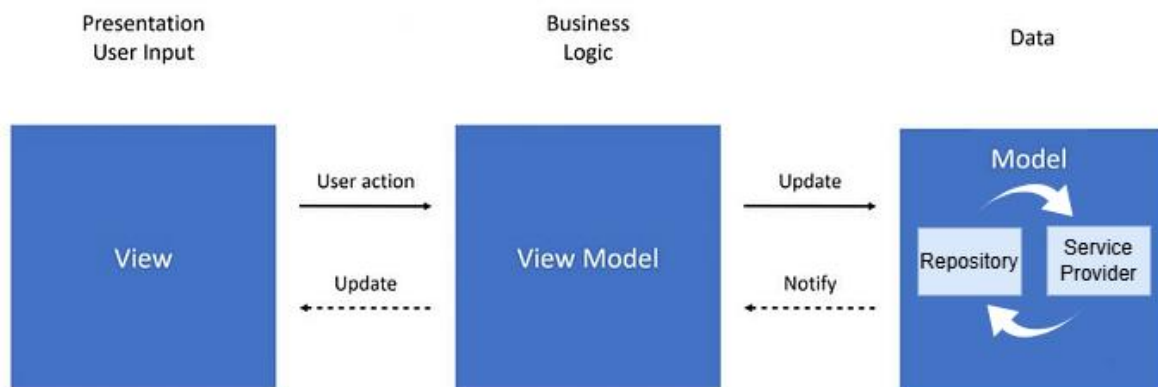
From these three scenarios, two key architectural requirements emerge:

1. **Separation of Concerns** – Clearly defining the roles of the view, controller, model, and service provider.
2. **State Management** – Ensuring model instances are properly managed across the application or within specific modules.

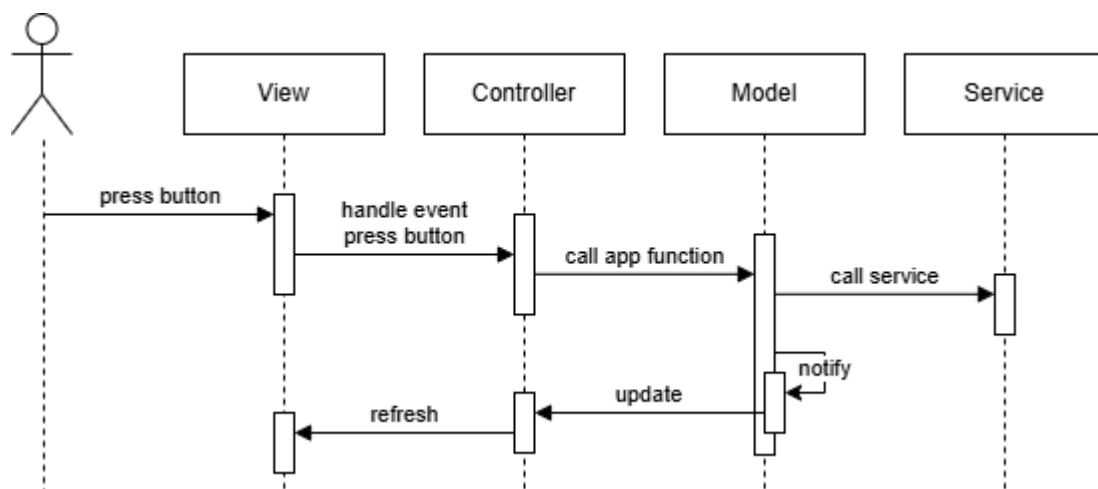
Flutter offers several state management libraries, including GetX, BLoC, or Riverpod, each with its own strengths. While we could have chosen any of these as the foundation for ALF, they may not fully meet above requirements or introduce complexity due to extensive features and options. To maintain full control over the framework and establish a standardized architecture with clear design guidelines, we developed ALF from scratch, ensuring that these requirements are fully addressed.

2.2 DESIGN

A variation of the MVC architecture pattern, as discussed in **Unleashing Creativity: Exploring Architecture Patterns in Flutter** (<https://medium.com/@samra.sajjad0001/unleashing-creativity-exploring-architecture-patterns-in-flutter-12b7465bc927>), proposes separating the view, controller, and model while further breaking down the model into a repository and a service provider.



This variation of MVC architecture closely aligns with the third scenario outlined in the previous section, which covers the full flow from user input to backend interaction.



It integrates well with the application layers we previously defined as part of the framework requirements. To understand how this translates into a Flutter implementation, we can map the MVC components to the Application Layers Framework, as illustrated in the following table.

No	MVC Component	ALF Component
1	View	View class
2	User action	Handle event by controller
3	Update (from View Model to View)	View state as Dictionary
4	View Model	Controller class
5	Update (from View Model to Model)	Call repository
6	Notify	Notify controller
7	Repository (from Model)	Repository class
8	Service Provider (from Model)	Service provider class

Under the hood, the Application Layers Framework leverages a stateful widget to refresh the UI and a stream controller to manage events, with their implementation details abstracted by the framework. As an application developer, the primary focus is on coding the view and controller and defining how events are handled while backend communication is abstracted through the repository layer.

EVENT-DRIVEN ARCHITECTURE AND STATE MANAGEMENT

User interactions trigger events that are consumed and processed by the controller. The controller determines which functions to call in response to an event and updates the page with new data upon completion. Each field's state within a page is stored in the controller.

Repositories serve as the model layer in the Model-View-Controller (MVC) pattern, existing within the scope of the application or a specific application module. Since a module can span across multiple pages, repositories facilitate data sharing between them. Their lifecycle is tied to the duration of the module, though some repositories may persist throughout the entire application lifecycle (e.g., a session repository).

DECOUPLING BACKEND ACCESS

Repositories handle interactions with backend services or data providers, ensuring that changes to backend implementations do not affect the frontend. If backend providers are stateless, the singleton pattern can be considered for efficiency, though it is not mandatory. What matters is maintaining a distinct application layer for backend access.

DESIGN PRINCIPLES

ALF not only defines a structured application architecture but also enforces consistency in implementation, making applications easier to understand and maintain. To achieve this, the following design principles should be adhered to:

1. State Management via Repositories

- Repositories define the state of an application or module.
- They facilitate data sharing between pages within a module or across the entire application.
- Avoid passing application data via the constructor of page or view classes.

2. Page Implementation

- Pages are to be extended from the ALF view class, which inherits from `StatefulWidget`.
- ALF encapsulates state management separately from the stateful widget, which is only responsible for refreshing the UI when needed.
- For partial view updates independent of ALF, consider using `StatefulBuilder`.

3. Controller Implementation

- Controllers can subscribe to repository changes and trigger events accordingly.

- Each page must have exactly one controller, managing its own events and states.
- 4. Page State Representation**
 - Field names within a page state are defined using an enumerator associated with the controller.
 - The actual state is stored in a dictionary (e.g., text input fields), with field names defined in the enumerator.
 - 5. Static vs. Dynamic Page Display**
 - Most pages have a static layout, with only field values changing.
 - For dynamic elements (e.g., displaying a dialog box after an event), use a “hidden” field in the page state.
 - 6. Event Handling**
 - Any action that modifies the page state (e.g., click button) is to be defined as an event.
 - Each page has an enumerator listing all possible events it can trigger.
 - 7. Entity Grouping**
 - Related attributes within a repository should be grouped into entities.
 - E.g., username, age, gender, email, and phone can form a `UserProfile` entity.
 - 8. Backend Interaction via Service Providers**
 - Always use a service provider to interact with the backend instead of a repository.
 - The service provider encapsulates backend communication details and converts data into entities used by the repository.
 - This ensures that frontend implementation remains unaffected by backend changes.

With these core design principles established, we can now explore how the Application Layers Framework is implemented to meet the requirements defined earlier.

2.3 IMPLEMENTATION

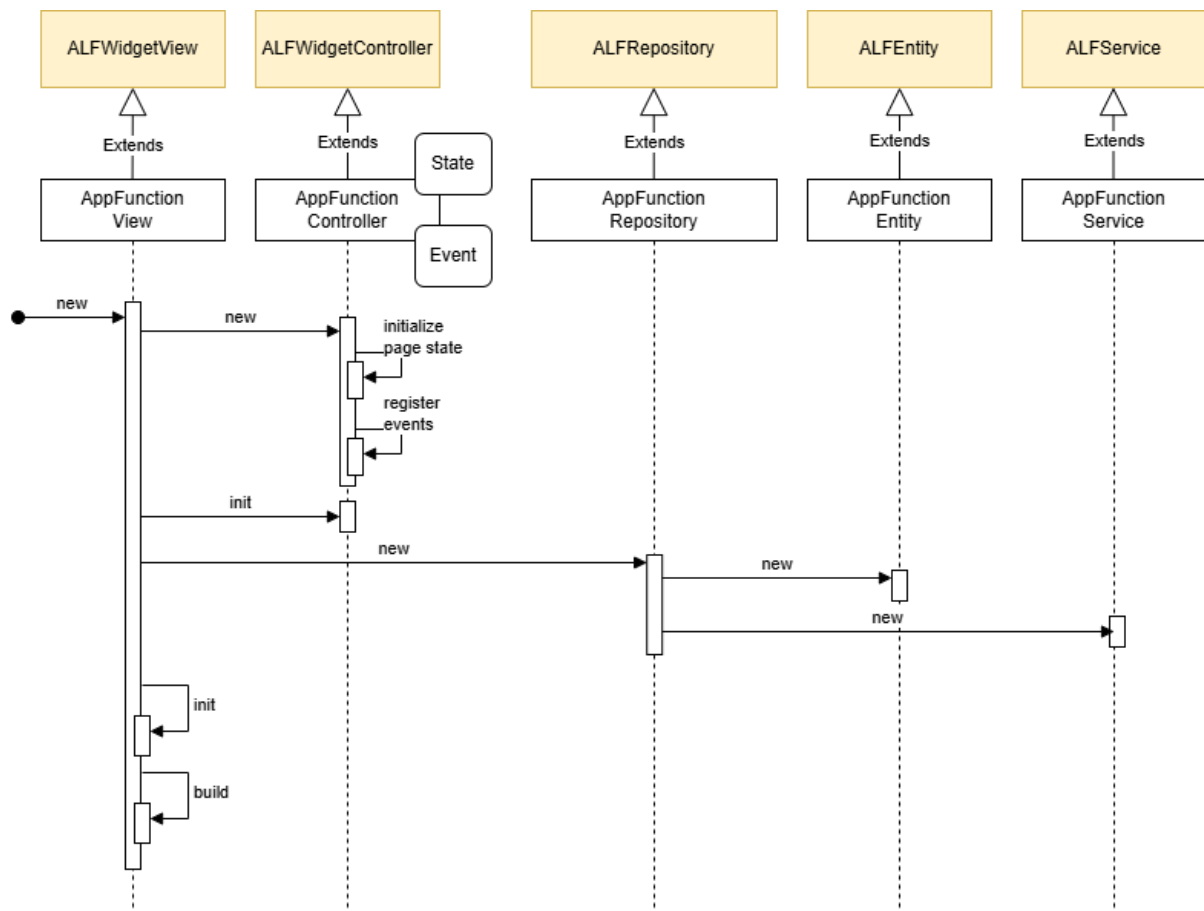
For each of the components mentioned earlier in the design, we will now explore their implementation by examining common scenarios. As part of the naming convention, all classes implemented within the Application Layers Framework are prefixed with **ALF**.

INITIALIZING THE VIEW

A page implementation consists of several key components:

- **Page/View Class** – Defines the visual representation of the page.
- **Controller Class** – Manages interactions and events.
- **State Enumerator** – Specifies different states of the view.
- **Event Enumerator** – Defines events that can be triggered by the page.
- **Repository Class** – Handles application functions and maintains state using entities.
- **Entity Class** – Represents domain objects that store and structure application data.
- **Service Class** – Encapsulates backend communication and provides interfaces for repository interaction.

The sequence diagram below illustrates the high-level process for initializing the page or view.



Whenever a new page is instantiated (extending from `ALFWidgetView`), it is always associated with a controller (inherited from `ALFWidgetController`). This controller receives events triggered by user interactions.

- The initial state is stored in a dictionary and defined when creating the controller.
- This state is used to display initial data on the page and is updated whenever an event is triggered.
- Events are registered with the controller to specify their handling logic.
- An initial event may be triggered upon page initialization, such as loading data from the backend before rendering the page.

Above, a new controller is associated with the view, and an initial event is triggered. A new repository is created and scoped within the application function. The `build()` method is overridden to define the user interface, while the `init()` method can also be overridden to execute logic during page initialization before rendering.

Where necessary, the repository interacts with the backend service provider to retrieve or update data. A repository (inherited from `ALFRepository`) implements application functions or business logic that respond to triggered events. It typically contains entities (inherited from `ALFEntity`) that describe domain objects.

- A combination of entities forms the repository object, which holds application data.
- The repository is usually initialized from the first page of an application module.
- Backend communication is handled via a service class (inherited from `ALFService`), encapsulating the interaction details and exposing interfaces for repository use.

All classes inherit from their respective parent classes within the Application Layers Framework, ensuring that each application layer has the necessary functionalities.

Despite the complexity of the framework implementation, the actual usage is quite straightforward. Below is a code example demonstrating a typical page implementation:

```
class AppFunctionView extends ALFWidgetView {
  AppFunctionView({super.key}) : super(
    controller: AppFunctionController(),
    event: AppFunctionEvent.initialize,
    repositories: [
      ALFRepositoryProvider<AppFunctionRepository>(create: (context) =>
AppFunctionRepository()),
    ]);

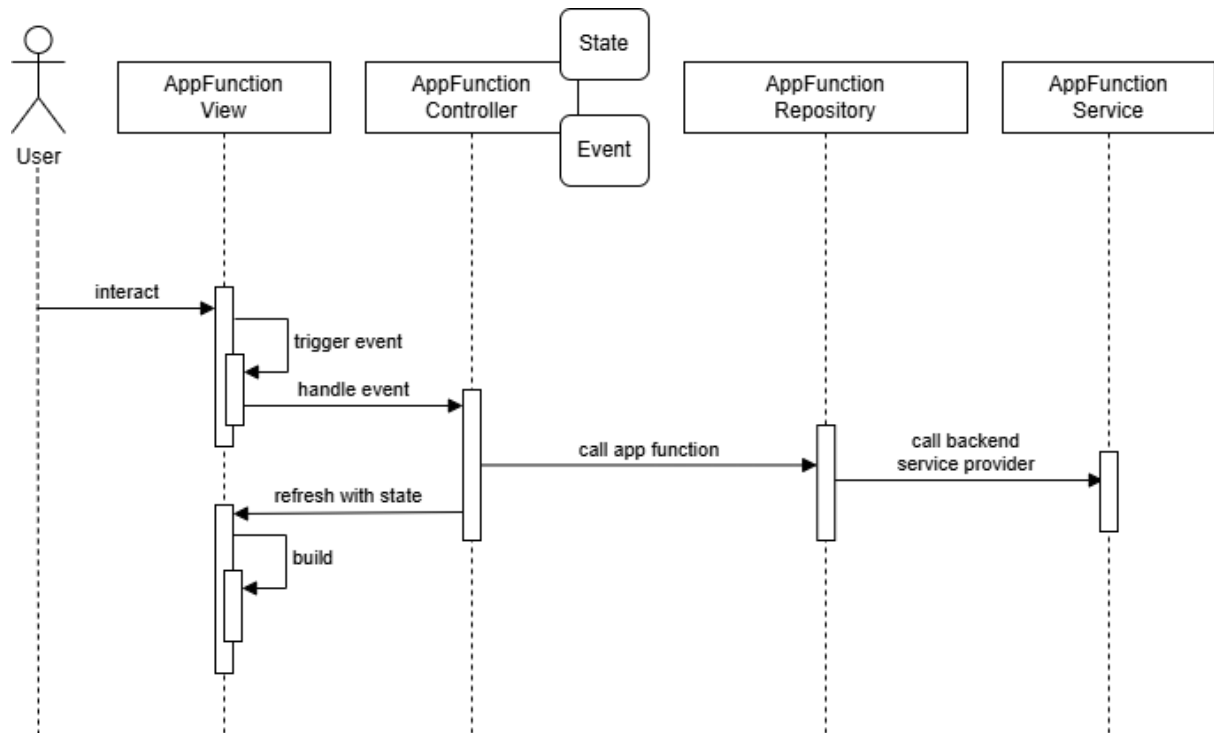
  @override
  Widget build(BuildContext context) {
    return ...; // my UI page
  }
}
```

This implementation associates a new controller with the view and triggers an initial event. A new repository is created and scoped within the application function. The `build()` method is overridden to define the user interface, while the `init()` method can also be overridden to execute logic during page initialization before rendering the page. The implementation of the controller can be seen later further below.

TRIGGERING AN EVENT

We have already seen that an initial event can be triggered when initializing the page. However, events can occur at any time and same flow is followed when handling the event:

1. The controller handles the event, provided it was registered during controller creation.
2. If the event requires executing an application function, the repository is called.
3. The application function may modify the state of the page, and upon completion, the page refreshes to reflect the updated state.



An event can be triggered inside the `build()` method of the page, as shown in the example below:

```

class AppFunctionView extends ALFWidgetView {
  AppFunctionView({super.key}) : super(
    controller: AppFunctionController(),
    event: AppFunctionEvent.initialize,
    repositories: [
      ALFRepositoryProvider<AppFunctionRepository>(create: (context) =>
AppFunctionRepository()),
    ]);

  @override
  Widget build(BuildContext context) {
    return ...
    onPressed: () => trigger(event: AppFunctionEvent.submit),
    ...; // my UI page
  }
}

```

IMPLEMENTING THE CONTROLLER

The controller defines:

- **Events** – Implemented as an enumerator.
- **States** – Implemented as an enumerator.
- **Event Handling Logic** – Managed within the controller itself.

The state object stores all fields from the page in a dictionary, where the key represents the field name and the value holds the actual data. These fields are initialized when the controller class is instantiated.

```
enum AppFunctionEvent { initialize, changeUsername, submit }
enum AppFunctionState { username, loading }

class AppFunctionController extends ALFWidgetController {
  AppFunctionController()
    : super(
      state: {
        AppFunctionState.username: "",
        AppFunctionState.loading: false,
      },
    ) {
    register(event: AppFunctionEvent.initialize, trigger: initialize);
    register(event: AppFunctionEvent.changeUsername, trigger: changeUsername);
    register(event: AppFunctionEvent.submit, trigger: submit);
  }

  void initialize({dynamic params}) {
    update(state: {AppFunctionState.loading: true});
    ... <e.g. connect to backend>
    update(state: {AppFunctionState.loading: false});
  }

  void changeUsername({dynamic params}) {
    ...
    update(state: {AppFunctionState.username: ...});
  }

  void submit({dynamic params}) {
    ...
    refreshView();
  }
}
```

From the above code snippet:

- The page includes a `username` field, which is initialized and stored in the state dictionary when creating the controller.
- A hidden field (`loading`) is also initialized. If `loading` is `true`, the page displays a loading indicator, executes backend logic, and then sets `loading` to `false` to display the actual page.

Events are registered with corresponding methods to handle them. Since events can have parameters, the method signature allows an optional parameter. As a naming convention, the event name matches the method name that handles it.

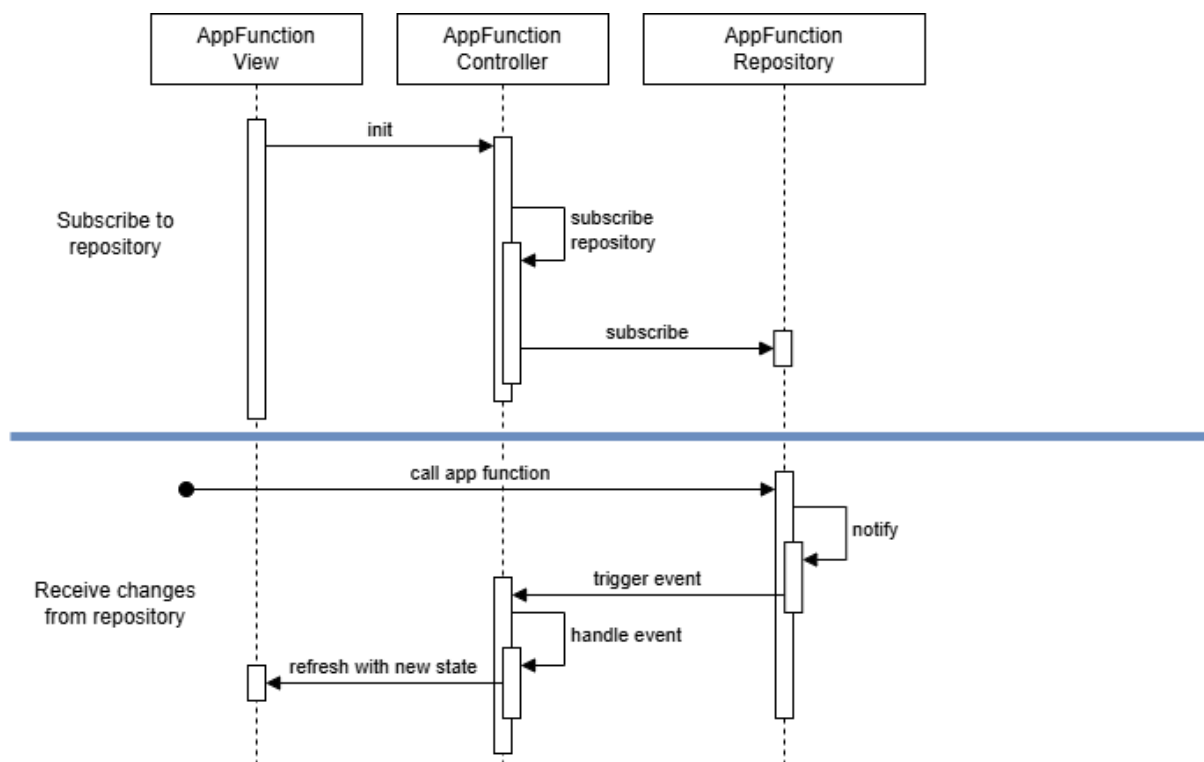
The controller provides two methods to refresh the page with updated data:

- `update()` – Updates the state before refreshing the page.
- `refreshView()` – Refreshes the page directly, useful when the page state or repository state has changed through other means.

SUBSCRIBING TO CHANGES FROM THE REPOSITORY

In the Model-View-Controller (MVC) pattern, changes to the model notify the controller, which then performs the necessary actions, such as updating the view, to reflect these changes.

To facilitate this, the controller provides a method called `subscribeRepository()`, allowing it to subscribe to changes from the repository. A common use case for this functionality is language switching, where changing the language requires updating labels across all pages dynamically.



Subscription to repository changes is typically done during the initialization of the controller in the `init()` method. During this process:

1. The controller specifies which types of changes it is interested in.
2. It defines which event should be triggered when such changes occur.
3. The controller handles the event in the same way as other events, as discussed earlier.

Below is an example demonstrating how the controller subscribes to repository changes:

```

enum AppFunctionEvent { ..., notify }
...
class AppFunctionController extends ALFWidgetController {
  AppFunctionController()
    : super(
      state: {
        ...
      }
    ) {
    ...
  }
}
  
```

```

        },
    ) {
        ...
        register(event: AppFunctionEvent.notify, trigger: notify);
    }

    void notify({dynamic params}) {
        ...
    }
    ...
    @override
    void init() {
        subscribeRepository<AppFunctionRepository>(
            listenChange: AppFunctionChangeType.appChange,
            event: AppFunctionEvent.notify,
        );
    }
}

```

The repository can notify the controller when a relevant change occurs, as shown in the example below:

```

class AppFunctionRepository extends ALFRepository {
    ...
    void appFunction() {
        ...
        notify(change: AppFunctionChangeType.appChange);
    }
}

```

At this point, we have covered the requirements, followed by the framework design, along with code snippets demonstrating how to define different application layers using the Application Layers Framework.

Now, let us put everything together and implement a sample application using ALF.

2.4 SHOWCASE

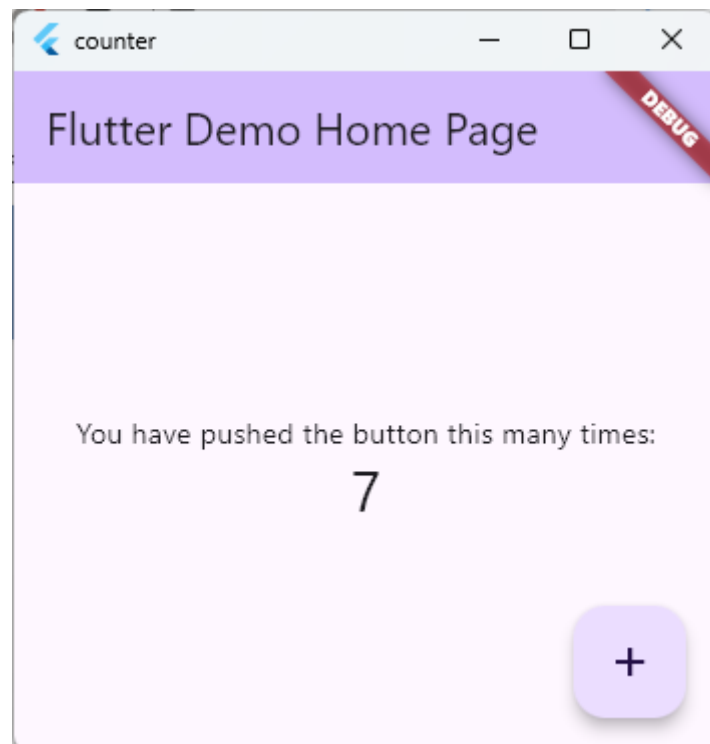
We will showcase the features of the Application Layers Framework by developing a sample application. This sample application will be used throughout this and subsequent chapters to illustrate best practices and technical requirements. Our focus is on demonstrating the framework's technical aspects, with a minimal UI layout to keep things simple and clear.

PREPARATION AND DESIGN

To begin developing a Flutter application from scratch, run the following command:

```
$ flutter create sample_app
```

This command initializes a basic Flutter project that displays a page with a centered counter and a button positioned at the bottom right. Each time the button is pressed, the counter increments by one.



Below is the source code generated by the command above, with all comments removed to improve readability.

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
```

```

Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple)),
    home: const MyHomePage(title: 'Flutter Demo Home Page'),
  );
}

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

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

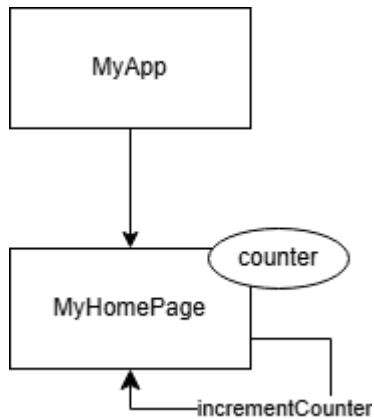
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

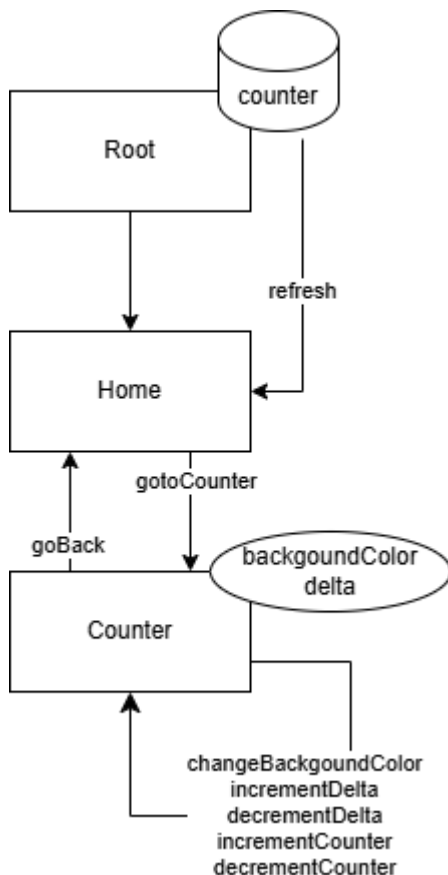
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(backgroundColor: Theme.of(context).colorScheme.inversePrimary, title:
Text(widget.title)),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text('You have pushed the button this many times:'),
            Text('$_counter', style: Theme.of(context).textTheme.headlineMedium),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(onPressed: _incrementCounter, tooltip:
'Increment', child: const Icon(Icons.add)),
    );
  }
}

```

By visualizing the navigation flow along with the relevant events and states, we gain a deeper understanding of the application's overall structure. In this simple Flutter example, the diagram highlights the `MyApp` and `MyHomePage` widgets, along with the `incrementCounter` event, which can be triggered to update the counter state.



Next, let us transform above implementation by integrating the Application Layers Framework, along with enhancements that showcase its key features. In the updated navigation structure, additional events have been introduced, and the counter is now stored in a repository to persist throughout the entire application. Page-specific fields, such as `backgroundColor` and `delta` from the Counter page in our sample app, are defined within the page state. These elements are illustrated in the diagram below.



In the original implementation, `MyApp` is a stateless widget that serves as a non-visible entry point. We will rename it to `Root`, as it will become the landing page of our sample application. Similarly, `MyHomePage` will be renamed to `Home`. While the `Home` page still displays the counter, incrementing its value now requires navigating to a separate page called `Counter`, where we demonstrate page-level state management.

The `Counter` page introduces triggerable events that modify both its local state and the global `counter` attribute, which is stored in a repository to maintain persistence across the application. This design allows the counter value to remain accessible, even when navigating back to the `Home` page.

To ensure the `Home` page reflects updates to the counter whenever it is incremented or decremented in the `Counter` page, its controller subscribes to repository changes. The `backgroundColor` field, declared within the page state, is used for demonstration purposes to show how altering a page-specific field can impact UI presentation. Another local field, `delta`, determines the increment or decrement step for the counter.

Importantly, the `Counter` page resets to its initial state each time it is launched via navigation from the `Home` page.

Based on the application requirements outlined above, the following classes will be implemented using the Application Layers Framework.

No	Layer/Role	Class/Enum	Parent Class	Attributes	Methods
1	Counter Entity	CounterEntity	ALFEntity	counter	-
2	Counter Repository	CounterRepository	ALFRepository	counterEntity	incrementCounter() decrementCounter()
3	Root Page	RootPage	ALFWidgetView	-	build()
4	Root Controller	RootController	ALFWidgetController	-	-
5	Home Page	HomePage	ALFWidgetView	-	build()
6	Home Controller	HomeController	ALFWidgetController	-	gotoCounter()
7	Home Event	HomeEvent	(enum)	gotoCounter	
8	Counter Page	CounterPage	ALFWidgetView	-	build()
9	Counter Controller	CounterController	ALFWidgetController	-	changeBackgroundColor() incrementDelta() decrementDelta() incrementCounter() decrementCounter()
10	Counter Event	CounterEvent	(enum)	changeBackgroundColor incrementDelta decrementDelta incrementCounter decrementCounter	-
11	Counter State	CounterState	(enum)	backgroundColor delta	-

The implementation is fairly straightforward. It involves creating a counter repository class along with its associated entity, and defining the relevant pages with their respective controllers, events, and states based on the navigation structure presented in the diagram above.

As a starting point, make sure the Application Layers Framework package is included in your `pubspec.yaml` file, since it forms the foundation of the sample application's architecture.

```
...
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.8
```

```

    application_layers_framework: any
    ...
dev_dependencies:
    ...

```

After saving the file, run the following command to fetch and integrate the framework into your Flutter project:

```
$ flutter pub get
```

Now we can proceed with implementing the classes and enums listed in the table above. To maintain focus on the core application logic, all import statements have been omitted from the source code below.

CODING THE MODEL AND LANDING PAGE

First, we need to consider the business or domain objects relevant to the application. Although our sample app is quite simple and only includes a counter, it is still important to follow a proper design approach by incorporating an entity layer. This means we will encapsulate the counter in a dedicated `CounterEntity` class. The implementation is shown below.

```

class CounterEntity extends ALFEntity {
    int counter = 0;
}

```

This entity will be utilized by the `CounterRepository` as implemented below. The repository offers functionality to increment and decrement the counter value and notifies any subscribed controllers of changes. Since the counter is displayed on the `Home` page, its controller will subscribe to the repository during the page's initialization to reflect updates in real time.

```

class CounterRepository extends ALFRepository {
    CounterEntity counterEntity = CounterEntity();

    void increment(int delta) {
        counterEntity.counter = counterEntity.counter + delta;
        notify();
    }

    void decrement(int delta) {
        counterEntity.counter = counterEntity.counter - delta;
        notify();
    }
}

```

A Flutter application typically begins with a landing page. Although this page will not be visually rendered, it still adheres to the principles of the Application Layers Framework, namely, it is linked to a controller that can handle events and manage state to determine navigation and behavior.

In our sample application, the landing page is named `Root`. While it does not currently manage state or handle events, we still implement a controller for it. This ensures consistency across all pages and prepares the app for future enhancements, should new requirements arise. For example, some mobile applications display introductory screens on first launch, and subsequently show the home page on future visits.

```
class RootController extends ALFWidgetController {}
```

The implementation of the page is shown below.

```
class RootPage extends ALFWidgetView {
  RootPage({super.key})
    : super(
        controller: RootController(),
        repositories: [ALFRepositoryProvider<CounterRepository>(create: (context) =>
CounterRepository())],
      );

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
useMaterial3: true),
      home: HomePage(title: "Flutter Demo Home Page"),
    );
  }
}
```

There is minimal logic at this stage, primarily the initialization of `CounterRepository` to make it accessible across the entire application, and rendering the home page via `HomePage`. You can disregard the syntax error for now, as `HomePage` will be implemented in the section below.

CODING THE HOME PAGE

Next, we move on to the `HomePage` page, which is responsible for displaying the initial counter value. Since this page relies on its associated controller, we will first examine the controller implementation.

```
enum HomeEvent { refresh, gotoCounter }

class HomeController extends ALFWidgetController {
  HomeController() {
    register(event: HomeEvent.refresh, trigger: refreshView);
    register(event: HomeEvent.gotoCounter, trigger: gotoCounter);
  }

  void gotoCounter({dynamic params}) {
    Container page = Container();
    Navigator.push(context, MaterialPageRoute(builder: (context) => page));
  }

  @override
  void init() {
    subscribeRepository<CounterRepository>(event: HomeEvent.refresh);
  }
}
```

There are two events to handle, as illustrated earlier in the navigation diagram. The first involves a button on the page that navigates to the `Counter` page, where users can increment or decrement the counter. This is

triggered by `HomeEvent.gotoCounter`, which uses Flutter's `Navigator` to move to the next page. Since the `Counter` page has not been implemented yet, we will use an empty container as a placeholder.

The second event handles refreshing the `Home` page whenever the counter is updated in the repository. This is possible because the `init()` method subscribes to repository changes, triggering `HomeEvent.refresh`. For this event, we will use ALF's built-in `refreshView()` method, which streamlines UI updates.

Compared to the original implementation, `HomePage` now extends `ALFWidgetView`, with its UI logic encapsulated in the `build()` method. The counter is retrieved from the repository, and pressing the navigation button triggers the event to proceed to the next page.

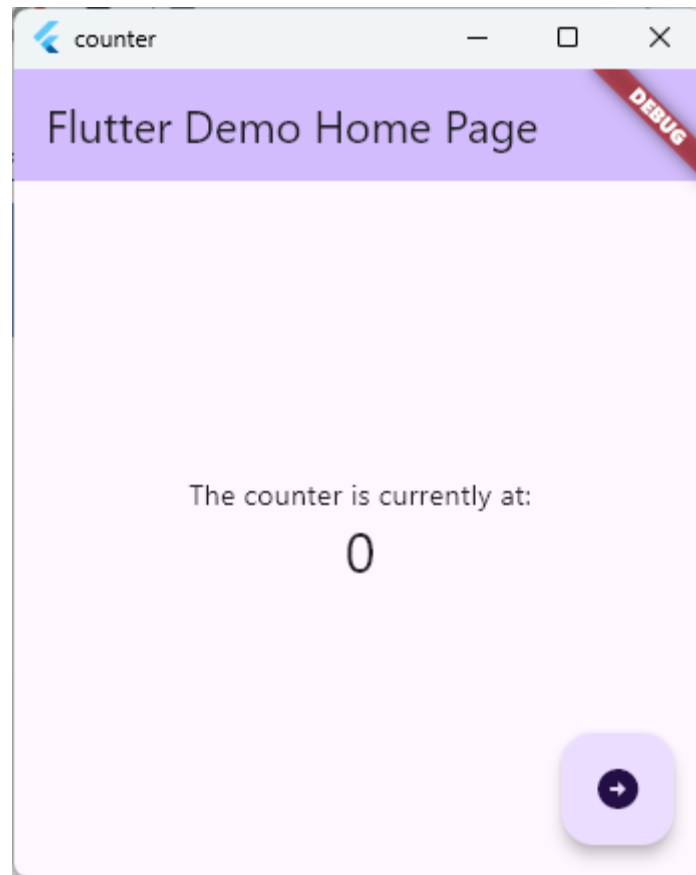
```
class HomePage extends ALFWidgetView {
  final String title;

  HomePage({required this.title, super.key}) : super(controller: HomeController());

  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      appBar: AppBar(backgroundColor: Theme.of(context).colorScheme.inversePrimary, title:
Text(title)),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text("The counter is currently at:"),
            Text('${counterRepository.counterEntity.counter}', style:
Theme.of(context).textTheme.headlineMedium),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => trigger(event: HomeEvent.gotoCounter),
        tooltip: "Go to counter",
        child: const Icon(Icons.arrow_circle_right),
      ),
    );
  }
}
```

At this stage, you can already run the application to display the initial counter value, along with a button to navigate to the next page, which will be the final and more complex part of the implementation. To enable this, modify the `main()` function to use `RootPage` as the application's entry point.

```
void main() {
  runApp(RootPage());
}
```



CODING THE COUNTER PAGE

As usual, we begin by implementing the controller along with its associated states and events. This page displays three fields: `counter`, `backgroundColor`, and `delta`. The `counter` is already defined in the repository, while the remaining two fields are initialized within the controller's constructor using a dictionary. Each field is keyed by an entry from the state enumerator, allowing compile-time validation and reducing the risk of typos during development.

According to the navigation diagram, five distinct events need to be handled, each represented by its own event method in the implementation below. The event responsible for changing the background color accepts an additional parameter that specifies the target color. Handling this event involves updating the corresponding value in the page state and refreshing the page to reflect the change.

The events for incrementing and decrementing the counter invoke the relevant application functions from the repository, followed by refreshing the page. Adjusting the `delta` value works similarly to the background color change: the field is incremented or decremented by one, then the view is refreshed to display the updated state.

```
enum CounterEvent { changeBackgroundColor, incrementDelta, decrementDelta, incrementCounter,
decrementCounter }

enum CounterState { backgroundColor, delta }

class CounterController extends ALFWidgetController {
  CounterController() : super(state: {CounterState.backgroundColor: Colors.white,
CounterState.delta: 1}) {
    register(event: CounterEvent.changeBackgroundColor, trigger: changeBackgroundColor);
```

```

    register(event: CounterEvent.incrementDelta, trigger: incrementDelta);
    register(event: CounterEvent.decrementDelta, trigger: decrementDelta);
    register(event: CounterEvent.incrementCounter, trigger: incrementCounter);
    register(event: CounterEvent.decrementCounter, trigger: decrementCounter);
  }

  void changeBackgroundColor({dynamic params}) {
    update(state: {CounterState.backgroundColor: params});
  }

  void incrementDelta({dynamic params}) {
    update(state: {CounterState.delta: state[CounterState.delta] + 1});
  }

  void decrementDelta({dynamic params}) {
    update(state: {CounterState.delta: state[CounterState.delta] - 1});
  }

  void incrementCounter({dynamic params}) {
    CounterRepository counterRepository = repo<CounterRepository>();
    counterRepository.increment(state[CounterState.delta]);
    refreshView();
  }

  void decrementCounter({dynamic params}) {
    CounterRepository counterRepository = repo<CounterRepository>();
    counterRepository.decrement(state[CounterState.delta]);
    refreshView();
  }
}

```

As previously mentioned, the controller's `update()` method modifies the page state and automatically refreshes the view unless the `refresh` parameter is explicitly set to `false`. Since the counter's increment and decrement actions do not utilize the `update()` method, they must explicitly invoke `refreshView()` to reflect changes in the UI.

With event handling now in place, the page implementation can render its fields by accessing the state managed within the controller and retrieving the counter value from `CounterRepository` through the widget tree context. User interactions will trigger events that are dispatched to and processed by the controller.

```

class CounterPage extends ALFWidgetView {
  CounterPage({super.key}) : super(controller: CounterController());

  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      backgroundColor: state[CounterState.backgroundColor],
      appBar: AppBar(backgroundColor: Theme.of(context).colorScheme.inversePrimary, title:
Text("Flutter Demo Counter Page")),
      body: Center(
        child: SizedBox(
          width: 400,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,

```

```

        children: <Widget>[
          Column(
            mainAxisAlignment: MainAxisAlignment.min,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              ListTile(
                title: Text("White color"),
                leading: Radio<Color>(
                  value: Colors.white,
                  groupValue: state[CounterState.backgroundColor],
                  onChanged: (Color? value) => trigger(event:
CounterEvent.changeBackgroundColor, params: value!),
                ),
              ),
              ListTile(
                title: Text("Blue color"),
                leading: Radio<Color>(
                  value: Colors.blue,
                  groupValue: state[CounterState.backgroundColor],
                  onChanged: (Color? value) => trigger(event:
CounterEvent.changeBackgroundColor, params: value!),
                ),
              ),
              ListTile(
                title: Text("Green color"),
                leading: Radio<Color>(
                  value: Colors.green,
                  groupValue: state[CounterState.backgroundColor],
                  onChanged: (Color? value) => trigger(event:
CounterEvent.changeBackgroundColor, params: value!),
                ),
              ),
            ],
          ),
          Row(
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
              ElevatedButton(
                onPressed: () => trigger(event: CounterEvent.decrementCounter),
                child: const Icon(Icons.remove),
              ),
              const SizedBox(width: 10),
              Text("Counter value: ${counterRepository.counterEntity.counter}"),
              const SizedBox(width: 10),
              ElevatedButton(
                onPressed: () => trigger(event: CounterEvent.incrementCounter),
                child: const Icon(Icons.add),
              ),
            ],
          ),
          const SizedBox(height: 10),
          Row(
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
              ElevatedButton(
                onPressed: state[CounterState.delta] <= 1 ? null : () => trigger(event:
CounterEvent.decrementDelta),
                child: const Icon(Icons.remove),
              ),

```

```

    ),
    const SizedBox(width: 10),
    Text("Delta value: ${state[CounterState.delta]}"),
    const SizedBox(width: 10),
    ElevatedButton(
      onPressed: () => trigger(event: CounterEvent.incrementDelta),
      child: const Icon(Icons.add),
    ),
  ],
),
],
),
),
),
),
);
}
}

```

After implementing `CounterPage`, be sure to update `HomeController` to use this new page implementation.

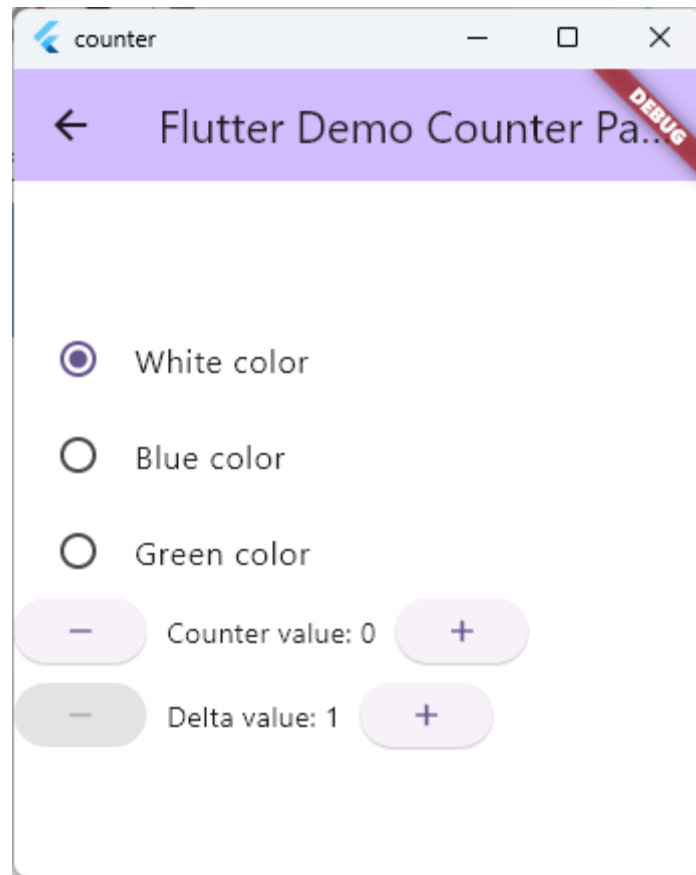
```

class HomeController extends ALFWidgetController {
  ...
  void gotoCounter({dynamic params}) {
    CounterPage page = CounterPage();
    Navigator.push(context, MaterialPageRoute(builder: (context) => page));
  }
  ...
}

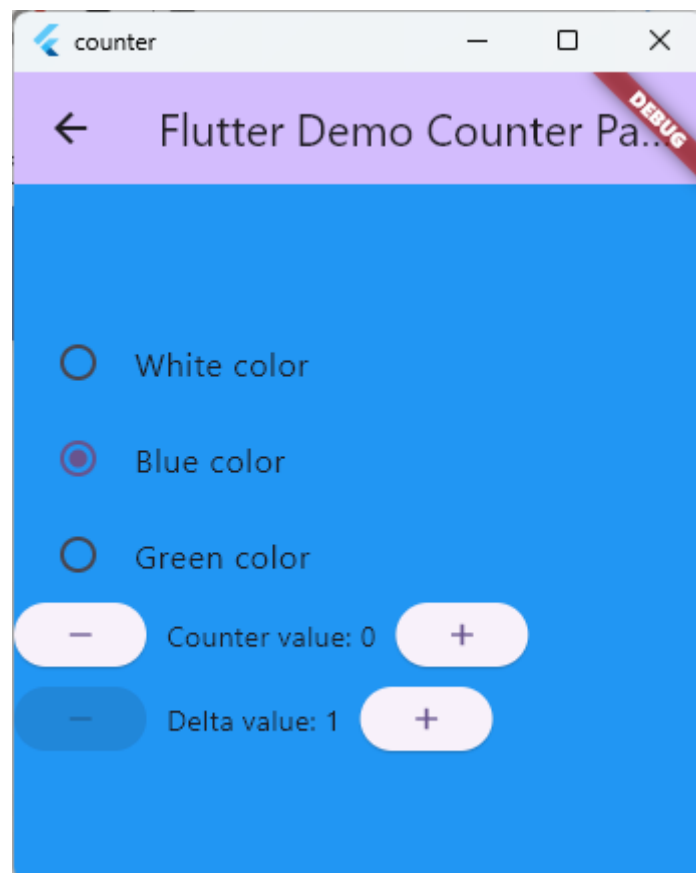
```

RUNNING THE APPLICATION

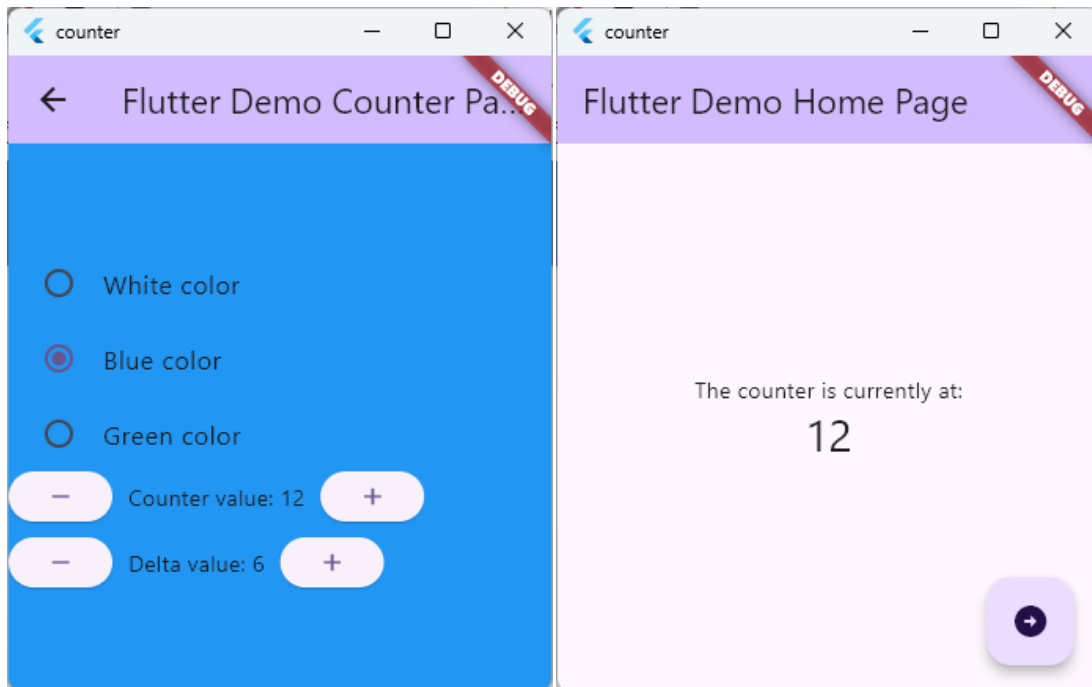
Once the application is launched and navigated to the `Counter` page, the following screen will be displayed. While it may not be visually refined, it effectively fulfills its intended purpose.



For example, selecting a color option will instantly update the page's background color.



When the counter value is increased and the user returns to the `Home` page, the updated value is immediately reflected thanks to the page's subscription to changes from `CounterRepository`.



When navigating back to the `Counter` page, you will notice that the `backgroundColor` and `delta` attributes have been reset to their initial values. To preserve these values across page transitions, they would need to be stored in the repository rather than within the page state. This behavior helps illustrate the distinction between repository-managed data, which persists throughout the application or a module, and page state, which is scoped to the lifecycle of a single view.

3 INTERNATIONALIZATION

REQUIREMENTS AND SETUP

When designing an application, it is common to display text and labels in English or another familiar language such as your native tongue. However, supporting multiple languages can significantly improve the accessibility and reach of your application across diverse user bases.

Flutter offers built-in support for internationalization (i18n), providing a standardized approach to localizing your app. For comprehensive guidance, refer to the official Flutter documentation: **Internationalizing Flutter apps** (<https://docs.flutter.dev/ui/accessibility-and-internationalization/internationalization>).

So, where should we begin integrating this into our own application?

While Flutter enables localization functionality, how it is applied depends on your app's structure and requirements. Key considerations include:

- Defining which parts of the app should be localized
- Translating content into target languages
- Automatically adapting to the device's locale
- Allowing users to change the language within the app

These insights provide a foundation for internationalization, though the implementation will vary depending on the specific needs and flow of your application.

Let us walk through the integration process using the sample app from the previous chapter.

First, start by adding the necessary dependency packages to your `pubspec.yaml` file:

```
...
dependencies:
...
  application_layers_framework: any
  flutter_localizations:
    sdk: flutter
  intl: any
...
flutter:
...
  generate: true
...
```

The internationalization package reads its configuration from the `l10n.yaml` file, which is located in the root directory of the Flutter project.

```
synthetic-package: false
arb-dir: assets/l10n
template-arb-file: l10n_en.arb
output-dir: lib/common/l10n
```

```
output-localization-file: l10n.dart
```

In our sample application, all language files will be stored under the `assets/l10n` directory. The base language file will be `l10n_en.arb`, while additional files for other languages will follow the naming convention: prefixed with `l10n` and suffixed with the corresponding locale (e.g. `l10n_en_uk.arb`).

During the build process, the localization package generates helper classes for use during development. These are stored in the `l10n.dart` file under the `lib/common/l10n` directory as defined in the `l10n.yaml` file, and provide access to localized strings throughout the app.

These configurations form the foundation for localizing the application and enable seamless integration of multiple language options.

EXTERNALIZING TEXTS AND LABELS

The next step is to identify all hardcoded strings, such as UI text and labels, within the source code and move them to the base language file to support localization. We will demonstrate this process using the `RootPage` class as an example.

```
class RootPage extends ALFWidgetView {
  RootPage({super.key})
    : super(
        controller: RootController(),
        repositories: [ALFRepositoryProvider<CounterRepository>(create: (context) =>
CounterRepository())],
      );

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
useMaterial3: true),
      home: HomePage(title: "Flutter Demo Home Page"),
    );
  }
}
```

We have identified two hardcoded strings that can be externalized for localization, as outlined below.

```
{
  "page_root_app_title": "Flutter Demo",
  "page_root_home_title": "Flutter Demo Home Page"
}
```

The keys defined in the language file serve as variable names that will later replace hardcoded strings in the source code. It is essential to follow a consistent naming convention to ensure each key is easily recognizable and clearly indicates its intended purpose especially in larger development teams, where naming conflicts can become problematic.

For our implementation, we have adopted the following convention:

```
<component>_<componentName>_<fieldName>_<purpose>
```

This format promotes readability and maintainability across the project. Using the earlier example, the derived key name follows this structure and clearly communicates its context and usage.

No	Key	Example	Remarks
1	<component>	page	In this example, the texts are found inside a page.
2	<component name>	Root	The page name.
3	<field name>	app home	If possible, use the field name defined by the state. Otherwise use a pseudo field name that describes its purpose.
4	<purpose>	Title	Describes what this label or text is used for.

Externalizing the remaining hardcoded texts is straightforward. Simply go through each page implementation, identify any static text, and move those strings into the base language file. After completing this exercise, the full base language file named `l10n_en.arb` located in `assets/l10n/` will contain all localized entries using the naming convention we established earlier.

```
{
  "page_Root_app_title": "Flutter Demo",
  "page_Root_home_title": "Flutter Demo Home Page",

  "page_Home_counter_label": "The counter is currently at:",
  "page_Home_gotoCounter_hint": "Go to counter",

  "page_Counter_appBar_title": "Flutter Demo Counter Page",
  "page_Counter_backgroundColor_white": "White color",
  "page_Counter_backgroundColor_blue": "Blue color",
  "page_Counter_backgroundColor_green": "Green color",
  "page_Counter_counter_label": "Counter value:",
  "page_Counter_delta_label": "Delta value:"
}
```

TRANSLATING TO SUPPORTED LANGUAGES

Suppose we want to extend our application to support Spanish and German alongside English, but we are not fluent in those languages. In such cases, we can leverage Google Translate to generate initial translations, which can later be fine-tuned as needed.

Manually translating each string using the Google Translate website can be time-consuming and prone to error. To streamline this process, we can use the Google Translate API to automate the generation of base translation files for the target languages.

The Python script below demonstrates how to perform this automated translation, producing `.arb` files that can be directly integrated into your Flutter project's localization workflow.

```
import asyncio
import json
import shutil
from googletrans import Translator
from os.path import exists
import shutil

def fullPath(folder, file):
    return "{folder}/{file}".format(folder = folder, file = file)
```

```

async def translateText(value, src, dest):
    async with Translator() as translator:
        result = await translator.translate(value, src=src, dest=dest)
        return result

translator = Translator()

folder = 'assets/l10n'
source = 'l10n_en.arb'
sourceOriginal = 'l10n_en.arb.orig'
targets = {
    "de_de": "l10n_de_de.arb",
    "es_es": "l10n_es_es.arb",
}
copies = [
    (source, sourceOriginal),
    (source, "l10n_en_uk.arb"),
    ("l10n_de_de.arb", "l10n_de.arb"),
    ("l10n_es_es.arb", "l10n_es.arb"),
]

with open(fullPath(folder, source)) as file:
    data = json.load(file)

if not exists(fullPath(folder, sourceOriginal)):
    shutil.copyfile(fullPath(folder, source), fullPath(folder, sourceOriginal))

with open(fullPath(folder, sourceOriginal)) as file:
    origData = json.load(file)

for key in data:
    if key not in origData.keys():
        origData[key] = ""

for language in targets:
    targetData = {}
    if exists(fullPath(folder, targets[language])):
        with open(fullPath(folder, targets[language]), encoding="utf-8") as file:
            targetData = json.load(file)

    targetDataCopy = targetData.copy()
    for key in targetData:
        if key not in data.keys():
            del targetDataCopy[key]
    targetData = targetDataCopy

    for key in data:
        if not key.startswith("@"):
            if key not in targetData.keys() or data[key] != origData[key]:
                value = data[key]
                print("Source {key} = {value}".format(key = key, value = value))
                loop = asyncio.new_event_loop()
                task = loop.create_task(translateText(value, src = "en", dest = language, ))
                loop.run_until_complete(task)
                translation = task.result()
                targetData[key] = translation.text
                print("Dest {key} = {value}".format(key = key, value = targetData[key]))

```

```

with open(fullPath(folder, targets[language]), "w", encoding="utf-8") as file:
    file.write("{}")
    first = True
    for key in targetData:
        if not first:
            file.write(',')
        else:
            first = False
        file.write('\n\n    "{key}_orig": "{value}",'.format(key = key, value =
data[key]))
        file.write('\n    "{key}":    "{value}"'.format(key = key, value =
targetData[key]))
        file.write("\n}")

for src, dest in copies:
    print("Copy {0} to {1}".format(fullPath(folder, src), fullPath(folder, dest)))
    shutil.copyfile(fullPath(folder, src), fullPath(folder, dest))

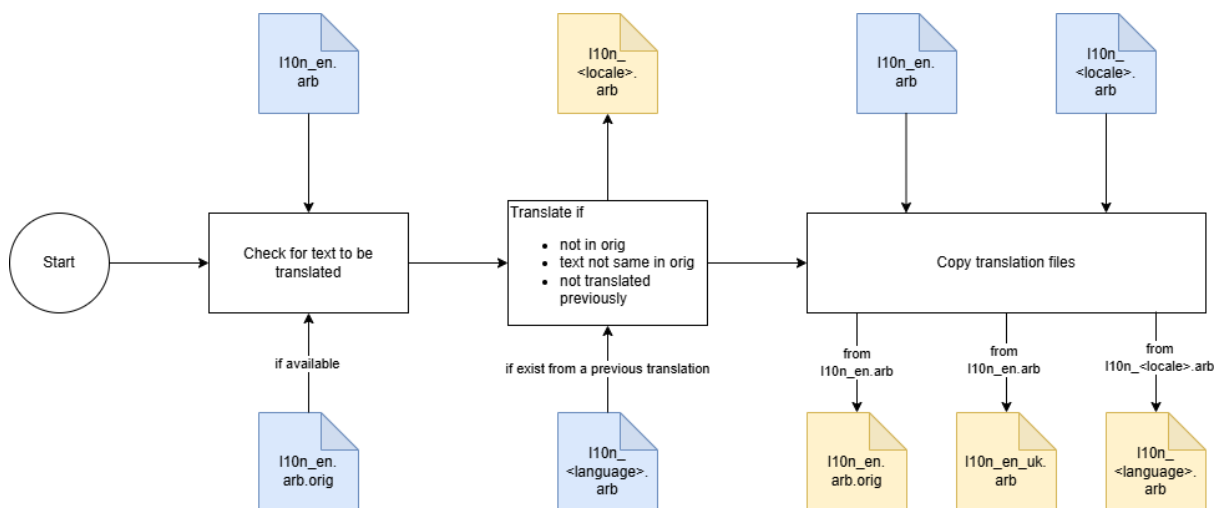
```

Python will not be used at runtime; instead, it will serve as a development utility for building automation tools. As a convention, these scripts should be organized under the `assets/tools` directory within the Flutter project.

Since these tools may rely on third-party packages such as `googletrans`, it is best practice to set up a dedicated Python virtual environment. This ensures that external dependencies will not interfere with the system-wide Python installation.

Running the script will execute the following tasks:

```
$ python assets/tools/translate.py
```



In our example, English serves as the base language. The target languages, German and Spanish, can be customized within the Python script to suit your localization needs.

```

...
targets = {
    "de_de": "l10n_de_de.arb",
    "es_es": "l10n_es_es.arb",
}
copies = [

```

```
...
  ("l10n_de_de.arb", "l10n_de.arb"),
  ("l10n_es_es.arb", "l10n_es.arb"),
]
...
```

The output of the German translation appears as shown below, with each translated string preceded by its original English base text for easier comparison and reference.

```
{
  "page_root_app_title_orig": "Flutter Demo",
  "page_root_app_title":     "Flattern -Demo",

  "page_root_home_title_orig": "Flutter Demo Home Page",
  "page_root_home_title":     "Flutter -Demo -Startseite",

  "page_home_counter_label_orig": "The counter is currently at:",
  "page_home_counter_label":     "Der Schalter ist derzeit um:",

  "page_home_gotoCounter_hint_orig": "Go to counter",
  "page_home_gotoCounter_hint":     "Gehen Sie zur Konter",

  "page_counter_appBar_title_orig": "Flutter Demo Counter Page",
  "page_counter_appBar_title":     "Flutter -Demo -Zählerseite",

  "page_counter_backgroundColor_white_orig": "White color",
  "page_counter_backgroundColor_white":     "Weiße Farbe",

  "page_counter_backgroundColor_blue_orig": "Blue color",
  "page_counter_backgroundColor_blue":     "Blaue Farbe",

  "page_counter_backgroundColor_green_orig": "Green color",
  "page_counter_backgroundColor_green":     "Grüne Farbe",

  "page_counter_counter_label_orig": "Counter value:",
  "page_counter_counter_label":     "Gegenwert:",

  "page_counter_delta_label_orig": "Delta value:",
  "page_counter_delta_label":     "Deltawert:"
}
```

If you are familiar with German, you may notice that some of the translated strings are not entirely accurate or contextually appropriate. As mentioned earlier, translations can be fine-tuned manually, and importantly, any adjustments you make will not be overwritten when you rerun the script provided the original base text remains unchanged.

To support this workflow, we maintain a copy of the original English source file, `l10n_en.arb`, as `l10n_en.arb.orig`. This allows the script to compare the current base file against the original, ensuring that only new or modified entries trigger an update.

Below is the corrected version of `l10n_de_de.arb` with manually refined translations.

```
{
```

```

"page_Root_app_title_orig": "Flutter Demo",
"page_Root_app_title":      "Flutter-Demo",

"page_Root_home_title_orig": "Flutter Demo Home Page",
"page_Root_home_title":      "Flutter-Demo-Startseite",

"page_Home_counter_label_orig": "The counter is currently at:",
"page_Home_counter_label":      "Der Zähler ist derzeit:",

"page_Home_gotoCounter_hint_orig": "Go to counter",
"page_Home_gotoCounter_hint":      "Gehe zum Zähler",

"page_Counter_appBar_title_orig": "Flutter Demo Counter Page",
"page_Counter_appBar_title":      "Flutter-Demo-Zählerseite",

"page_Counter_backgroundColor_white_orig": "White color",
"page_Counter_backgroundColor_white":      "Weiße Farbe",

"page_Counter_backgroundColor_blue_orig": "Blue color",
"page_Counter_backgroundColor_blue":      "Blaue Farbe",

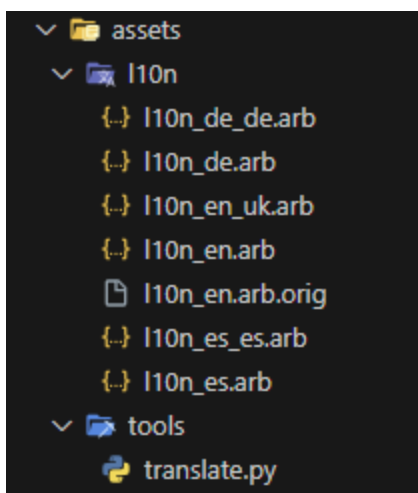
"page_Counter_backgroundColor_green_orig": "Green color",
"page_Counter_backgroundColor_green":      "Grüne Farbe",

"page_Counter_counter_label_orig": "Counter value:",
"page_Counter_counter_label":      "Zählerwert:",

"page_Counter_delta_label_orig": "Delta value:",
"page_Counter_delta_label":      "Deltawert:"
}

```

Since I am not proficient in Spanish, we will keep the existing Spanish translation unchanged. At this stage, your project should now contain the following language files:

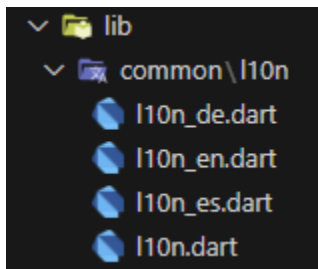


BUILDING AND USING HELPER AND UTILITY CLASSES

Now that the language files are in place, we can generate the necessary helper classes by running the following command within the Flutter project directory:

```
$ flutter gen-l10n
```

After executing the command above, the generated helper classes will appear as shown in the screenshot below.



It is important to note that whenever .arb language files are modified, the helper classes must be regenerated since the localized texts are compiled into these classes during build time.

The next step is to integrate the generated helper classes into our sample application. To maintain a clean and consistent localization interface, we encapsulate the functionality within a singleton utility class. This abstraction simplifies access to localized resources throughout the app and reduces dependency on implementation details.

At this stage, the initial locale is hardcoded to German during the singleton's initialization. Later we will enhance this logic to support dynamic locale switching allowing the user to select or change the language at runtime.

```
class L10nUtil extends ALFUtil {
  static final L10nUtil _instance = L10nUtil._init();

  factory L10nUtil() {
    return _instance;
  }

  late AppLocalizations _appLocalizations;

  L10nUtil._init() {
    locale = Locale("de", "DE");
  }

  set localeByCode(String localeCode) {
    locale = codeToLocale(localeCode);
  }

  set locale(Locale locale) {
    _appLocalizations = lookupAppLocalizations(locale);
  }

  Locale get locale {
    return codeToLocale(_appLocalizations.localeName);
  }

  String localeToCode(Locale locale) {
    return "${locale.languageCode}_${locale.countryCode}";
  }

  Locale codeToLocale(String code) {
    List<String> splitCode = code.split("_");
    return Locale(splitCode[0], splitCode[1]);
  }

  AppLocalizations translate() {
    return _appLocalizations;
  }
}
```

Utility classes are typically stateless and accessible throughout the entire application. Implementing them as singletons enhances efficiency by ensuring a single shared instance is reused wherever needed.

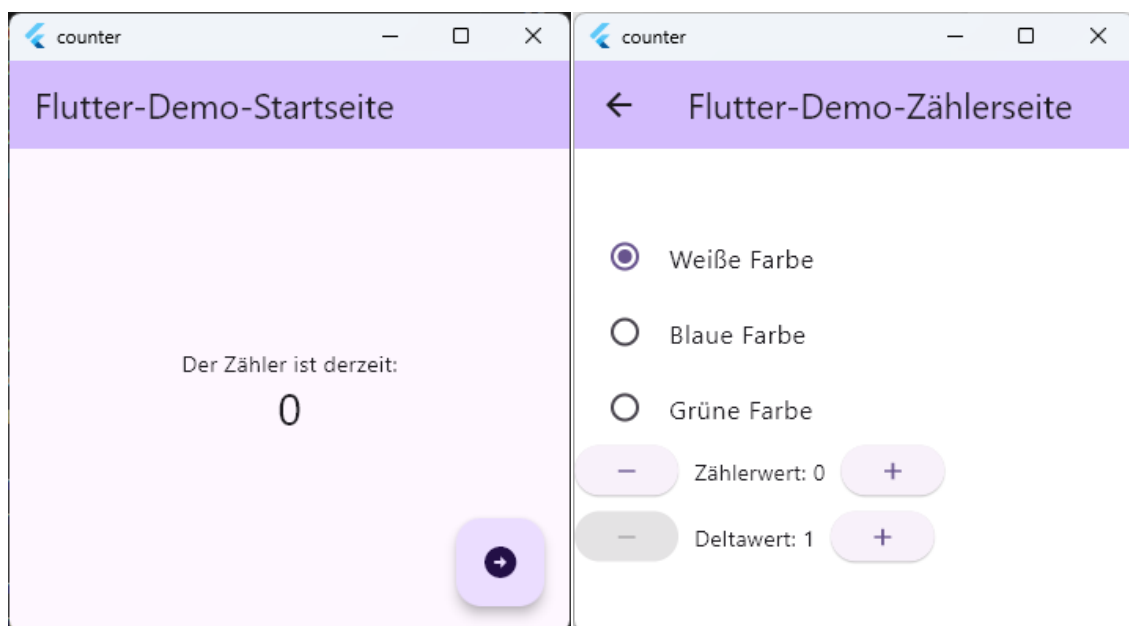
The utility class includes a `translate()` method, which will be used to replace hardcoded text throughout the application's page implementations. For instance, when revisiting the `RootPage` class, all static strings should now be replaced using this method to ensure localization support.

Additionally, we configure localization properly by passing the recommended parameters to the `MaterialApp` constructor. While making these updates, we also take the opportunity to hide the debug banner, an optional visual element useful during development, but purely cosmetic and not needed in production.

```
class RootPage extends ALFWidgetView {
  RootPage({super.key})
    : super(
        controller: RootController(),
        repositories: [ALFRepositoryProvider<CounterRepository>(create: (context) => CounterRepository())],
      );

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: L10nUtil().translate().page_Root_app_title,
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
      locale: L10nUtil().locale,
      debugShowCheckedModeBanner: false,
      theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple), useMaterial3: true),
      home: HomePage(title: L10nUtil().translate().page_Root_home_title),
    );
  }
}
```

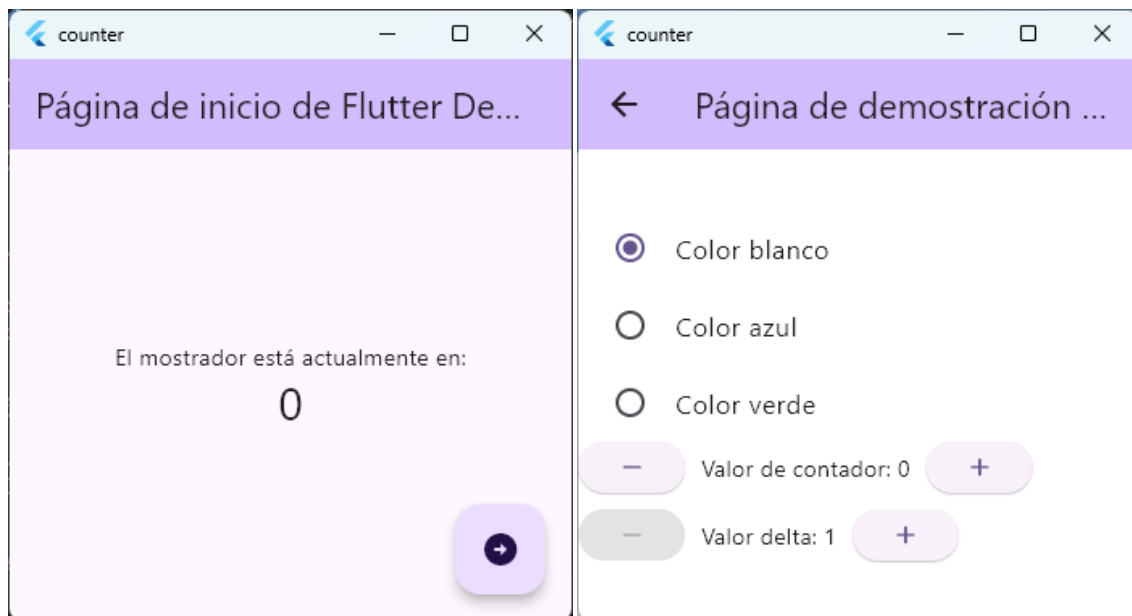
Hardcoded strings have now been replaced with references to localization keys from the language file. Repeat this process across the remaining pages of the app, then run the application. You will see that the UI texts and labels now appear in German dynamically rendered using Flutter's internationalization framework rather than fixed, hardcoded values.



To display the application in Spanish, simply update the initial locale in the `L10nUtil` class to use the Spanish locale.

```
class L10nUtil extends ALFUtil {
  ...
  L10nUtil._init() {
    locale = Locale("es", "ES");
  }
  ...
}
```

Rerun or hot reload the application to observe how the texts and labels have been seamlessly updated to Spanish leveraging the same underlying codebase and localization configuration.



CHANGING LANGUAGE FROM USER INTERFACE

While the translation mechanism functions correctly, manually hardcoding the initial locale is not a practical approach for production environments. Instead, language switching should be built into the application as a formal requirement allowing users to change the language while the app is running.

There are two commonly used UI patterns for enabling this:

- A settings page dedicated to language selection.
- An icon in the application bar for quick language switching.

In this implementation, we will adopt the second approach by adding a language toggle icon to the application bar. To support this functionality, we will introduce a dedicated repository that is globally accessible and provides a method to update the current locale.

Technically, we could have reused the existing counter repository as a quick workaround. However, to maintain clean architecture and adhere to the principle of separation of concerns, we will implement a distinct session repository. This repository represents the current user session state, which, in this context, includes the active locale.

The session repository exposes a setter method to update the locale. Crucially, it also ensures that all subscribed controllers are notified when the language changes so the UI can respond accordingly.

```
enum SessionChangeType { locale }

class SessionRepository extends ALFRepository {
  late Locale _currentLocale;

  SessionRepository() {
    currentLocale = L10nUtil().locale;
  }

  Locale get currentLocale {
    return _currentLocale;
  }

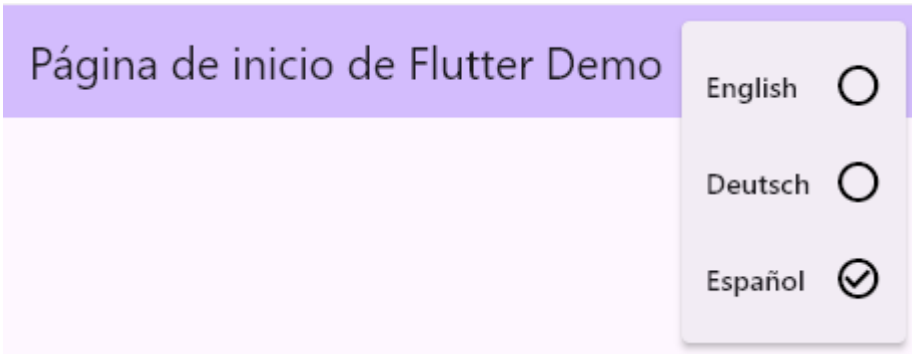
  set currentLocale(Locale locale) {
    _currentLocale = locale;
    L10nUtil().locale = locale;
    notify(change: SessionChangeType.locale);
  }
}
```

Ideally, the locale stored in the session repository should be wrapped in a dedicated class, something like a language entity that represents a language more clearly and provides useful structure. It is tempting to take shortcuts and go with quicker solutions, especially when things seem simple early on. But it is important to stick to good design principles. If you skip those now, your code can quickly become messy, hard to understand, and even harder to maintain later sometimes requiring big refactoring just to clean things up.

As mentioned earlier, we will include a language selection icon in the application bar of each page, allowing users to choose their preferred language directly from the UI.




Once the icon is clicked, a list of available languages will be displayed, allowing the user to select their preferred option.



English ☐

Deutsch ☐

Español ☒

Since the language selection feature needs to be available across all pages, we will implement a shared application bar that can be reused consistently throughout the app. To achieve this, we introduce a new parent class named `AppPageView`, which will be extended by each page implementation.

By centralizing the app bar logic in this base class, we not only ensure a uniform interface for language switching, but also create a flexible foundation to support other common UI elements and behaviors across the application.

The `AppPageView` includes newly defined methods for constructing the application bar with integrated language selection controls. It also demonstrates how the session repository is leveraged in this context to manage and update the active locale.

```
abstract class AppPageView extends ALFWidgetView {
  const AppPageView({
    super.key,
    required super.controller,
    super.event,
    super.params,
    super.options = const [ALFWidgetViewOptions.safeArea, ALFWidgetViewOptions.unfocus],
    super.repositories,
  });

  SessionRepository get session {
    return repo<SessionRepository>();
  }

  AppBar buildAppBar({required BuildContext context, required String title}) {
    List<Widget> actions = [_buildLanguageChoice(context: context)];
    return AppBar(backgroundColor: Theme.of(context).colorScheme.inversePrimary, title: Text(title), actions:
actions);
  }

  Widget _buildLanguageChoice({required BuildContext context}) {
    final Map<String, Locale> supportedLocales = const {"English": Locale("en", "UK"), "Deutsch": Locale("de", "DE"),
"Español": Locale("es", "ES")};

    SessionRepository sessionRepo = session;
    return PopupMenuButton<Locale>(
      icon: const Icon(Icons.language),
      onSelected: (locale) async {
        if (locale != sessionRepo.currentLocale) {
          sessionRepo.currentLocale = locale;
        }
      },
      itemBuilder: (BuildContext context) {
        return supportedLocales.keys.map((label) {
          Locale? locale = supportedLocales[label];
          Icon icon = locale == sessionRepo.currentLocale
            ? Icon(Icons.check_circle_outline, color: theme(context).shadowColor)
            : Icon(Icons.circle_outlined, color: theme(context).shadowColor);
          return PopupMenuItem<Locale>(
            value: locale,
            child: Row(
              children: [
                Expanded(child: Text(label)),
                icon,
              ],
            ),
          );
        }).toList();
      },
    );
  }
}
```

The next step is to update the `RootPage`, `HomePage`, and `CounterPage` page implementations to inherit from the newly introduced parent class, `AppPageView`. The session repository will be instantiated within `RootPage` at the start of the application lifecycle. Meanwhile, `HomePage` and `CounterPage` will utilize the shared `buildAppBar()` method to construct their application bars ensuring consistent behavior and UI across all screens.

```
class RootPage extends AppPageView {
  RootPage({super.key})
    : super(
      controller: RootController(),
    )
}
```

```

        repositories: [
          ALFRepositoryProvider<SessionRepository>(create: (context) => SessionRepository()),
          ALFRepositoryProvider<CounterRepository>(create: (context) => CounterRepository()),
        ],
      );
    ...
  }

class HomePage extends AppPageView {
  ...
  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      appBar: buildAppBar(context: context, title: title),
      body: Center(
        ...
      )
    );
  }

class CounterPage extends AppPageView {
  ...
  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      backgroundColor: state[CounterState.backgroundColor],
      appBar: buildAppBar(context: context, title: L10nUtil().translate().page_Counter_appBar_title),
      body: Center( ...
    );
  }
}

```

So far, language selection is available across all pages via the application bar, and the session repository responsible for storing the current locale is initialized in the application's landing page. However, selecting a different language does not yet update the user interface. This happens because the page controllers are not currently subscribed to locale changes.

To resolve this, we will introduce a new application-specific base controller class, similar to how we structured shared page components. All individual controllers will inherit from this base class, which listens to locale changes in the session repository. Whenever the locale is updated, the base controller will trigger an event, prompting the corresponding page to refresh its content using the newly selected language.

```

enum AppPageEvent { changeLanguage }

abstract class AppPageController extends ALFWidgetController {
  AppPageController({super.state, super.status}) {
    register(event: AppPageEvent.changeLanguage, trigger: refreshView);
  }

  @override
  @mustCallSuper
  void init() {
    subscribeRepository<SessionRepository>(event: AppPageEvent.changeLanguage, listenChange:
    SessionChangeType.locale);
  }
}

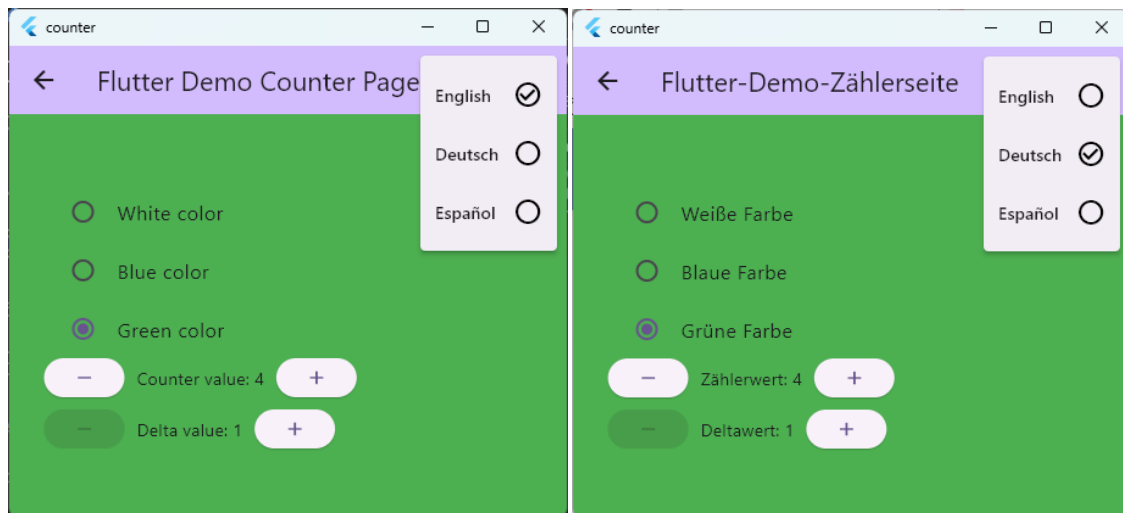
```

Now all controllers are to inherit from the newly introduced base class. It is important to note that if a controller like `HomeController` overrides the `init()` method to listen for other repository updates (e.g. the counter repository), it must explicitly invoke the `init()` method of the base class as part of its own initialization.

To enforce this behavior, the base class's `init()` method is annotated with `@mustCallSuper`. This ensures that any overriding implementation maintains the required initialization sequence and continues to respond to locale changes appropriately.

```
class RootController extends AppPageController { ... }
class HomeController extends AppPageController {
  ...
  @override
  void init() {
    super.init();
    subscribeRepository<CounterRepository>(event: HomeEvent.refresh);
  }
}
class CounterController extends AppPageController { ... }
```

Once a language is selected from the application bar, the texts and labels across the user interface are updated instantly to reflect the new locale.



A quick note regarding the session repository: we currently retrieve it from the context hierarchy each time it is needed via a getter method defined in the parent view class. Since the session repository is designed to exist as a single shared instance throughout the app, repeated lookups can be optimized by caching it.

However, because `ALFWidgetView` is immutable by design, it cannot store the cached reference. Instead, we move this caching responsibility to the associated controller. We will introduce a new attribute in the controller that lazily initializes the session repository on first access. From that point onward, the view can simply reference the controller to retrieve the cached repository instance.

As a result, both `AppPageController` and `AppPageView` will be updated to support this improved access strategy.

```
abstract class AppPageController extends ALFWidgetController {
  SessionRepository? _session;
  ...
  SessionRepository get session {
    _session ??= repo<SessionRepository>();
    return _session!;
  }
}

abstract class AppPageView extends ALFWidgetView {
  ...
  SessionRepository get session {
    AppPageController con = controller as AppPageController;
    return con.session;
  }
  ...
  Widget _buildLanguageChoice({required BuildContext context}) {
    final Map<String, Locale> supportedLocales = const {
      "English": Locale("en", "UK"),
      "Deutsch": Locale("de", "DE"),
      "Español": Locale("es", "ES")
    };
  }
}
```

```

return PopupMenuButton<Locale>(
  icon: const Icon(Icons.language),
  onSelected: (locale) async {
    if (locale != session.currentLocale) {
      session.currentLocale = locale;
    }
  },
  itemBuilder: (BuildContext context) {
    return supportedLocales.keys.map((label) {
      Locale? locale = supportedLocales[label];
      Icon icon =
        locale == session.currentLocale
          ? Icon(Icons.check_circle_outline, color: theme(context).shadowColor)
          : Icon(Icons.circle_outlined, color: theme(context).shadowColor);
      return PopupMenuItem<Locale>(value: locale, child: Row(children: [Expanded(child: Text(label)), icon]));
    }).toList();
  },
);
}
...
}

```

Since the session repository is now cached within the controller, the local instance previously used in the `_buildLanguageChoice()` method is no longer necessary and can be removed.

4 COMMON APPLICATION CLASSES

In the previous chapter, we implemented an application bar that includes a language selection icon visible across all pages. To ensure this feature remains consistent and maintainable throughout the app, we can introduce an additional abstraction layer above the existing Application Layers Framework. This layer will centralize the language selector logic, making it reusable across all page views while keeping the UI cohesive.

```
abstract class AppPageView extends ALFWidgetView { ... }
abstract class AppPageController extends ALFWidgetController { ... }
enum AppPageEvent { changeLanguage }
```

AppPageEvent is not a class, but an enumerator. It is included here for completeness.

While the Application Layers Framework provides a solid foundation of reusable and shared components, certain application-specific requirements can also be generalized for reuse across the entire app. However, these may not be applicable to other projects that follow different design or functional needs. As such, this type of reusable logic is not part of the core framework itself.

To accommodate this, we introduce a set of common application-level classes that extend the respective classes from the Application Layers Framework. These are defined proactively, even when immediate reuse is not required ensuring that future needs can be met without modifying existing implementations, as all app components already inherit from these common classes.

At this stage, the following abstraction classes can be implemented. They mirror the signatures of their base counterparts and, where necessary, define corresponding constructors to maintain compatibility.

No	ALF Parent Class	Common Application Class
1	ALFBeamerUtil	AppBeamerUtil
2	ALFBeamPage	AppBeamPage
3	ALFBeamLocation	AppBeamLocation
4	ALFBeamerDelegate	AppBeamerDelegate
5	ALFBeamer	AppBeamer
6	ALFBeamerState	AppBeamerState
7	ALFBeamerParser	AppBeamerParser
8	ALFBeamerBackButtonDispatcher	AppBeamerBackButtonDispatcher
9	ALFException	AppException
10	ALFRepositoryProvider	AppRepositoryProvider
11	ALFMultiRepositoryProvider	AppMultiRepositoryProvider
12	ALFEntity	AppEntity
13	ALFRepository	AppRepository
14	ALFService	AppService
15	ALFUtil	AppUtil
16	ALFValidator	AppValidator

Below is the implementation of the remaining common application classes, as outlined in the table above.

```
abstract class AppBeamerUtil extends ALFBeamerUtil {}

class AppBeamPage extends ALFBeamPage {
  AppBeamPage({super.title, required super.navPath, super.transition, required super.page});
}
```

```

abstract class AppBeamLocation extends ALFBeamLocation {
  AppBeamLocation({required super.routeInformation, required super.pages});
}

class AppBeamerDelegate extends ALFBeamerDelegate {
  AppBeamerDelegate({required super.initialPath, required super.locations});
}

class AppBeamer extends ALFBeamer {
  const AppBeamer({super.key, required super.routerDelegate, super.createBackButtonDispatcher,
    super.backButtonDispatcher, super.repositories});
  @override
  AppBeamerState createState() => AppBeamerState();
}

class AppBeamerState extends ALFBeamerState {}

class AppBeamerParser extends ALFBeamerParser {
  AppBeamerParser({super.onParse});
}

class AppBeamerBackButtonDispatcher extends ALFBeamerBackButtonDispatcher {
  AppBeamerBackButtonDispatcher({required super.delegate, super.onBack, super.alwaysBeamBack = false,
    super.fallbackToBeamBack = true});
}

class AppException extends ALFException {
  const AppException({super.message, super.code, super.type, super.originalException});
}

class AppRepositoryProvider<T extends AppRepository> extends ALFRepositoryProvider<T> {
  AppRepositoryProvider({required super.create, super.dispose, super.key, super.child, super.lazy});

  AppRepositoryProvider.value({required super.value, super.key, super.child}) : super.value();

  static T of<T>(BuildContext context, {bool listen = false}) {
    return AppRepositoryProvider.of<T>(context, listen: listen);
  }
}

class AppMultiRepositoryProvider extends ALFMultiRepositoryProvider {
  AppMultiRepositoryProvider({required super.providers, required super.child, super.key});
}

abstract class AppEntity extends ALFEntity {}

abstract class AppRepository extends ALFRepository {}

abstract class AppService extends ALFService {}

abstract class AppUtil extends ALFUtil {}

abstract class AppValidator<T, E> extends ALFValidator<T, E> {
  const AppValidator.pure(super.value) : super.pure();
  const AppValidator.dirty(super.value) : super.pure();
}

```

To align with best practices in our sample application, we will update the remaining classes to inherit from the previously defined common application classes, rather than directly extending those provided by the Application Layers Framework.

```

class L10nUtil extends AppUtil { ... }
class CounterEntity extends AppEntity { ... }
class CounterRepository extends AppRepository { ... }
class SessionRepository extends AppRepository { ... }

```

Where applicable, any new classes introduced in the sample application should now inherit from these common application classes, rather than extending the base framework classes directly.

5 CENTRALIZED APPLICATION CONFIGURATIONS

Revisiting the sample application from earlier chapters, you may have noticed that the default locale and the list of supported locales are hardcoded across various code segments. This approach can quickly become difficult to maintain especially as the application scales and requirements evolve.

To improve flexibility and centralize configuration management, we will introduce a new utility class called `ConfigUtil`, implemented as a singleton. This class will serve as a centralized hub for application-level configurations, similar to how `L10nUtil` handles localization.

By relocating all hardcoded locale definitions into `ConfigUtil`, we ensure they are defined in one consistent location making future updates straightforward and reducing the risk of inconsistencies across the codebase.

```
class ConfigUtil extends AppUtil {
  static final ConfigUtil _instance = ConfigUtil._init();
  factory ConfigUtil() {
    return _instance;
  }
  ConfigUtil._init();

  final L10nConfig l10n = const L10nConfig();
}

class L10nConfig {
  const L10nConfig();
  final Locale defaultLocale = const Locale("en", "UK");
  final Map<String, Locale> supportedLocales = const {"English": Locale("en", "UK"), "Deutsch": Locale("de", "DE"),
  "Español": Locale("es", "ES")};
}
```

To further improve maintainability and separation of concerns, configuration values within the utility class can be grouped by category each encapsulated in its own dedicated configuration class. For example, localization-related settings are organized under an `L10nConfig` class, which is instantiated and exposed through the overarching `ConfigUtil` singleton.

This approach allows you to access structured, type-specific configurations in a clean and consistent way. For instance:

```
ConfigUtil().l10n.defaultLocale
ConfigUtil().l10n.supportedLocales
```

By centralizing configuration access through a shared utility, updates become significantly easier to manage especially as the application grows in complexity.

With the application configurations now centralized, the next step is to identify and update the various areas in the codebase where these values were previously hardcoded.

The default locale that was originally defined directly within the `L10nUtil` class will now reference the centralized configuration instead, ensuring consistency and making future updates easier to manage.

```
class L10nUtil extends AppUtil {
  ...
  L10nUtil._init() {
    locale = ConfigUtil().l10n.defaultLocale;
  }
}
```

```
...
}
```

The supported locales are hardcoded within the `AppPageView` and `RootPage` classes. These implementations will be updated to reference the centralized configuration instead, improving maintainability and consistency across the application.

```
abstract class AppPageView extends ALFWidgetView {
    ...
    Widget _buildLanguageChoice({required BuildContext context}) {
        final Map<String, Locale> supportedLocales = ConfigUtil().l10n.supportedLocales;
        ...
    }
    ...
}

class RootPage extends AppPageView {
    ...
    Widget build(BuildContext context) {
        return MaterialApp(
            ...
            supportedLocales: ConfigUtil().l10n.supportedLocales.values,
            ...
        );
    }
}
```

The `MaterialApp` class requires a list of supported locales, while our configuration defines them as a dictionary. To align with the expected format, we will extract the values from this dictionary and pass them as a list to the `supportedLocales` parameter of `MaterialApp`.

To add additional application configurations, simply define a new configuration class tailored to the specific category of settings. Once created, instantiate and reference it within the `ConfigUtil` class to make it accessible throughout the application.

The following is an illustrative example and is not utilized in the current sample application.

```
class ConfigUtil {
    ...
    final AppwriteConfig appwrite = const AppwriteConfig();
}

class AppwriteConfig {
    const AppwriteConfig();

    final String endpoint = "https://appwrite.example.com/v1";
    final String projectId = "1234567890abcdefghij";
    final String storageProfilePhotos = "647dddefr854876jdj";
    final String verificationUrl = "https://localhost/test";
}
```

Accessing these configurations follows the same pattern used for localization settings, ensuring consistency and simplifying retrieval across different configuration types.

```
ConfigUtil().appwrite.endpoint
ConfigUtil().appwrite.projectId
ConfigUtil().appwrite.storageProfilePhotos
ConfigUtil().appwrite.verificationUrl
```

Configurations may also vary depending on the environment in which the application is running (e.g. development, testing, production) or the platform it supports (e.g. iOS, Android, Windows). These differences may require platform- or environment-specific settings to be defined separately.

Determining the active environment or platform is not the focus of this chapter, but centralizing these configurations ensures they are easier to manage, update, and maintain as the application evolves.

6 CONSOLIDATED PACKAGE IMPORT

In earlier chapters, we intentionally omitted import statements from the sample application's code snippets to keep the focus on core concepts. Including all relevant imports, especially when classes are distributed across multiple files and third-party libraries, can be visually distracting and shift attention away from the aspects we aimed to highlight.

Now, for completeness, let us examine the full implementation of `RootPage`, this time including the necessary import statements.

```
import 'package:application_layers_framework/application_layers_framework.dart';
import 'package:flutter/material.dart';

import '../common/alf/app_view_controller.dart';
import '../common/l10n/l10n.dart';
import '../common/util/config_util.dart';
import '../common/util/l10n_util.dart';
import '../model/repository/counter_repository.dart';
import '../model/repository/session_repository.dart';
import '../home/home_page.dart';
import 'root_controller.dart';

class RootPage extends AppPageView {
  RootPage({super.key})
    : super(
        controller: RootController(),
        repositories: [
          ALFRepositoryProvider<SessionRepository>(create: (context) => SessionRepository()),
          ALFRepositoryProvider<CounterRepository>(create: (context) => CounterRepository()),
        ],
      );

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: L10nUtil().translate().page_Root_app_title,
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: ConfigUtil().l10n.supportedLocales.values,
      locale: L10nUtil().locale,
      debugShowCheckedModeBanner: false,
      theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple), useMaterial3: true),
      home: HomePage(title: L10nUtil().translate().page_Root_home_title),
    );
  }
}
```

Managing package imports across the application can easily become unstructured and error-prone especially as code evolves and new files or dependencies are introduced. Beyond implementing core application logic, developers must also ensure that each Dart file contains the appropriate and up-to-date import statements.

To simplify this process, we adopt a centralized import strategy, similar to the configuration approach introduced in the previous chapter. This involves creating a consolidated import file conventionally named `/common/export.dart` that aggregates all internal Dart files for consistent access and easier maintenance.

Broadly, the application relies on two types of packages:

- **Project-specific Dart files** defined within the Flutter application
- **Third-party packages** declared in `pubspec.yaml`

For project-specific files, we will automate the aggregation of imports using a Python script. This script scans the Flutter project for Dart files and generates the `export.dart` file dynamically. To keep the export list clean, we specify a set of exclusions to omit files that should not be exported ensuring the generated file reflects only what is needed without manual oversight.

```
import os

outputFile = "lib/common/export.dart"
exclusions = [
    "/main.dart",
    "/common/export.dart",
    "/common/import.dart",
]

with open(outputFile, "w") as file:
    for root, d_names, f_names in os.walk("lib"):
        for f in f_names:
            filename = os.path.join(root, f)[3:].replace("\\", "/")
            if filename not in exclusions and filename.endswith(".dart"):
                file.write('export "{0}";\n'.format(filename))
```

After running the script on our sample application, the export file is automatically generated to consolidate all implemented Dart files. Whenever new files are added or existing ones removed, rerunning the script will instantly update the export list keeping your internal imports clean, accurate, and effortlessly maintainable.

```
export "/common/alf/app_beamer.dart";
export "/common/alf/app_exception.dart";
export "/common/alf/app_model.dart";
export "/common/alf/app_util.dart";
export "/common/alf/app_validator.dart";
export "/common/alf/app_view_controller.dart";
export "/common/l10n/l10n.dart";
export "/common/l10n/l10n_de.dart";
export "/common/l10n/l10n_en.dart";
export "/common/l10n/l10n_es.dart";
export "/common/util/config_util.dart";
export "/common/util/l10n_util.dart";
export "/model/entity/counter_entity.dart";
export "/model/repository/counter_repository.dart";
export "/model/repository/session_repository.dart";
export "/page/counter/counter_controller.dart";
export "/page/counter/counter_page.dart";
export "/page/home/home_controller.dart";
export "/page/home/home_page.dart";
export "/page/root/root_controller.dart";
export "/page/root/root_page.dart";
```

Next, we move on to the second category: third-party packages. These will be managed manually in a dedicated import file located at `/common/import.dart`.

In our sample application, scan all Dart files for import statements referencing third-party libraries those declared in `pubspec.yaml` and include corresponding export statements in the `import.dart` file. This file should also include the previously generated `export.dart`, consolidating both internal and external dependencies into a centralized module for cleaner project-wide access.

```
export 'package:flutter/material.dart';
export 'package:application_layers_framework/application_layers_framework.dart';

export '/common/export.dart';
```

With both package types now consolidated, individual Dart files no longer need to manually list extensive import statements. Instead, a single one-liner importing `import.dart` suffices streamlining the setup and allowing developers to focus on implementing application logic rather than managing dependencies.

```
import '/common/import.dart';

class RootPage extends AppPageView {
  ...
}
```

Apply this same approach across all Dart files in the sample application. Regardless of changes to file names, directory structures, or package references, only the centralized `export.dart` and `import.dart` files need to be updated eliminating the need to modify individual import statements and preserving consistency throughout the codebase.

NAMING CONFLICTS

When using consolidated package imports, naming conflicts may occasionally occur especially if your class names overlap with those from third-party libraries. The recommended approach for resolving such conflicts is to rename your custom classes to avoid ambiguity.

In cases where conflicting names originate from third-party packages, Dart provides mechanisms to handle them gracefully. You can refer to **Managing Conflicting Imports in Dart: A Quick Guide** (<https://saw2110.medium.com/managing-conflicting-imports-in-dart-a-quick-guide-36d055e6ca75>) for practical strategies to manage these scenarios effectively.

7 STRUCTURING YOUR FLUTTER PROJECT

FOLDER AND FILE STRUCTURE

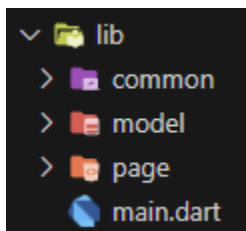
In the previous chapters, we implemented numerous classes, but their file locations and folder structures were not explicitly shown. Since Flutter offers flexibility in how project files are organized including the ability to customize the main entry point, it is important to establish a clear and maintainable folder structure.

To support scalability and ease of navigation, we aim to separate application components logically. This organization ensures that each type of functionality is easy to locate, understand, and maintain as the codebase evolves.

Reflecting on the components covered so far, a typical Flutter application may include:

- Common and reusable code
- Entities
- Pages
- Controllers (handling events and state)
- Repositories
- Service providers

The Application Layers Framework itself is packaged as an external library and imported into the application just like any third-party dependency. The rest of the components can be neatly organized using three top-level folders to keep the structure simple and intuitive.

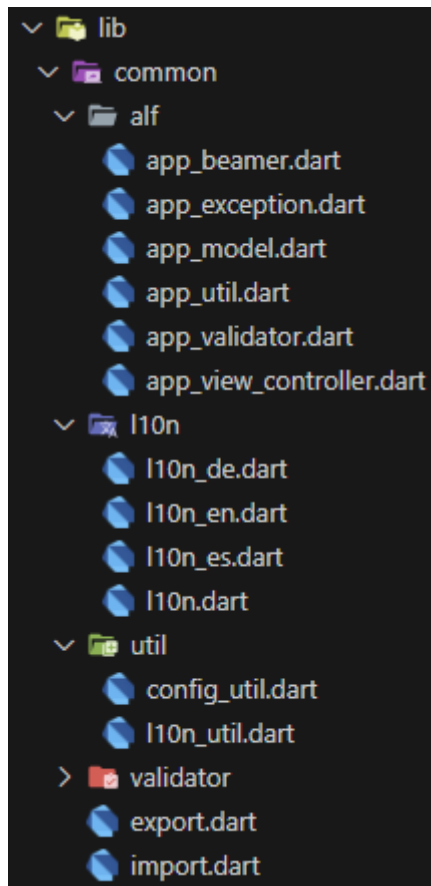


The `common` folder serves as the central location for all shared and reusable code across the application. This includes consolidated package imports, as discussed in a previous chapter, as well as application-specific utilities that help enforce consistent behavior and visual styling such as page backgrounds, app bars, logout handling, and language selection.

These shared components can be implemented as abstract parent classes for pages, controllers, and other elements, and are conventionally placed under the `lib/common/alf/` folder in our sample application.

Additionally:

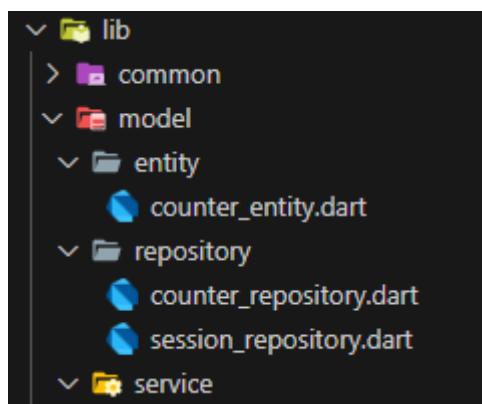
- Localization helper classes are generated under `lib/common/l10n/`
- Utility classes are organized in `lib/common/util/`
- The `lib/common/validator/` folder is reserved for field validation logic, which will be covered in a future chapter; for now, consider it a placeholder



The `model` folder houses classes related to business logic, including entities, service providers, and repositories. These are organized into dedicated subfolders:

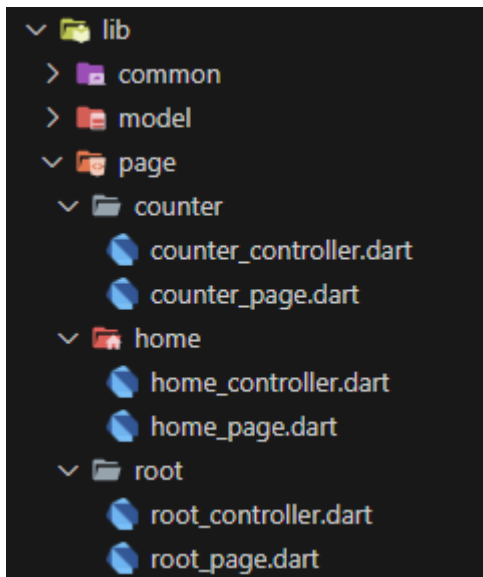
- `lib/model/entity/` for entity definitions
- `lib/model/service/` for service provider classes
- `lib/model/repository/` for repository implementations

Each class resides in its own Dart file to maintain modularity and improve readability. While service providers have not yet been introduced, they will be covered in a later chapter. For now, the `service` folder serves as a placeholder in preparation for those upcoming implementations.



The final top-level folder is the `page` folder, which contains subfolders for each individual page implementation. Within each subfolder, the logic and UI components are separated for clarity which is page

layout and presentation are defined in a dedicated page Dart file, while business logic and state management are handled in a corresponding controller Dart file.



Lastly, while the entry point can be customized in Flutter, we retain `lib/main.dart` as the standard location for the application's startup logic.

NAMING CONVENTIONS

Folder naming conventions follow Dart's official style guidelines, as outlined in the **Effective Dart: Style** (<https://dart.dev/effective-dart/style>) documentation.

These conventions also extend to files and classes, which should be suffixed based on their functional roles as illustrated in the following table.

No	Type	Dart File		Class/Enum Name	
		Suffix	Example	Suffix	Example
1	Utility	<code>_util</code>	<code>config_util.dart</code>	Util	<code>ConfigUtil</code>
2	Validator	<code>_validator</code>	<code>email_validator.dart</code>	Validator	<code>EmailValidator</code>
3	Business entity	<code>_entity</code>	<code>counter_entity.dart</code>	Entity	<code>CounterEntity</code>
4	Service provider	<code>_service</code>	<code>chat_service.dart</code>	Service	<code>ChatService</code>
5	Repository	<code>_repository</code>	<code>counter_repository.dart</code>	Repository	<code>CounterRepository</code>
6	Page / View	<code>_page</code>	<code>counter_page.dart</code>	Page	<code>CounterPage</code>
7	Page state	<code>_controller</code>	<code>counter_controller.dart</code>	State	<code>CounterState</code>
8	Event	<code>_controller</code>	<code>counter_controller.dart</code>	Event	<code>CounterEvent</code>
9	Controller	<code>_controller</code>	<code>counter_controller.dart</code>	Controller	<code>CounterController</code>

These naming conventions have already been adopted and consistently applied throughout our sample application, ensuring alignment with Dart's recommended style and promoting a clean, maintainable codebase.

8 NAVIGATION AND ROUTING

Navigating between screens is a fundamental part of building interactive applications and Flutter offers a flexible and powerful system for handling it. Whether you are transitioning between a splash screen and a login page, navigating through tabs, or managing deep links and nested routes, Flutter equips you with a comprehensive set of tools to orchestrate smooth and intuitive navigation flows.

This chapter explores the navigation and routing requirements when working with the Application Layers Framework. It includes a review and evaluation of relevant packages, guiding you toward building clean, maintainable navigation flows that elevate the overall user experience.

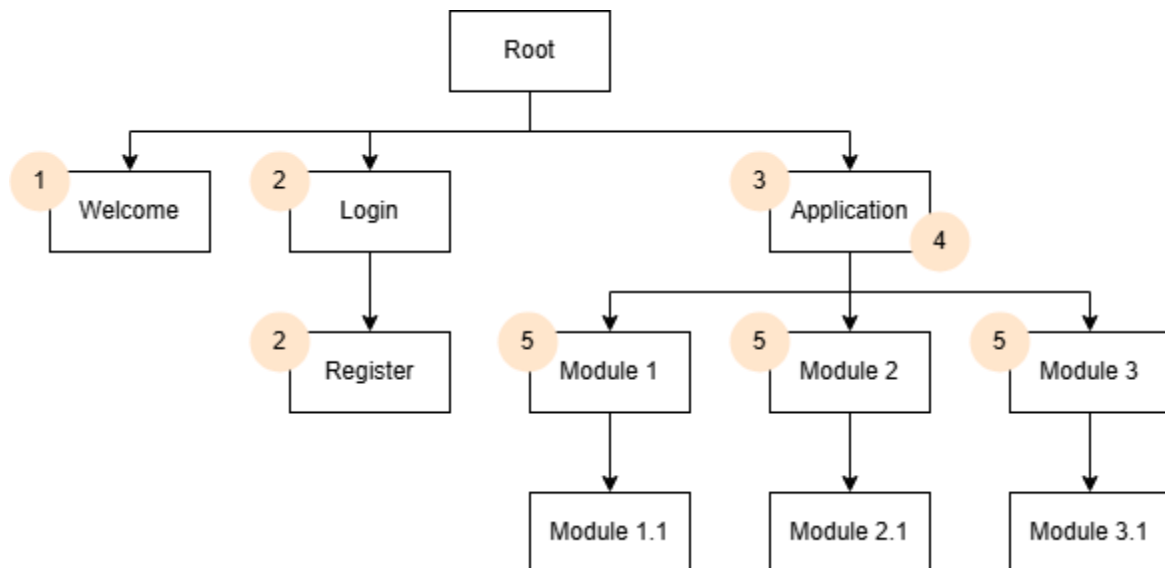
8.1 REQUIREMENTS AND EVALUATION

REQUIREMENTS

Consistent and intuitive navigation is a cornerstone of strong user experience (UX) and often plays a decisive role in user acceptance of an application. But what defines good navigation and routing in practice?

The diagram below highlights key elements of a smooth and user-friendly navigation flow:

1. Displaying a **welcome and introduction screen** when the app is launched for the first time
2. Presenting the **login screen** after the introduction, with an option for **user registration**
3. Transitioning to the main application upon **successful authentication**
4. Enabling seamless **navigation across core sections** via a navigation bar
5. Supporting **nested navigation flows** for individual modules to keep transitions clean and structured



THIRD-PARTY PACKAGES

Flutter introduces the Navigator 2.0 framework to support advanced navigation patterns with maximum flexibility. However, while powerful, its implementation details can be quite complex posing usability challenges for developers seeking clean and maintainable navigation flows.

To strike a balance between flexibility and developer friendliness, several third-party packages wrap around the Navigator 2.0 API, abstracting much of its complexity while still offering customizable and robust routing capabilities. Among these, two stand out: **GoRouter** and **Beamer**.

- **GoRouter** (https://pub.dev/packages/go_router) has been acknowledged by the Flutter team as a candidate for a de facto industry standard. It simplifies routing configuration and supports declarative navigation. However, certain limitations have emerged particularly around state management compatibility when used in conjunction with the Application Layers Framework. These caveats can affect the overall architectural cohesion of the app.
- **Beamer** (<https://pub.dev/packages/beamer>) offers a similar developer experience while addressing some of GoRouter's integration limitations. Beamer supports nested navigation and route-aware

widgets, aligning more closely with the structure of the Application Layers Framework. While concerns may arise regarding long-term package maintenance, Beamer is actively developed and well-established within the Flutter community as of this writing. That said, the same maintenance concerns apply to GoRouter, given the evolving ecosystem.

It is worth noting that other navigation packages exist and may be equally suitable. However, since Beamer adequately meets the framework's routing requirements, additional evaluations were deemed unnecessary for the scope of this project.

PACKAGE EVALUATION

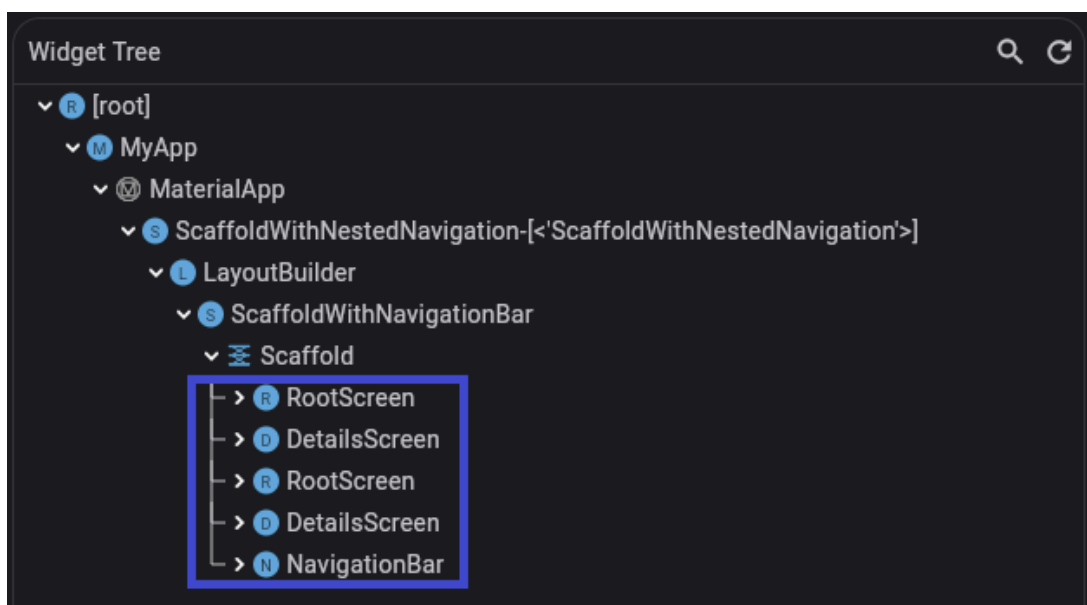
There are already excellent tutorials available online that demonstrate how to use **GoRouter** and **Beamer** in Flutter. As such, this section will not delve into step-by-step usage of these packages. Instead, it will reference the following article and focus on outlining the rationale behind selecting Beamer over GoRouter:

Flutter Bottom Navigation Bar with Stateful Nested Routes using GoRouter (codewithandrea.com)
[\(https://codewithandrea.com/articles/flutter-bottom-navigation-bar-nested-routes-gorouter/\)](https://codewithandrea.com/articles/flutter-bottom-navigation-bar-nested-routes-gorouter/)

GOROUTER

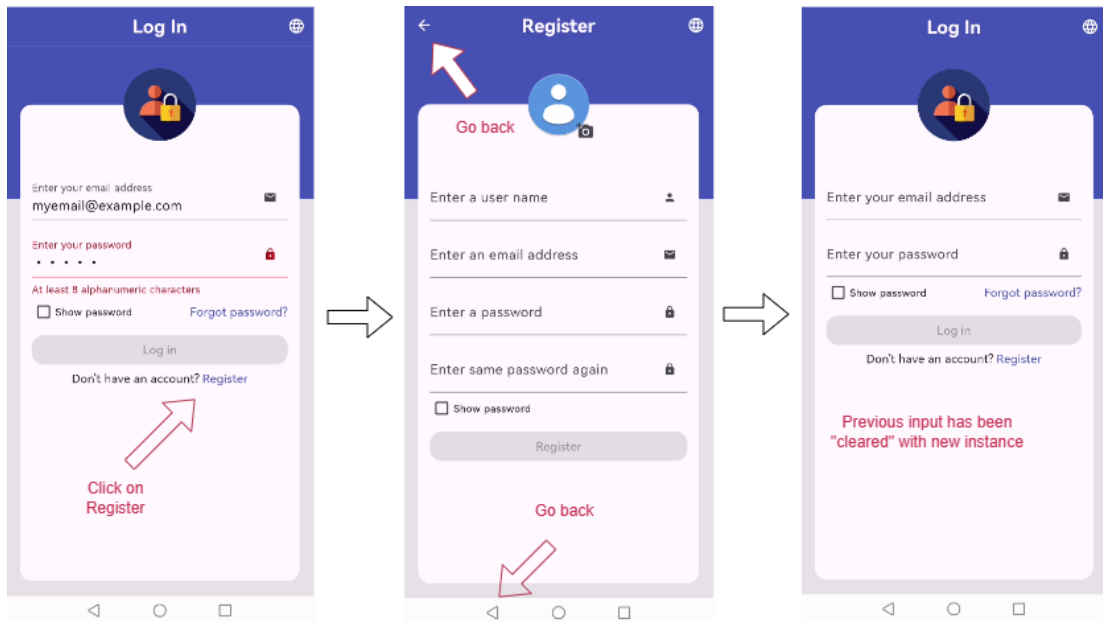
According to the referenced article, using GoRouter solely for navigation works well and delivers a smooth experience. However, when integrating it with state management in the Application Layers Framework, two key limitations emerge.

1. A closer inspection of the widget hierarchy especially around the navigation bar reveals that all pages across application modules reside at the same navigation depth. This structure makes it challenging to scope and isolate state objects specifically to individual modules. Typically, state objects are stored at the widget level that defines the navigation bar. Unfortunately, doing so results in these state objects being accessible across the entire application rather than being restricted to their respective modules, reducing modularity and increasing the risk of unintended state sharing.



2. When navigating back, commonly referred to as a `pop` operation, a new instance of the page is created, rather than restoring the previous one. As a result, any prior page state, including user input, is lost and reinitialized from scratch. To preserve page state, developers must manually restore the

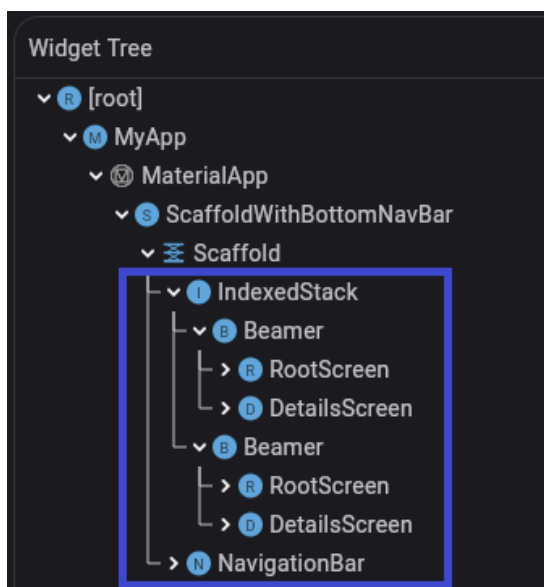
page to its previous condition before navigation occurred. While one might argue that state preservation falls outside the responsibilities of a navigation framework, from a UX standpoint, retaining page state across navigations is often a highly desirable feature. Without it, users may experience disruptions or frustration when returning to a screen they previously interacted with.



BEAMER

Although the referenced article does not walk through Beamer's implementation in detail, it does provide source code demonstrating the same application flow using Beamer. By running this code and inspecting the application behavior, we can revisit the earlier points that highlighted limitations of the GoRouter package.

1. Examining the widget explorer reveals a clear separation between application modules particularly at the navigation bar level. Each module maintains its own `Beamer` instance, allowing state objects to be scoped and stored independently within the module. This modular structure supports better isolation and aligns well with the architectural goals of the Application Layers Framework.

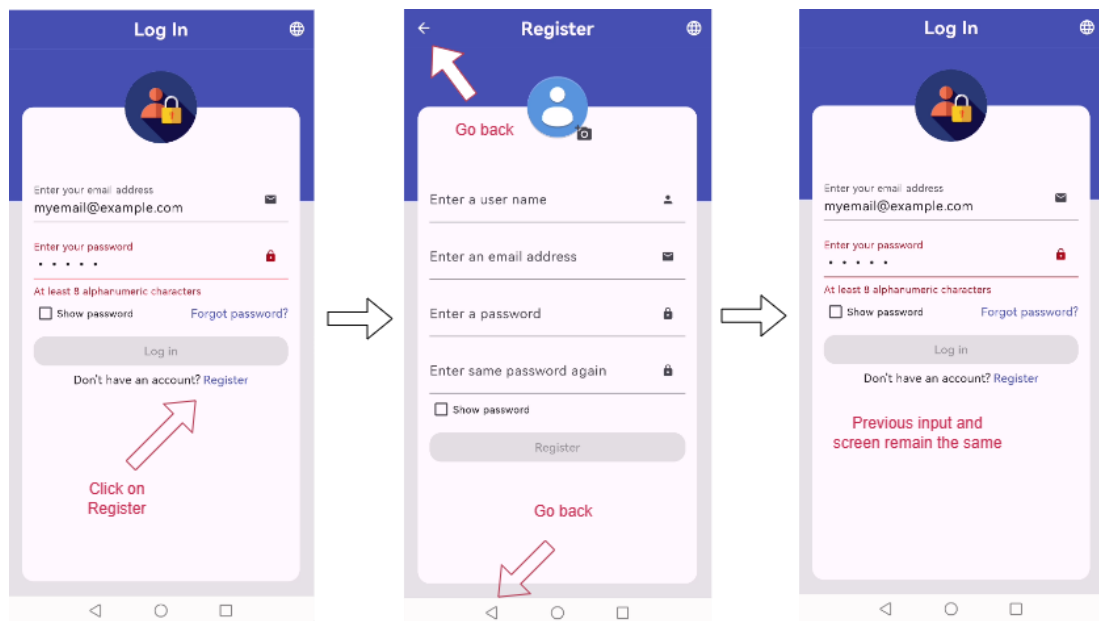


Storing a state object (repository) within a Beamer instance is supported and can be accomplished using the approach shown in the code snippet below. This technique allows you to embed module-specific state alongside other Beamer configurations enabling better encapsulation and modular navigation.

```
...
body: IndexedStack(
  index: state[NavigationState.tabIndex],
  children: [
    AppBeamer(
      routerDelegate: BeamerUtil().navigationHomeOneDelegate,
      repositories: [AppRepositoryProvider<CounterRepository>(create: (context) =>
CounterRepository(navigation: 1))],
    ),
    AppBeamer(
      routerDelegate: BeamerUtil().navigationHomeTwoDelegate,
      repositories: [AppRepositoryProvider<CounterRepository>(create: (context) =>
CounterRepository(navigation: 2))],
    ),
    AppBeamer(routerDelegate: BeamerUtil().navigationChatDelegate),
  ],
),
...

```

2. Navigating backward, commonly known as a `pop` operation, automatically preserves the previous page's state without requiring any additional setup or custom logic. This behavior is built into the navigation flow and ensures that user input or page-specific data remains intact when returning to a previously visited screen. The following example illustrates how this works in practice.



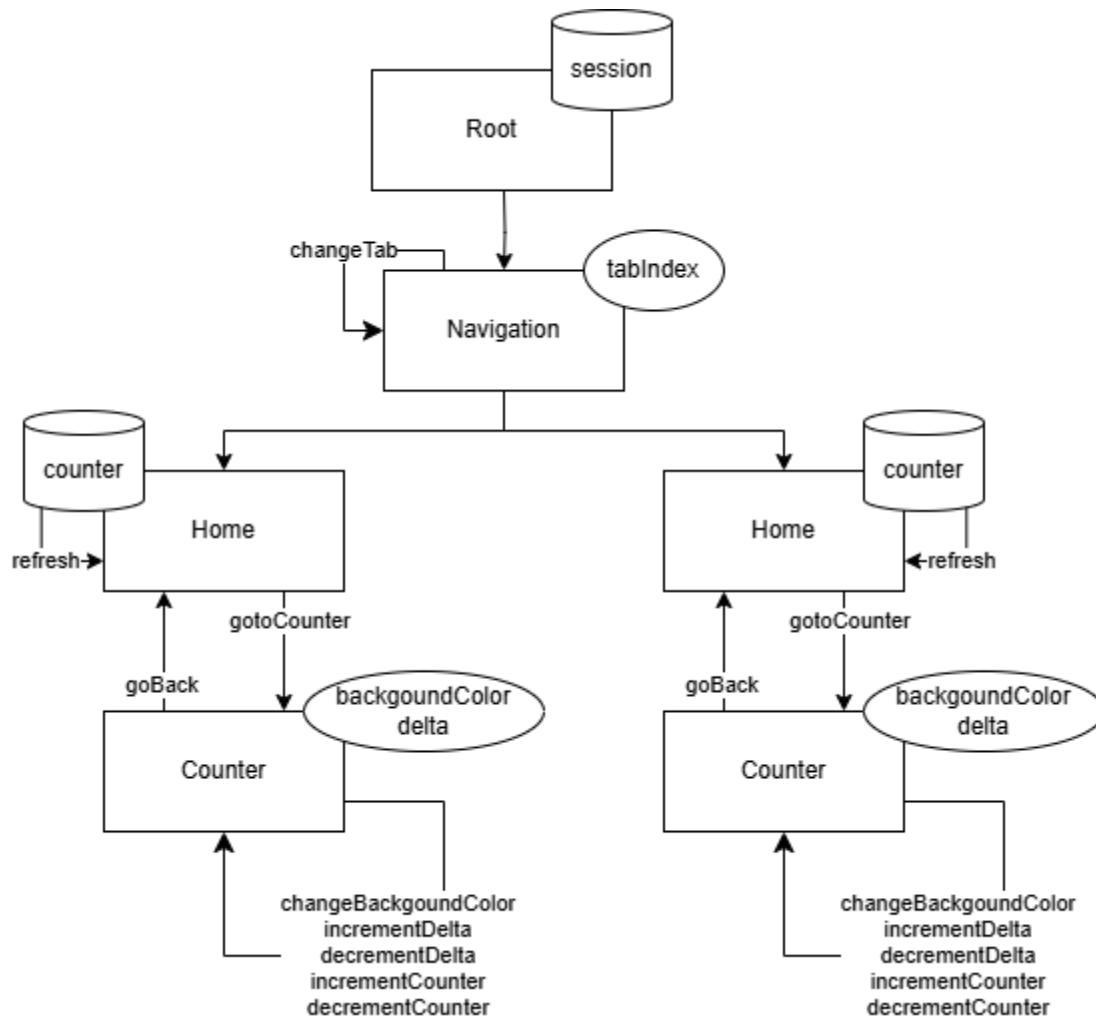
PACKAGE SELECTION

While both packages take a similar approach to abstracting the complexity of Flutter's Navigator 2.0 framework making navigation and routing more approachable, key technical differences emerge upon closer inspection. These differences ultimately determine which solution best aligns with the architectural goals.

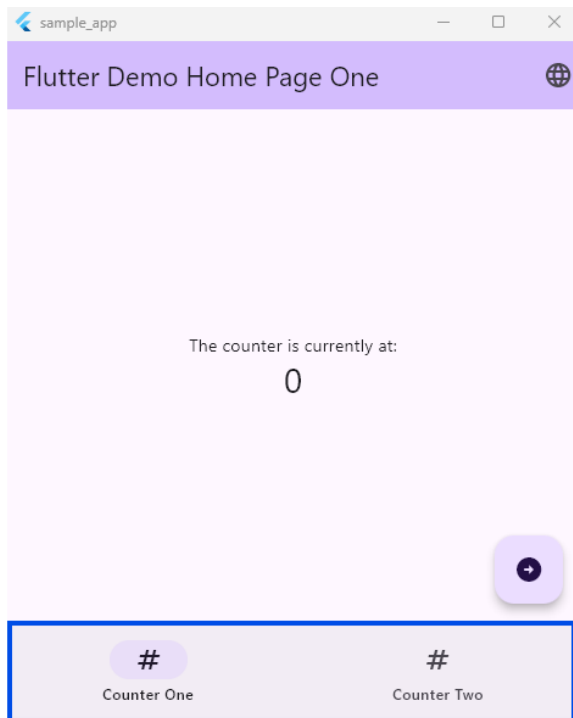
After evaluating their integration capabilities, Beamer has been chosen for use with the Application Layers Framework, as it avoids the limitations encountered with GoRouter and offers better compatibility with the framework's state management and modular structure.

8.2 INTEGRATION WITH ALF

In our sample application, the initial navigation flow consists of two pages cycling back and forth in a straightforward manner. For demonstration purposes, we now extend this setup by introducing a second navigation path that reuses the same page components. However, each path is backed by a separate counter repository, allowing each instance to maintain its own independent counter state. As a result, although the pages appear identical, the state remains scoped and isolated within each navigation context.



In addition to duplicating the same navigation path, the diagram also introduces a new **Navigation** page that leverages Flutter's `IndexedStack` widget to seamlessly switch between different navigation flows. This page uses a bottom navigation bar with two tabs, allowing users to toggle between the navigation stacks while preserving each stack's state independently. The screenshot below illustrates how the tabs are configured to facilitate this interaction.



The `Navigation` page extends the `Application Layers Framework`, consistent with the approach used in other pages. Within its implementation, the `Beamer` package is employed to construct two independent navigation flows and embedded within an `IndexedStack` widget.

The currently selected tab index is managed within the page's state, enabling seamless switching between navigation paths. To maintain isolated state for each path, a dedicated counter repository is instantiated per flow each receiving a constructor parameter that identifies the navigation path it belongs to.

```
class NavigationPage extends AppPageView {
  NavigationPage({super.key}) : super(controller: NavigationController());

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: IndexedStack(
        index: state[NavigationState.tabIndex],
        children: [
          AppBeamer(
            routerDelegate: BeamerUtil().navigationHomeOneDelegate,
            repositories: [AppRepositoryProvider<CounterRepository>(create: (context) => CounterRepository(navigation: 1))],
          ),
          AppBeamer(
            routerDelegate: BeamerUtil().navigationHomeTwoDelegate,
            repositories: [AppRepositoryProvider<CounterRepository>(create: (context) => CounterRepository(navigation: 2))],
          ),
        ],
      ),
      bottomNavigationBar: NavigationBar(
        selectedIndex: state[NavigationState.tabIndex],
        destinations: [
          NavigationDestination(label: L10nUtil().translate().page_Navigation_counterOne_navbarLabel, icon: const Icon(Icons.numbers)),
          NavigationDestination(label: L10nUtil().translate().page_Navigation_counterTwo_navbarLabel, icon: const Icon(Icons.numbers)),
        ],
        onDestinationSelected: (value) => trigger(event: NavigationEvent.changeTab, params: value),
      ),
    );
  }
}
```

```
}
}
```

Tab switching is not handled automatically out of the box. Instead, when a different tab is selected, an event is dispatched to the controller, which updates the selected tab index and refreshes the page accordingly. Beyond this interaction, the controller implementation follows the same pattern established in previous pages.

```
enum NavigationEvent { changeTab }

enum NavigationState { tabIndex }

class NavigationController extends AppPageController {
  NavigationController() : super(state: {NavigationState.tabIndex: 0}) {
    register(event: NavigationEvent.changeTab, trigger: changeTab);
  }

  void changeTab({dynamic params}) {
    update(state: {NavigationState.tabIndex: params});
  }
}
```

Whenever new Dart files are added to the project, rerun the export script described in chapter 6 (Consolidated Package Import) to regenerate the central export file. This ensures that class definitions from newly added files become accessible to other parts of the application through shared imports maintaining consistency and avoiding manual updates across multiple Dart files.

You may have noticed a new utility class, `BeamerUtil`, referenced within the `NavigationPage`. This class encapsulates the navigation logic built around the **Beamer** package, streamlining route handling and improving code modularity.

`BeamerUtil` extends functionality provided by the Application Layers Framework, inheriting shared navigation capabilities that are commonly reused throughout the app. These include methods that navigate to specific destinations based on application workflow logic, helping enforce consistency and reduce boilerplate across navigation scenarios.

```
import '/common/import.dart';

class NavigationLocation extends AppBeamLocation {
  NavigationLocation({required super.routeInformation, required super.pages});
}

class NavigationHomeOneLocation extends AppBeamLocation {
  NavigationHomeOneLocation({required super.routeInformation, required super.pages});
}

class NavigationHomeTwoLocation extends AppBeamLocation {
  NavigationHomeTwoLocation({required super.routeInformation, required super.pages});
}

class BeamerUtil extends AppBeamerUtil {
  static final BeamerUtil _instance = BeamerUtil._init();

  final String navigationPath = "/navigation";
  final String navigationHomeOnePath = "/navigation/home_one";
  final String navigationHomeOneCounterOnePath = "/navigation/home_one/counter_one";
  final String navigationHomeTwoPath = "/navigation/home_two";
  final String navigationHomeTwoCounterTwoPath = "/navigation/home_two/counter_two";

  late final AppBeamerDelegate rootDelegate;
  late final AppBeamerDelegate navigationHomeOneDelegate;
  late final AppBeamerDelegate navigationHomeTwoDelegate;

  factory BeamerUtil() {
```

```

    return _instance;
  }

  @override
  String get initialPath {
    return navigationPath;
  }

  BeamerUtil._init() {
    rootDelegate = AppBeamerDelegate(
      initialPath: navigationPath,
      locations: {
        navigationPath: (routeInformation) => NavigationLocation(
          routeInformation: routeInformation,
          pages: [AppBeamPage(navPath: navigationPath, page: NavigationPage())],
        ),
      },
    );
    navigationHomeOneDelegate = AppBeamerDelegate(
      initialPath: navigationHomeOnePath,
      locations: {
        navigationHomeOnePath: (routeInformation) => NavigationHomeOneLocation(
          routeInformation: routeInformation,
          pages: [
            AppBeamPage(navPath: navigationHomeOnePath, page: HomePage()),
            AppBeamPage(navPath: navigationHomeOneCounterOnePath, page: CounterPage()),
          ],
        ),
      },
    );
    navigationHomeTwoDelegate = AppBeamerDelegate(
      initialPath: navigationHomeTwoPath,
      locations: {
        navigationHomeTwoPath: (routeInformation) => NavigationHomeTwoLocation(
          routeInformation: routeInformation,
          pages: [
            AppBeamPage(navPath: navigationHomeTwoPath, page: HomePage()),
            AppBeamPage(navPath: navigationHomeTwoCounterTwoPath, page: CounterPage()),
          ],
        ),
      },
    );
  }
}

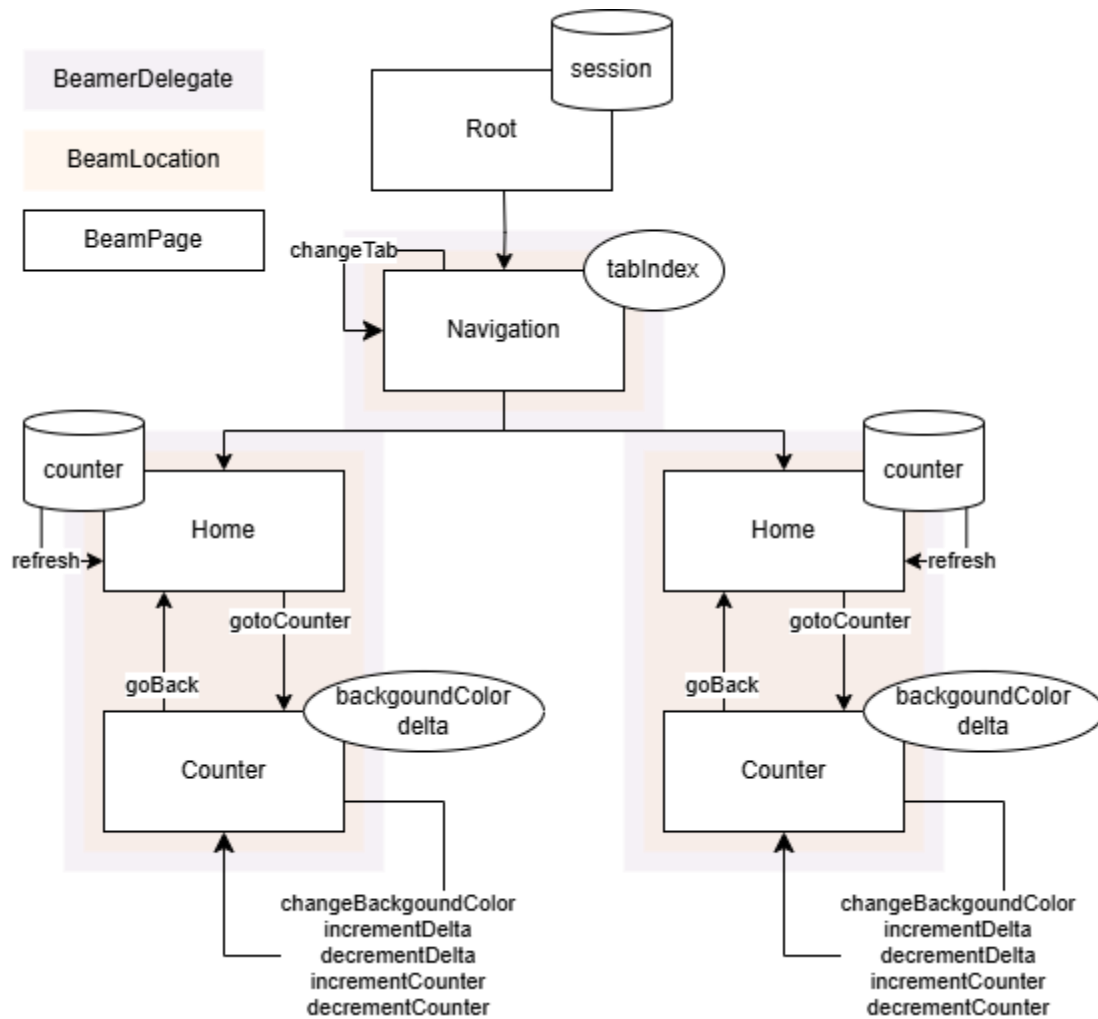
```

Since the Beamer package is already encapsulated within the Application Layers Framework, it does not need to be explicitly listed in the project's `pubspec.yaml` file. Its functionality is seamlessly integrated through the framework, reducing the need for direct dependency management.

We will not dive into every technical detail of how to implement navigation using Beamer, as its official documentation already offers comprehensive coverage. Instead, this section highlights the key components that make up Beamer's navigation architecture and how they are applied within our sample application.

- Each navigation path is managed by a `BeamerDelegate`, inherited in our project as `AppBeamerDelegate`. The application begins with a single root delegate, but additional delegates are introduced when navigation branches into separate flows. In our example, navigation branches into two distinct Home flows, each controlled by its own `BeamerDelegate` instance.
- A `BeamLocation`, inherited as `AppBeamLocation`, defines the different route destinations within a specific navigation path. For each logical location, a dedicated class is implemented extending `BeamLocation`.
- A `BeamPage`, inherited as `AppBeamPage`, represents the actual screen that is rendered upon navigation. This encapsulates the widget tree for the corresponding route.

For visual clarity, these components are highlighted in the navigation diagram below, illustrating how they work together to organize and modularize the routing structure.



Previously, the `Root` page used `Home` as its default entry screen. With the transition to using `Beamer` for page navigation, the `Root` implementation has been updated to rely on the root `BeamerDelegate` for routing control as shown in the code snippet below.

Additionally, the creation of the `MaterialApp` instance has been refactored to use a dedicated class method for navigation and routing.

As part of this reorganization, the counter repository has been removed from the `Root`'s scope. Instead, each navigation path defined within the `Navigation` component now manages its own counter repository, while the session repository remains accessible within `Root` for shared session-level state management.

```
class RootPage extends AppPageView {
  RootPage({super.key})
    : super(
      controller: RootController(),
      repositories: [ALFRepositoryProvider<SessionRepository>(create: (context) => SessionRepository())],
    );

  @override
  Widget build(BuildContext context) {
    return MaterialApp.router(
      title: L10nUtil().translate().page_root_app_title,
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: ConfigUtil().l10n.supportedLocales.values,
    );
  }
}
```

```

        locale: L10nUtil().locale,
        debugShowCheckedModeBanner: false,
        theme: ThemeData(colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple), useMaterial3: true),
        routerDelegate: BeamerUtil().rootDelegate,
        routeInformationParser: AppBeamerParser(),
        backButtonDispatcher: AppBeamerBackButtonDispatcher(delegate: BeamerUtil().rootDelegate),
    );
}
}

```

The `Navigation` page introduces new labels, and the application bar titles for the `Home` and `Counter` pages will vary depending on the active navigation path. To support this dynamic labeling, new entries will be added to the base language file.

Following this update, the established procedure outlined in chapter 3 (Internationalization) should be used to provide translations for these entries across the target languages.

```

{
  ...
  "page_Navigation_counterOne_navbarLabel": "Counter One",
  "page_Navigation_counterTwo_navbarLabel": "Counter Two",
  ...
  "page_Home_appBarOne_title": "Flutter Demo Home Page One",
  "page_Home_appBarTwo_title": "Flutter Demo Home Page Two",
  ...
  "page_Counter_appBarOne_title": "Flutter Demo Counter Page One",
  "page_Counter_appBarTwo_title": "Flutter Demo Counter Page Two",
  ...
}

```

Remember that the counter repository includes an identifier, which must be specified when invoking its constructor. This indicator ensures that each instance is correctly associated with its respective navigation path or context.

```

class CounterRepository extends AppRepository {
  ...
  final int navigation;
  CounterRepository({required this.navigation});
  ...
}

```

This indicator determines which application bar title should be displayed by each corresponding page. Since the title is now dynamically resolved based on this context, `HomePage` no longer requires the title to be passed explicitly through its constructor.

```

class HomePage extends AppPageView {
  HomePage({super.key}) : super(controller: HomeController());

  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      appBar: buildAppBar(
        context: context,
        title: counterRepository.navigation == 1 ? L10nUtil().translate().page_Home_appBarOne_title :
        L10nUtil().translate().page_Home_appBarTwo_title,
      ),
      ...
    );
  }
}

```

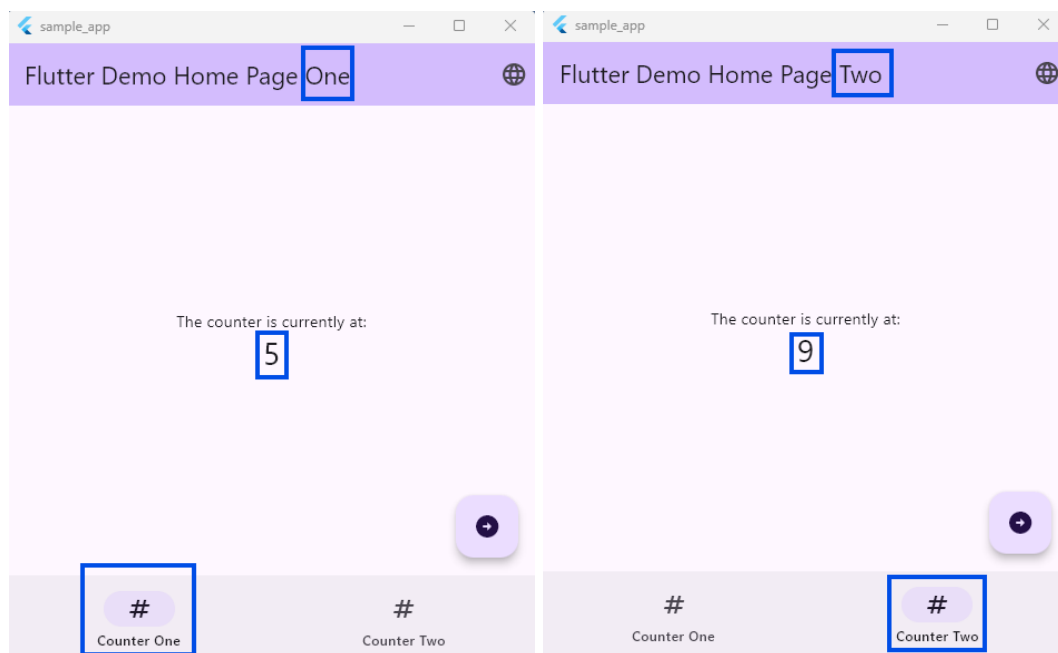
```
class CounterPage extends AppPageView {
  CounterPage({super.key}) : super(controller: CounterController());

  @override
  Widget build(BuildContext context) {
    CounterRepository counterRepository = repo<CounterRepository>();
    return Scaffold(
      backgroundColor: state[CounterState.backgroundColor],
      appBar: buildAppBar(
        context: context,
        title: counterRepository.navigation == 1 ? L10nUtil().translate().page_Counter_appBarOne_title :
        L10nUtil().translate().page_Counter_appBarTwo_title,
      ),
      ...
    );
  }
}
```

Additionally, since navigation is now handled using the Beamer package, all navigation operations are performed using the methods encapsulated within the `BeamerUtil` class. This utility class abstracts the Beamer-specific logic and ensures consistency across the application's navigation flows.

```
class HomeController extends AppPageController {
  ...
  void gotoCounter({dynamic params}) {
    CounterRepository counterRepository = repo<CounterRepository>();
    BeamerUtil().go(
      context: context,
      location: counterRepository.navigation == 1 ? BeamerUtil().navigationHomeOneCounterOnePath :
      BeamerUtil().navigationHomeTwoCounterTwoPath,
    );
  }
  ...
}
```

With all the necessary changes in place, we can now launch the application and observe the updated behavior in action. Each counter operates independently, and both can be incremented or decremented through their respective `Counter` pages demonstrating proper state isolation across navigation paths.



9 FIELD AND PAGE VALIDATION

Before processing user input, it must be validated to ensure correctness. In general, validation falls into two categories:

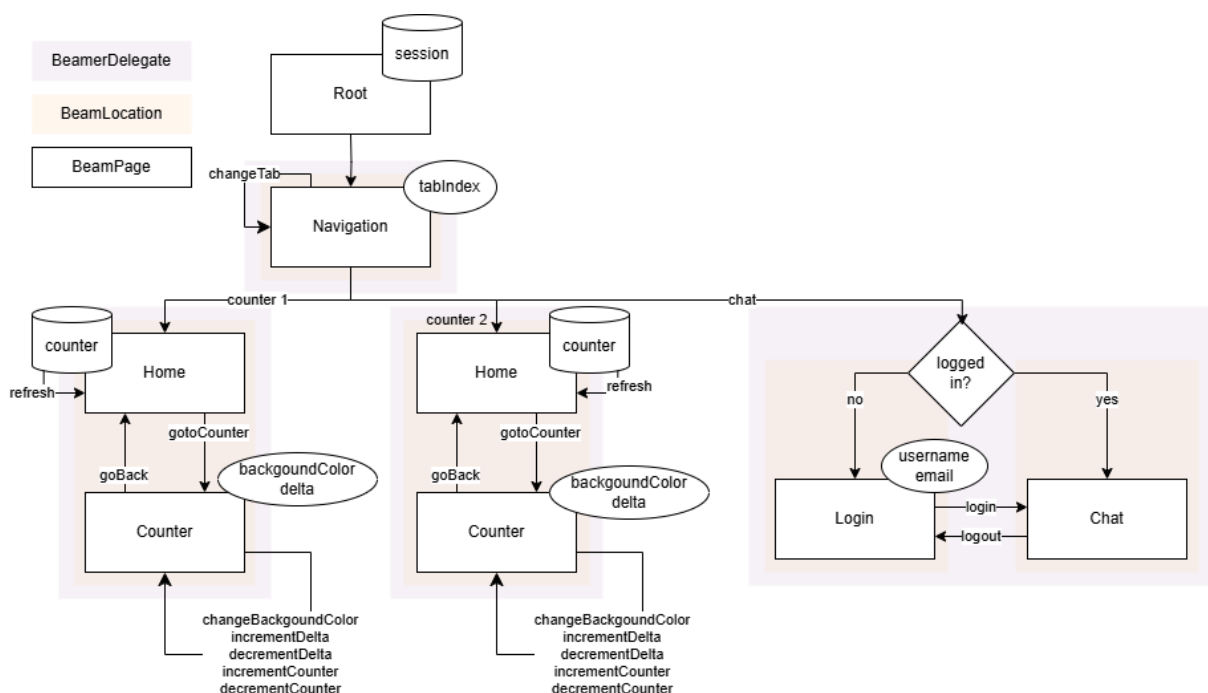
- **Field-level validation.** This checks individual fields for correctness. For example, an email address field may be validated by verifying that it contains an “@” symbol separating the username from the domain name.
- **Form-level (page-level) validation.** When a page contains multiple input fields, the form should only be submitted once all fields have passed validation. Fields may also have dependencies such as a drop-down list whose values are conditionally displayed based on another field’s selection. These interdependent checks are part of form-level validation.

While Flutter provides validation mechanisms through its widget system, these are tightly coupled with the UI components. This coupling can be problematic when applying the Application Layers Framework, which emphasizes a clear separation between views and controllers.

To maintain that separation, the Formz package (<https://pub.dev/packages/formz>) offers a more flexible solution. It allows developers to create custom validators that reside within the page’s state layer enabling UI-independent validation logic. This supports cleaner architecture and reduces the risk of mixing presentation and business logic.

To demonstrate how Formz integrates with ALF, we will introduce a new navigation path featuring a Login page. This addition will showcase how validation flows can be architected while keeping state, control logic, and UI concerns properly decoupled.

The updated navigation diagram will outline this new path in context with the existing application structure.



The `Login` page is intentionally kept straightforward, featuring two input fields: username and email address. Once both fields have been successfully validated, the user is redirected to the `Chat` page, a placeholder for future enhancements covered in the next chapter.

The emphasis of this chapter is on implementing robust validation logic for the `Login` page to ensure reliable and maintainable input handling.

Upon successful login, the user's username and email address will be stored in the session repository to persist user identity across navigation paths. To support this, a new entity named `UserProfileEntity` will be implemented, encapsulating the relevant user attributes.

The class structure for `UserProfileEntity` is outlined below.

```
class UserProfileEntity extends AppEntity {
  String? username;
  String? email;
}
```

The `UserProfileEntity` will be stored in the session repository, making it accessible throughout the entire lifecycle of the application. This ensures consistent availability of user identity and related attributes across all navigation paths and modules.

```
class SessionRepository extends AppRepository {
  ...
  UserProfileEntity userProfile = UserProfileEntity();
  ...
}
```

The `Formz` package supports the creation of custom input validators that can be easily integrated into a page's state model. In the context of our `Login` page, two validators will be defined:

- A username validator to ensure the input meets specific format or length requirements.
- An email validator to verify that the input contains a valid email structure.

Their respective implementations are shown in the code snippet below, providing a foundation for clean, decoupled validation logic that aligns with the architecture of the Application Layers Framework.

```
enum UsernameValidationError { invalid }

final class UsernameValidator extends AppValidator<String, UsernameValidationError> {
  const UsernameValidator.pure([super.value = '']) : super.pure();
  const UsernameValidator.dirty([super.value = '']) : super.dirty();

  static final _userNameRegex = RegExp(r'^([a-zA-Z ]+)$');

  @override
  UsernameValidationError? validator(String? value) {
    return _userNameRegex.hasMatch(value ?? '') ? null : UsernameValidationError.invalid;
  }
}

enum EmailValidationError { invalid }

final class EmailValidator extends AppValidator<String, EmailValidationError> {
  const EmailValidator.pure([super.value = '']) : super.pure();
  const EmailValidator.dirty([super.value = '']) : super.dirty();

  static final _emailRegex = RegExp(r'^[a-zA-Z0-9.!#$%&'*+/?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$');

  @override
  EmailValidationError? validator(String? value) {
```

```

    return _emailRegex.hasMatch(value ?? '') ? null : EmailValidationError.invalid;
  }
}

```

The Application Layers Framework simply encapsulates the parent class structure, providing architectural consistency without altering how validators are implemented using the Formz package. You can continue defining custom validators as documented, with no modifications required for compatibility with ALF.

Since the Formz package is already encapsulated within the Application Layers Framework, it does not need to be explicitly declared in the project's `pubspec.yaml` file. Its functionality is integrated via the framework, streamlining dependency management and reducing manual configuration.

Now that the custom validators are in place, we can integrate them into the `Login` page controller. Instead of directly storing raw field values, the controller maintains validator instances in the page state encapsulating both the input value and its validation status.

Whenever the username or email input is modified, an event is triggered, updating the corresponding validator with the new value. This automatically invokes the validator's internal logic (typically using a regular expression) to assess input correctness.

After validation, the page state is refreshed to reflect the updated validator, which is then exposed to the view for display purposes such as showing validation feedback. Each validator includes a flag indicating whether its input is valid.

When both the username and email validators return a valid state, the `Login` page becomes eligible for submission. At that point, an event is dispatched to navigate to the `Chat` page, and the validated user profile (username and email) are persisted in the session repository for use throughout the application lifecycle.

```

enum LoginEvent { changeEmail, changeUsername, submit }

enum LoginState { username, email }

class LoginController extends AppPageController {
  LoginController() : super(state: {LoginState.username: const UsernameValidator.pure(), LoginState.email: const EmailValidator.pure()}) {
    register(event: LoginEvent.changeEmail, trigger: changeEmail);
    register(event: LoginEvent.changeUsername, trigger: changeUsername);
    register(event: LoginEvent.submit, trigger: submit);
  }

  void changeEmail({dynamic params}) {
    EmailValidator email = EmailValidator.dirty(params);
    update(state: {LoginState.email: email.isValid ? email : EmailValidator.pure(params)});
  }

  void changeUsername({dynamic params}) {
    UsernameValidator username = UsernameValidator.dirty(params);
    update(state: {LoginState.username: username.isValid ? username : UsernameValidator.pure(params)});
  }

  void submit({dynamic params}) {
    session.userProfile.username = state[LoginState.username].value;
    session.userProfile.email = state[LoginState.email].value;
    BeamerUtil().go(context: context, location: BeamerUtil().navigationChatPath);
  }
}

```

With the `Login` page implementation underway, we focus on how the username and email text fields interact with their corresponding validators stored in the page state. Each input field retrieves its value directly from the associated validator, which also provides validation feedback.

If a field's input is invalid, an error message is displayed beneath the field. The submit button remains disabled until all validators in the page state report valid input. This ensures that the entire page is validated consistently before submission is permitted.

```
class LoginPage extends AppPageView {
  LoginPage({super.key}) : super(controller: LoginController());

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: buildAppBar(context: context, title: L10nUtil().translate().page_Login_appBar_title),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            TextField(
              onChanged: (value) => trigger(event: LoginEvent.changeUsername, params: value),
              decoration: InputDecoration(
                hintText: state[LoginState.username].value != "" ? L10nUtil().translate().page_Login_username_hint : null,
                errorText: state[LoginState.username].value != "" && state[LoginState.username].isNotValid ? L10nUtil().translate().page_Login_username_error : null,
                labelText: L10nUtil().translate().page_Login_username_label,
                border: OutlineInputBorder(),
              ),
            ),
            SizedBox(height: 16.0),
            TextField(
              keyboardType: TextInputType.emailAddress,
              onChanged: (value) => trigger(event: LoginEvent.changeEmail, params: value),
              decoration: InputDecoration(
                hintText: state[LoginState.email].value != "" ? L10nUtil().translate().page_Login_email_hint : null,
                errorText: state[LoginState.email].value != "" && state[LoginState.email].isNotValid ? L10nUtil().translate().page_Login_email_error : null,
                labelText: L10nUtil().translate().page_Login_email_label,
                border: OutlineInputBorder(),
              ),
            ),
            SizedBox(height: 16.0),
            ElevatedButton(
              onPressed: isValidState ? () => trigger(event: LoginEvent.submit) : null,
              child: Text(L10nUtil().translate().page_Login_submit_label),
            ),
          ],
        ),
      ),
    );
  }
}
```

As with all page implementations, we avoid hardcoding any text or labels. Instead, these values are externalized to the language file to support internationalization and localization. This update includes new entries specific to the `Login` page and an additional label for the third tab in `NavigationPage`.

After updating the language file, be sure to run the appropriate translation tools to propagate these changes across all target languages ensuring a consistent and localized user experience throughout the application.

```
{
  ...
  "page_Navigation_chat_navbarLabel": "Chat",
  ...
  "page_Login_appBar_title": "Login Page",
  "page_Login_username_label": "Enter your username",
}
```

```

    "page_Login_username_error": "Invalid username",
    "page_Login_email_label": "Enter your email address",
    "page_Login_email_error": "Invalid email address",
    "page_Login_submit_label": "Login"
  }
}

```

The `BeamerUtil` class has been enhanced to support the new Login and Chat navigation paths, as illustrated in the navigation diagram above.

The relevant class updates are provided below, reflecting how these new routes have been integrated into the navigation logic.

```

...
class NavigationLoginLocation extends AppBeamLocation {
  NavigationLoginLocation({required super.routeInformation, required super.pages});
}

class NavigationChatLocation extends AppBeamLocation {
  NavigationChatLocation({required super.routeInformation, required super.pages});
}
...
class BeamerUtil extends AppBeamerUtil {
  ...
  final String navigationLoginPath = "/navigation/login";
  final String navigationChatPath = "/navigation/chat";
  ...
  late final AppBeamerDelegate navigationChatDelegate;
  ...
  BeamerUtil._init() {
    ...
    navigationChatDelegate = AppBeamerDelegate(
      initialPath: navigationLoginPath,
      locations: {
        navigationLoginPath: (routeInformation) => NavigationLoginLocation(
          routeInformation: routeInformation,
          pages: [AppBeamPage(navPath: navigationLoginPath, page: LoginPage())],
        ),
        navigationChatPath: (routeInformation) => NavigationChatLocation(
          routeInformation: routeInformation,
          pages: [AppBeamPage(navPath: navigationChatPath, page: ChatPage())],
        ),
      },
    );
  }
}

```

To display the newly added navigation path within the navigation page, incorporate the following configuration into the `NavigationPage`. This ensures that the new route is properly registered and accessible through the navigation interface.

```

class NavigationPage extends AppPageView {
  ...
  Widget build(BuildContext context) {
    return Scaffold(
      body: IndexedStack(
        index: state[NavigationState.tabIndex],
        children: [
          ...
          AppBeamer(routerDelegate: BeamerUtil().navigationChatDelegate),
        ],
      ),
      bottomNavigationBar: NavigationBar(
        selectedIndex: state[NavigationState.tabIndex],
        destinations: [
          ...

```

```

        NavigationDestination(label: L10nUtil().translate().page_Navigation_chat_navbarLabel, icon: const
Icon(Icons.chat)),
    ],
    ...
}

```

Although the `Chat` page currently serves as a placeholder for future functionality, a minimal implementation is still required to ensure the application compiles and runs smoothly. This allows the navigation flow to remain functional while laying the groundwork for enhancements in the upcoming chapter.

```

class ChatPage extends AppPageView {
  ChatPage({super.key}) : super(controller: ChatController());

  @override
  Widget build(BuildContext context) {
    List<String> messages = ["Hello", "World!"];
    return Scaffold(
      appBar: buildAppBar(context: context, title: "Chat Page"),
      body: Column(
        children: [
          Expanded(
            child: ListView.builder(
              itemCount: messages.length,
              itemBuilder: (context, index) {
                return ListTile(title: Text(messages[index]));
              },
            ),
          ),
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: Row(
              children: [
                Expanded(
                  child: TextField(decoration: InputDecoration(hintText: 'Type your message here...')),
                ),
                IconButton(icon: Icon(Icons.send), onPressed: () => {}),
              ],
            ),
          ),
        ],
      ),
    );
  }
}

```

The corresponding controller implementation is provided below.

```

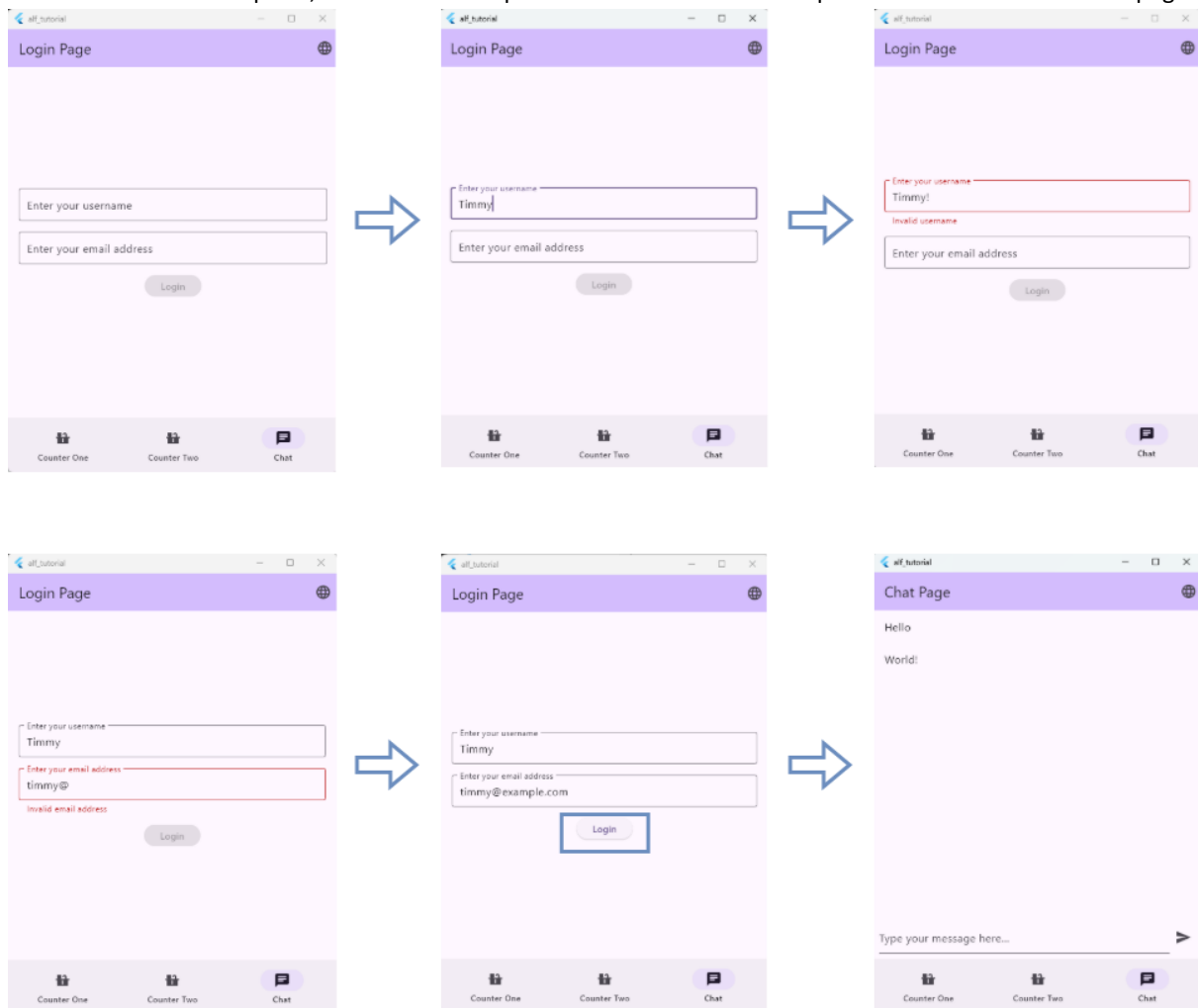
class ChatController extends AppPageController {}

```

Now that all required changes have been applied, we can launch the application to observe the field-level and form-level validation in action.

- When the username input contains only alphabetic characters, it passes validation without error.
- If a special character is entered in the username field, a validation error is triggered and displayed immediately.
- Similarly, the email address must follow a valid format; otherwise, a corresponding validation error will appear.

Once both fields contain valid input, the overall form validation succeeds, and the login button becomes enabled. At this point, users can press the button to proceed to the next page.



10 NOTIFICATION AND MESSAGING

A Flutter application can be designed as a standalone, single-user client with all presentation logic handled locally. In such cases, there is no need for notifications or messaging, since the application is not connected to external data sources. This is the approach we have followed so far in our sample application.

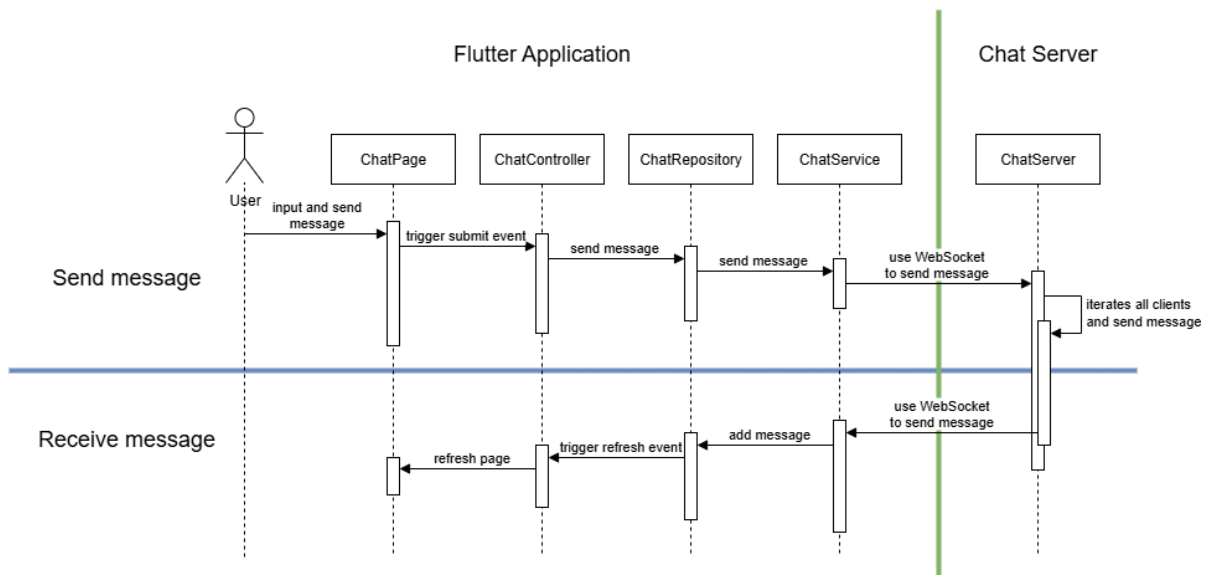
However, networked applications require the ability to respond to real-time updates such as stock market alerts or chat messages received from external systems. In these scenarios, presentation updates are triggered not just by user interactions, but also by backend events. To maintain architectural consistency, such updates follow the same design patterns established by the Application Layers Framework.

In the previous chapter, we introduced a `Chat` page as a placeholder. We will now enhance this page and establish connectivity with a backend server to enable real-time messaging between multiple users. This will be achieved using the `WebSocket` protocol (<https://en.wikipedia.org/wiki/WebSocket>), which supports efficient, bi-directional communication.

When multiple users are connected to the backend server:

- One user can type a message and send it to the server.
- The server then broadcasts this message to all other connected clients.
- Each client's Flutter application receives and immediately displays the message demonstrating real-time synchronization.

To better understand this integration and its alignment with ALF principles, the diagram below presents a high-level application flow and its key components.



Previously, our Flutter application operated as a standalone client with all presentation logic handled locally. We now introduce a chat server that connects to the application via `WebSocket`, transforming it into a network-enabled solution capable of real-time communication.

Once the chat repository is initialized from the chat page, the application establishes a `WebSocket` connection to the backend chat server. When the user enters a message and submits it, an event is triggered on the chat

controller, which invokes the `sendMessage` function from the chat repository. The message is then passed to the server using WebSocket, with all underlying communication logic encapsulated within the chat service.

Upon receiving the message, the chat server iterates through its pool of connected clients and broadcasts the message to each one including the sender. This flow is illustrated in the lower portion of the sequence diagram. The Flutter application receives this message through its WebSocket connection, and the chat service appends it to the chat repository.

The repository, in turn, notifies the chat controller, which subscribes to repository changes. This triggers a UI event to refresh the chat page and display the new message creating a seamless and reactive user experience.

Implementation begins at the far right of the diagram, working back toward the application's frontend layer. The chat server is a minimal Python-based program that:

- Listens on `localhost:8888`
- Accepts and registers WebSocket connections
- Echoes incoming messages to all connected clients, including the sender

```
import asyncio
import websockets

connectedClients = set()

async def handleClient(websocket):
    connectedClients.add(websocket)
    try:
        async for message in websocket:
            sendList = []
            for client in connectedClients:
                print(message)
                sendList.append(asyncio.create_task(client.send(message)))
            await asyncio.wait(sendList)
    finally:
        connectedClients.remove(websocket)

async def main():
    server = await websockets.serve(handleClient, "localhost", 8888)
    await server.wait_closed()

if __name__ == "__main__":
    asyncio.run(main())
```

Before continuing with the Flutter implementation, we need to set up a few prerequisites. Start by updating the `pubspec.yaml` file to include the necessary WebSocket dependency, which enables real-time communication between the client and the backend server.

```
...
dependencies:
  ...
  web_socket_channel: any
...
```

Next, open `lib/common/import.dart` and update it to export the WebSocket package, along with two additional dependencies required for the chat functionality.

```
export 'package:flutter/material.dart';
export 'package:application_layers_framework/application_layers_framework.dart';
export 'dart:convert';
export "package:flutter/scheduler.dart";
export 'package:web_socket_channel/web_socket_channel.dart';
```

```
export '/common/export.dart';
```

The `ChatService` class encapsulates all communication logic with the chat server, abstracting the underlying `WebSocket` interactions to maintain clean separation from the application's presentation and controller layers. It also handles the serialization and deserialization of data, converting backend message payloads to and from the `ChatMessageEntity`, which represents the structured format used for storing and displaying chat messages within the app.

```
class ChatService extends AppService {
  WebSocketChannel? chatServerChannel;

  Future<void> connect({required void Function({required ChatMessageEntity message}) receiveMessage}) async {
    chatServerChannel = WebSocketChannel.connect(Uri.parse(ConfigUtil().chat.url));
    await chatServerChannel?.ready;
    chatServerChannel?.stream.listen((payload) {
      if (payload is String) {
        Map<String, dynamic> data = jsonDecode(payload);
        ChatMessageEntity message = ChatMessageEntity(username: data["username"], text: data["text"]);
        receiveMessage(message: message);
      }
    });
  }

  void disconnect() {
    chatServerChannel?.sink.close();
  }

  void sendMessage({required ChatMessageEntity message}) {
    Map<String, dynamic> payload = {"username": message.username, "text": message.text};
    chatServerChannel?.sink.add(jsonEncode(payload));
  }
}
```

To highlight is the use of `ConfigUtil().chat.url` for the chat server connection URL, which has been externalized as part of the application's configuration. As a best practice, even seemingly trivial parameters should be defined as a centralized application configuration.

```
class ConfigUtil extends AppUtil {
  ...
  final ChatConfig chat = const ChatConfig();
}
...
class ChatConfig {
  const ChatConfig();
  final String url = "ws://localhost:8888";
}
```

Although `ChatMessageEntity` currently includes only two attributes, a chat message may eventually contain additional information such as images, timestamps, reply references, and more. To accommodate these potential future enhancements without requiring refactoring, the message structure has been encapsulated within a dedicated entity. This proactive approach supports scalability and preserves architectural flexibility as new features are introduced.

```
class ChatMessageEntity extends AppEntity {
  String? username;
  String? text;

  ChatMessageEntity({this.username, this.text});
}
```

With `WebSocket` communication abstracted by the `ChatService` provider, the `ChatRepository` remains streamlined and focused. It exposes a minimal set of core methods:

- `sendMessage()` — sends user-generated messages to the backend server.
- `addMessage()` — receives incoming messages from the server and adds them to the repository.
- `connect()` — establishes the `WebSocket` connection and registers a callback listener to handle incoming messages from the server.
- `disconnect()` — unregisters from the server, typically invoked during user logout. It is not used by the sample application and provided for completeness.

This design keeps the repository lightweight while ensuring seamless coordination between client-side messaging and server-side updates.

```
enum ChatChangeType { addMessage }

class ChatRepository extends AppRepository {
  List<ChatMessageEntity> messages = [];
  ChatService chatService = ChatService();

  ChatRepository();

  void addMessage({required ChatMessageEntity message}) {
    messages.add(message);
    notify(change: ChatChangeType.addMessage);
  }

  Future<void> connect() async {
    await chatService.connect(receiveMessage: addMessage);
  }

  void disconnect() {
    chatService.disconnect();
  }

  void sendMessage({required ChatMessageEntity message}) {
    chatService.sendMessage(message: message);
  }
}
```

Moving on to the controller implementation, the controller handles three primary events:

1. **Initialization event** — triggered to register the client with the chat server via `WebSocket`.
2. **Refresh event** — invoked when new messages are received from either the user or the backend, prompting an update of the UI.
3. **Submission event** — triggered when the user sends a message, forwarding the input to the backend server.

The controller subscribes to changes in the `ChatRepository`, allowing real-time messages received from the backend to dispatch the refresh event, ensuring the chat page remains synchronized.

The chat page has two input fields:

- A `TextEditingController` to enable easy control over text state such as resetting or initializing field values.
- A `ScrollController` attached to the message list, which automatically scrolls to the latest message when the page is refreshed.

```
enum ChatEvent { initialize, refresh, submit }
```

```
enum ChatState { textController, messageScrollController }

class ChatController extends AppPageController {
  ChatController()
    : super(
        state: {
          ChatState.textController: TextEditingController(text: ""),
          ChatState.messageScrollController: ScrollController(),
        },
      ) {
    register(event: ChatEvent.initialize, trigger: initialize);
    register(event: ChatEvent.refresh, trigger: refreshView);
    register(event: ChatEvent.submit, trigger: submit);
  }

  void initialize({dynamic params}) {
    repo<ChatRepository>().connect();
  }

  void submit({dynamic params}) {
    ChatMessageEntity message = ChatMessageEntity(username: session.userProfile.username, text:
state[ChatState.textController].value.text);
    repo<ChatRepository>().sendMessage(message: message);
    update(state: {ChatState.textController: TextEditingController(text: "")});
  }

  @override
  void init() {
    super.init();
    subscribeRepository<ChatRepository>(listenChange: ChatChangeType.addMessage, event: ChatEvent.refresh);
  }
}
```

While the implementation may appear lengthy or complex at first glance, the chat page logic is conceptually straightforward and follows three key steps:

1. **Initialization:** On page creation, an event is triggered to register the client with the chat server via WebSocket.
2. **Repository Setup:** The `ChatRepository` is initialized to store incoming messages and facilitate communication with the backend server.
3. **UI Construction:** The page is built with two main sections:
 - A message pane displaying all received chat messages.
 - An input pane for composing and submitting new messages.

This structure ensures clarity, maintainability, and alignment with the Application Layers Framework. The complete implementation is shown below.

```
class ChatPage extends AppPageView {
  final FocusNode inputFocusNode = FocusNode();

  ChatPage({super.key})
    : super(
        controller: ChatController(),
        event: ChatEvent.initialize,
        repositories: [AppRepositoryProvider<ChatRepository>(create: (context) => ChatRepository())],
      );

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: buildAppBar(context: context, title: l10nUtil().translate().page_chat_appBar_title),
      body: _buildBody(context),
      backgroundColor: Colors.lightBlue,
    );
  }
}
```

```

Widget _buildBody(BuildContext context) {
  return Column(
    children: [
      Expanded(child: _buildMessagePane(context)),
      Align(alignment: Alignment.bottomLeft, child: _buildInputPane(context)),
    ],
  );
}

Widget _buildInputPane(BuildContext context) {
  return Container(
    margin: const EdgeInsets.only(bottom: 15, left: 15, right: 15, top: 15),
    child: Container(
      decoration: BoxDecoration(color: Colors.blue.shade100, borderRadius: BorderRadius.circular(25.0)),
      child: Column(
        children: [
          Row(
            mainAxisAlignment: MainAxisAlignment.end,
            mainAxisAlignment: MainAxisAlignment.max,
            children: [
              Flexible(
                child: Container(
                  padding: const EdgeInsets.only(left: 20),
                  child: TextField(
                    controller: state[ChatState.textController],
                    onChanged: (value) => trigger(event: ChatEvent.refresh),
                    onSubmitted: (value) {
                      if (state[ChatState.textController].value.text != "") {
                        trigger(event: ChatEvent.submit);
                      }
                      inputFocusNode.requestFocus();
                    },
                    focusNode: inputFocusNode,
                    decoration: InputDecoration(
                      hintText: L10nUtil().translate().page_Chat_message_label,
                      hintStyle: TextStyle(color: Colors.black),
                      border: InputBorder.none,
                    ),
                  ),
                ),
              ),
            ],
          ),
          IconButton(
            icon: Icon(Icons.send, color: state[ChatState.textController].value.text != "" ? Colors.green :
Colors.grey),
            onPressed: state[ChatState.textController].value.text != ""
              ? () {
                  trigger(event: ChatEvent.submit);
                  inputFocusNode.requestFocus();
                }
              : null,
          ),
        ],
      ),
    ),
  );
}

Widget _buildMessagePane(BuildContext context) {
  ChatRepository chatRepo = repo<ChatRepository>();
  String username = session.userProfile.username!;
  ScrollController scrollController = state[ChatState.messageScrollController];
  SchedulerBinding.instance.addPostFrameCallback((_) {
    scrollController.jumpTo(scrollController.position.maxScrollExtent);
  });
  return ListView.builder(
    physics: const AlwaysScrollableScrollPhysics(),
    controller: scrollController,
    itemCount: chatRepo.messages.length,
    padding: const EdgeInsets.only(top: 10, bottom: 10),
    itemBuilder: (context, index) {
      ChatMessageEntity message = chatRepo.messages[index];

```

```

        return _buildMessage(message, message.username == username);
    },
);
}

Widget _buildMessage(ChatMessageEntity message, bool own) {
    return Container(
        padding: const EdgeInsets.only(left: 14, right: 14, top: 10, bottom: 10),
        child: Align(
            alignment: own ? Alignment.topLeft : Alignment.topRight,
            child: Container(
                decoration: BoxDecoration(
                    borderRadius: BorderRadius.only(
                        topLeft: Radius.circular(own ? 0 : 20),
                        topRight: const Radius.circular(20),
                        bottomLeft: const Radius.circular(20),
                        bottomRight: Radius.circular(own ? 20 : 0),
                    ),
                    color: own ? Colors.grey.shade200 : Colors.blue[200],
                ),
                padding: const EdgeInsets.all(16),
                child: Column(
                    crossAxisAlignment: CrossAxisAlignment.start,
                    children: [
                        Text(message.username!, style: const TextStyle(fontSize: 10, fontWeight: FontWeight.bold)),
                        Text(message.text!, style: const TextStyle(fontSize: 15)),
                    ],
                ),
            ),
        ),
    );
}
}

```

As with previous implementations, newly added texts and labels have been externalized to the language file to support localization. Refer to chapter 3 (Internationalization) for instructions on translating these entries into the target languages using the appropriate translation tools.

```

{
    ...
    "page_chat_appBar_title": "Chat Page",
    "page_chat_message_label": "Write message ..."
}

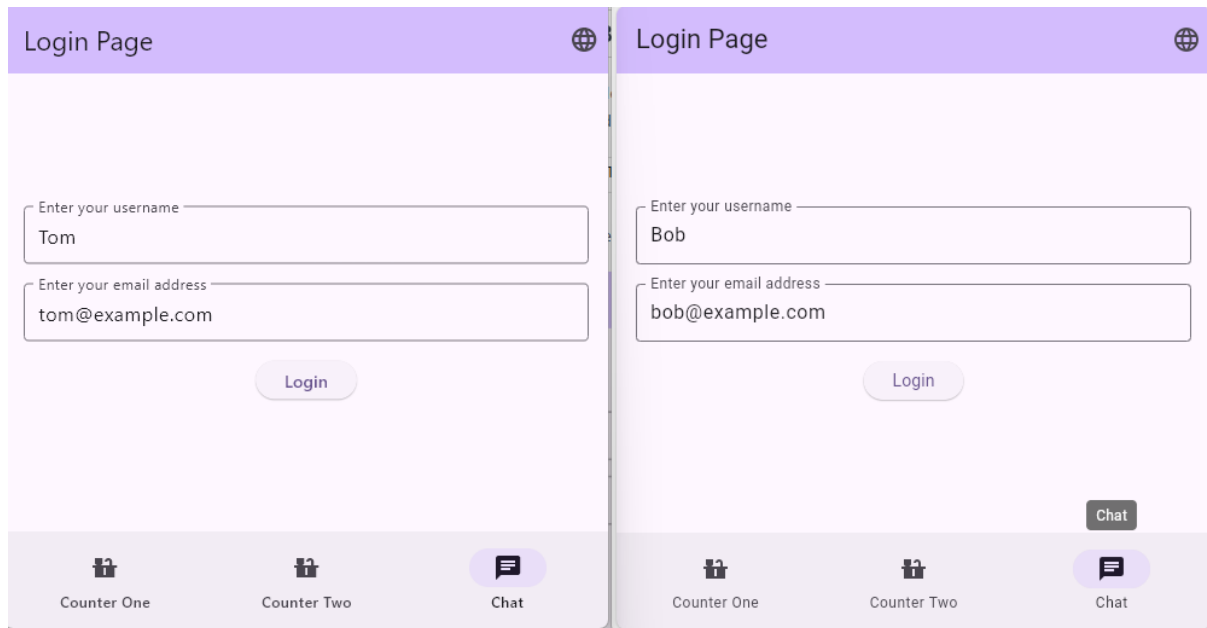
```

With the full implementation now complete, the next step is to launch the chat server from a new command-line interface (CLI) prompt. Once started, the server will remain active and continue listening for connections.

To terminate the server session on Windows, simply press `Ctrl + C` in the terminal.

```
$ python assets/tools/chat_server.py
```

Next, we will launch two instances of the sample application and log in using different usernames. This setup allows us to simulate multiple user sessions and observe real-time messaging behavior between clients through the chat interface.

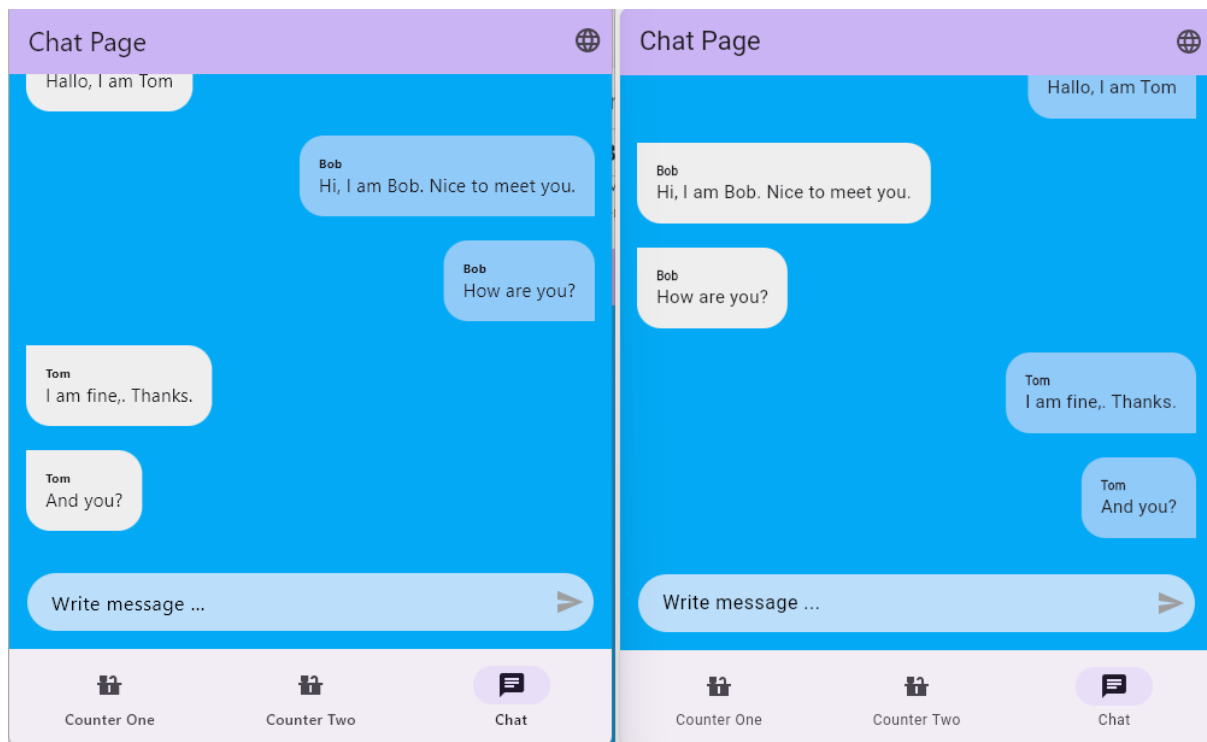


In a Flutter project, you cannot launch multiple instances of the application simultaneously on the same target platform. To run a second instance, open a new CLI prompt and launch the application using a different target platform, such as:

```
$ flutter run
Connected devices:
Windows (desktop) • windows • windows-x64 • Microsoft Windows [Version 10.0.26100.3775]
Chrome (web) • chrome • web-javascript • Google Chrome 135.0.7049.42
Edge (web) • edge • web-javascript • Microsoft Edge 135.0.3179.85
[1]: Windows (windows)
[2]: Chrome (chrome)
[3]: Edge (edge)
Please choose one (or "q" to quit): 3
...
```

In the chat interface, each user sees their own messages aligned to the left side of the screen, while messages from other participants appear on the right side. For example, Tom's messages are left-aligned on his screen, and those same messages appear right-aligned on Bob's screen. Likewise, Bob's messages are shown on the left for him and on the right for Tom.

This mirrored layout helps distinguish between sent and received messages, enhancing readability and user experience. Left for self, right for others, simple and intuitive.



On the server side, every message sent to connected clients is logged to the console along with the username of the sender for traceability. Because each message is broadcast to all active clients, including the sender, the same message appears twice in the console: once for each recipient.

```
$ python assets/tools/chat_server.py
{"username":"Tom","text":"Hallo, I am Tom"}
{"username":"Tom","text":"Hallo, I am Tom"}
{"username":"Bob","text":"Hi, I am Bob. Nice to meet you."}
{"username":"Bob","text":"Hi, I am Bob. Nice to meet you."}
{"username":"Bob","text":"How are you?"}
{"username":"Bob","text":"How are you?"}
{"username":"Tom","text":"I am fine,. Thanks."}
{"username":"Tom","text":"I am fine,. Thanks."}
{"username":"Tom","text":"And you?"}
{"username":"Tom","text":"And you?"}
```

Just like in real-time text messaging, typos are bound to happen. Unfortunately, I noticed the mistake only after capturing the screenshots. See if you can spot it!

This implementation demonstrates how the Application Layers Framework effectively separates controller and presentation logic from backend communication. This modular design allows the application to handle notifications and messaging from external systems independently of the user interface supporting clean and scalable architecture.

One area not addressed in this sample application is the handling of error cases, such as network unavailability. While this falls outside the current scope, implementing robust exception handling is strongly recommended to ensure a smoother and more resilient user experience.

11 FINAL WORDS

Every software development project introduces fresh experiences and sparks new ideas often uncovering opportunities for thoughtful refactoring. To avoid unnecessary rework, this guide has now reached a stage where you are equipped to start building a frontend application on top of a solid framework and architecture. This foundation allows you to focus on delivering user-driven features without being overwhelmed by non-functional concerns.

To help you build effectively around the Application Layers Framework, consider the following suggestions:

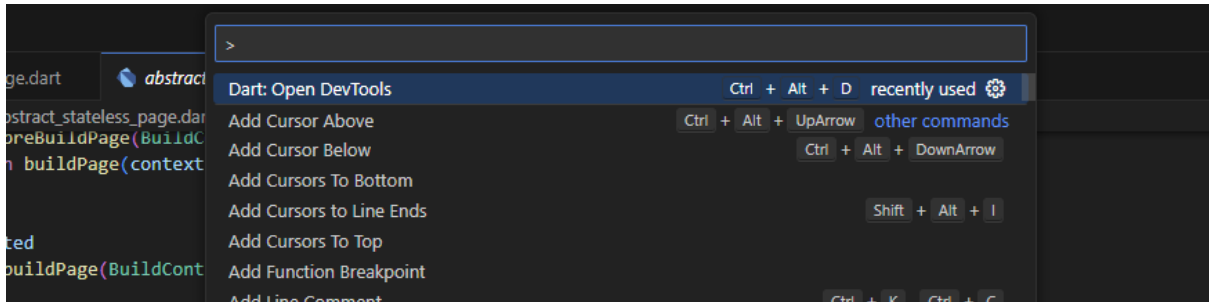
- Extract shared application classes and repositories into independent packages for modular reuse.
- Split large translation files into smaller language modules to support better collaboration and maintainability.
- Create helper tools for generating boilerplate code (e.g., views, controllers, repositories, services).
- Integrate utilities like translation handlers and code generators directly into your development environment.
- Identify and adopt best practices for unit testing, integration testing, and runtime debugging.
- Standardize project documentation aligned to the chosen architecture and framework.
- Use UI prototyping tools (e.g., Figma) to visualize flows and structure interfaces before diving into implementation.

These recommendations aim to enhance scalability, reduce friction during development, and support clean engineering practices.

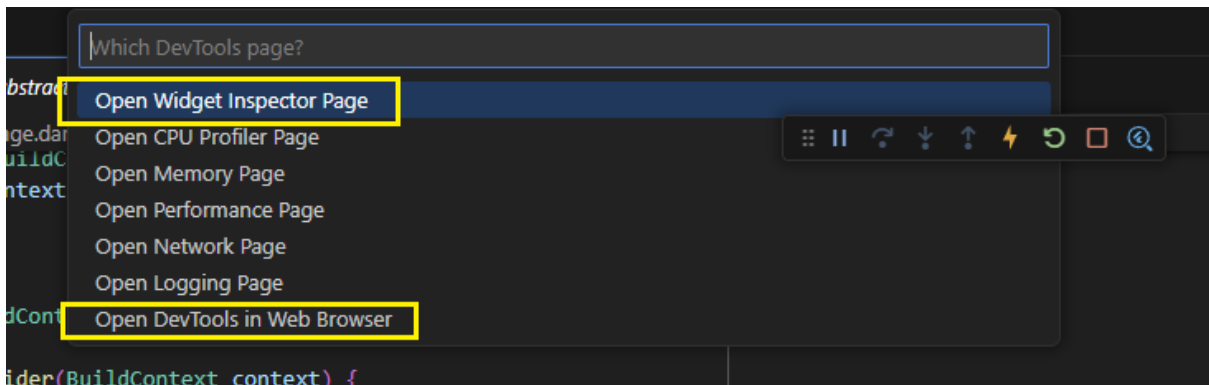
APPENDIX – HELP AND FAQ

HOW TO OPEN THE WIDGET EXPLORER

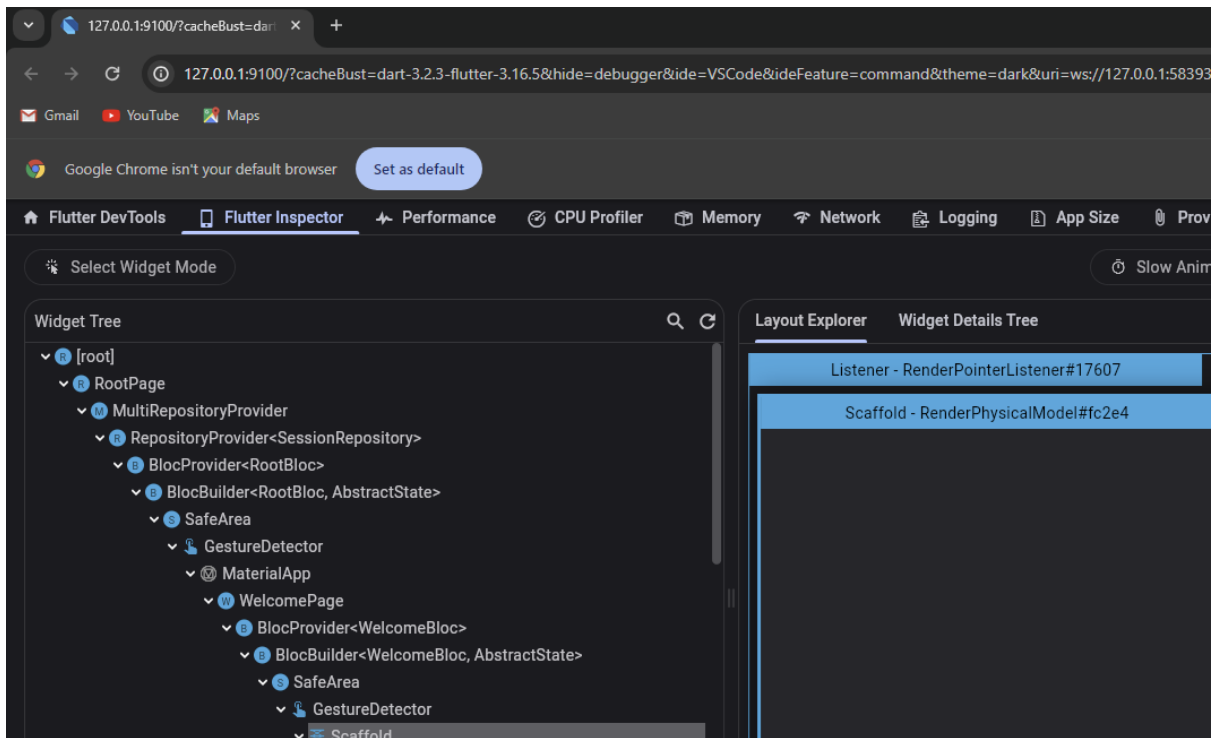
Once the application has been successfully launched in Visual Studio Code, press `F1` to open the command palette, then select `Dart: Open DevTools`. This will launch Dart's powerful suite of development tools for debugging, profiling, and inspecting your Flutter application in real time.



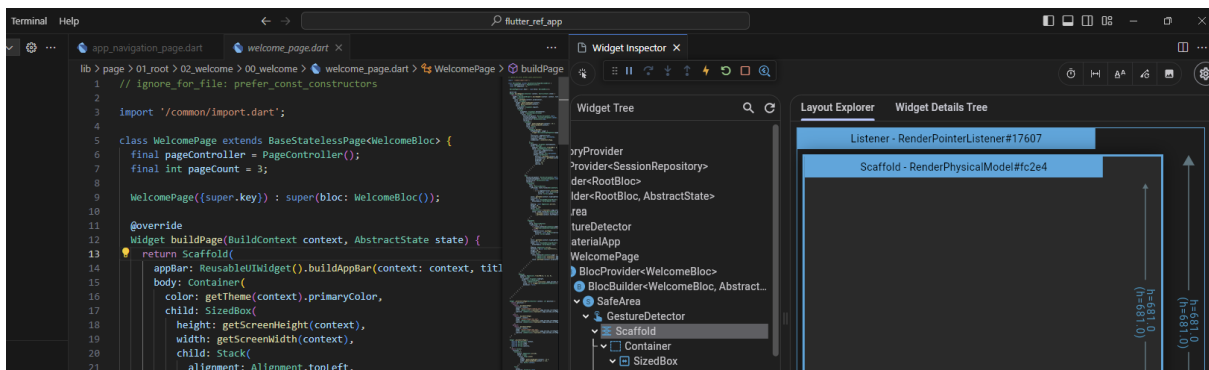
You can access the Widget Inspector either inside Visual Studio Code or inside a web browser. Both options provide full access to Flutter's visual tree, making it easier to troubleshoot layout issues and understand widget behavior in real time.



The web browser view offers a larger canvas, providing more space to visualize and interact with the widget tree in Dart DevTools. This expanded layout makes it easier to navigate complex UI structures and debug layout behavior more effectively.



When opened inside Visual Studio Code, Dart DevTools appears alongside the source code browser, consolidating both your development tools and runtime diagnostics into a unified view. While space may feel a bit more compact, having everything in one place enables quick navigation between code and debugging insights making for an efficient workflow.

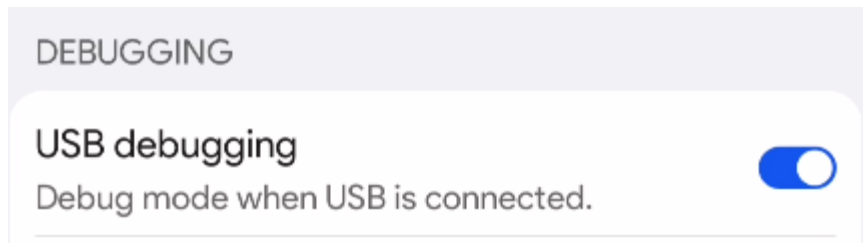


HOW TO TEST WITH ANDROID DEVICE

To proceed, make sure **USB debugging** is enabled on your Android device. You can activate this option by navigating to:

System Settings → Developer Options → USB Debugging

This allows your device to communicate with Visual Studio Code for building and testing applications.

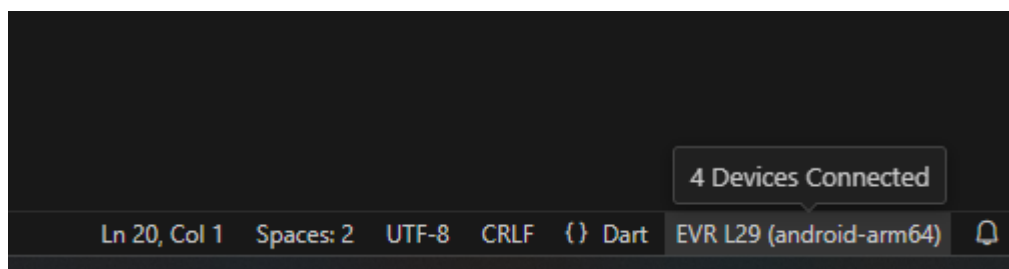


Since **Developer Options** are hidden by default on most Android devices, enabling them can vary slightly depending on the manufacturer and model. To activate this feature:

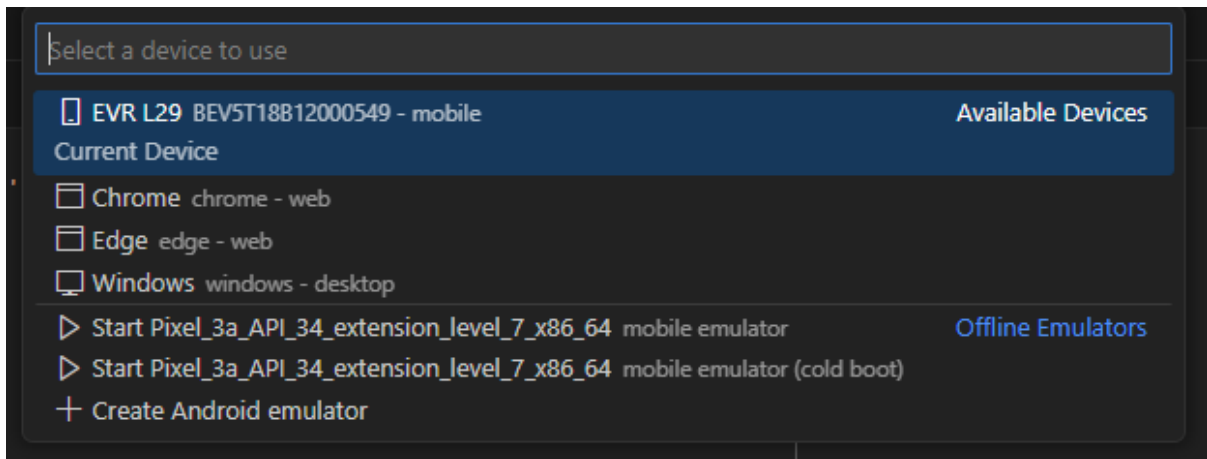
- Search online using keywords such as your **Android device model** along with “**enable Developer Options**” to find detailed steps specific to your device.
- Typically, this involves tapping the **Build Number** several times in the system's **About phone** section.

Once Developer Options are enabled:

1. **Connect your Android device** to your computer using a **USB data cable**.
2. When prompted, make sure to **allow data and file transfer** between your device and computer.
3. Open your **Flutter project** in **Visual Studio Code**.
4. In the **bottom right corner**, locate the **device selector** and confirm that your Android device appears as a connected target.



Click on the **device selector** in the bottom right corner of Visual Studio Code. A menu will appear at the top of the editor, allowing you to **choose the target device** on which to launch your Flutter application.



Select your **Android device** from the device selector. Once chosen, every time you launch your Flutter application from **Visual Studio Code**, it will automatically run on the selected device fully integrated with VS Code's built-in **debugging tools**, such as breakpoints, hot reload, and performance profiling.

HOW TO MIRROR ANDROID DEVICE SCREEN

Now that your Android device is set up for testing Flutter applications, you might also want to **mirror the device screen directly onto your computer monitor**. For this, you can use the open-source tool **scrcpy**: <https://github.com/Genymobile/scrcpy>. It allows full interaction with your Android device from your computer, similar to using an Android emulator.

To get started on **Windows**:

1. Download scrcpy by following the instructions provided at the link above.
2. Extract the downloaded files to a desired folder.
3. Open a **command prompt** in that folder.

To list all connected Android devices, use the following command:

```
C:\Programs\scrcpy>adb devices
List of devices attached
BEV5T18B12000549      device
```

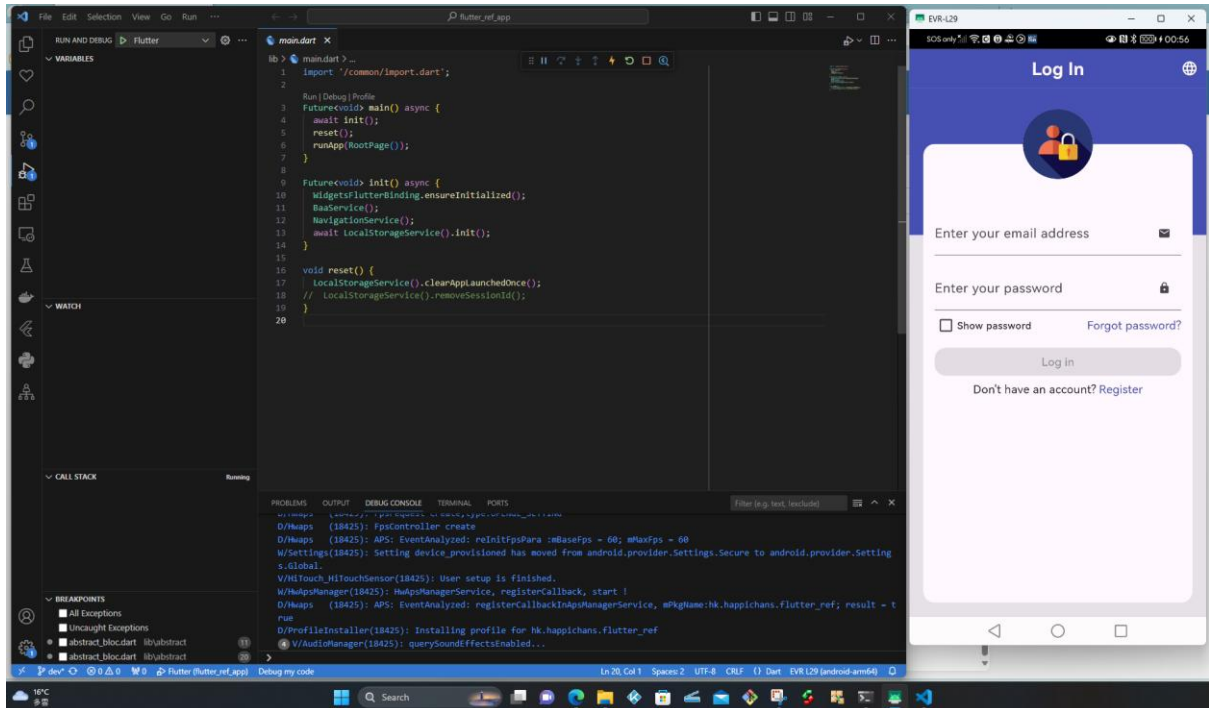
```
C:\Programs\scrcpy>
```

This will show the **device identifiers** for every Android device currently connected to your computer via USB.

Once an **Android device is connected**, you can mirror its screen to your computer by simply running the **scrcpy executable**. This launches the device interface on your desktop, enabling seamless interaction as if using an emulator.

```
C:\Programs\scrcpy>scrcpy-console.bat
scrcpy 2.1.1 <https://github.com/Genymobile/scrcpy>
INFO: ADB device found:
INFO:      -->  (usb)      BEV5T18B12000549      device  EVR_L29
C:\Programs\scrcpy\scrcpy-server: 1 file pushed, 0 skipped. 121.4 MB/s (56995 bytes in 0.000s)
[server] INFO: Device: [HUAWEI] HUAWEI EVR-L29 (Android 10)
[server] WARN: Audio disabled: it is not supported before Android 11
INFO: Renderer: direct3d
WARN: Demuxer 'audio': stream explicitly disabled by the device
INFO: Texture: 1080x2240
```

Now that the screen is mirrored to your computer, the **Flutter application** will appear on the mirrored display just as it would in an **Android emulator**. This setup allows you to interact with the physical device directly from your desktop, providing a seamless development and testing experience.



Since this setup **only mirrors the screen** from a physical Android device, it consumes significantly **less memory** than running a full-fledged Android simulator. This makes it an efficient option for development and testing.

Additionally, the **Flutter application** can be run concurrently with the **Visual Studio Code editor**, enabling you to leverage VS Code's integrated debugging tools—such as breakpoints, hot reload, and log inspection—while interacting with the app through the mirrored device interface.

HOW TO IMPLEMENT A SINGLETON IN DART

Singletons in Dart can be implemented using the **factory constructor pattern**, which ensures that only one instance of the class exists throughout the application lifecycle. Here's a clean and concise example:

```
class Singleton {  
  static final Singleton _instance = Singleton._init();  
  
  factory Singleton() {  
    return _instance;  
  }  
  
  Singleton._init() {  
    ...  
    <initialize the singleton instance>  
    ...  
  }  
}
```

Whenever the `Singleton()` constructor is called, it **returns the same instance** of the class ensuring that only one object is created and shared throughout the application lifecycle.

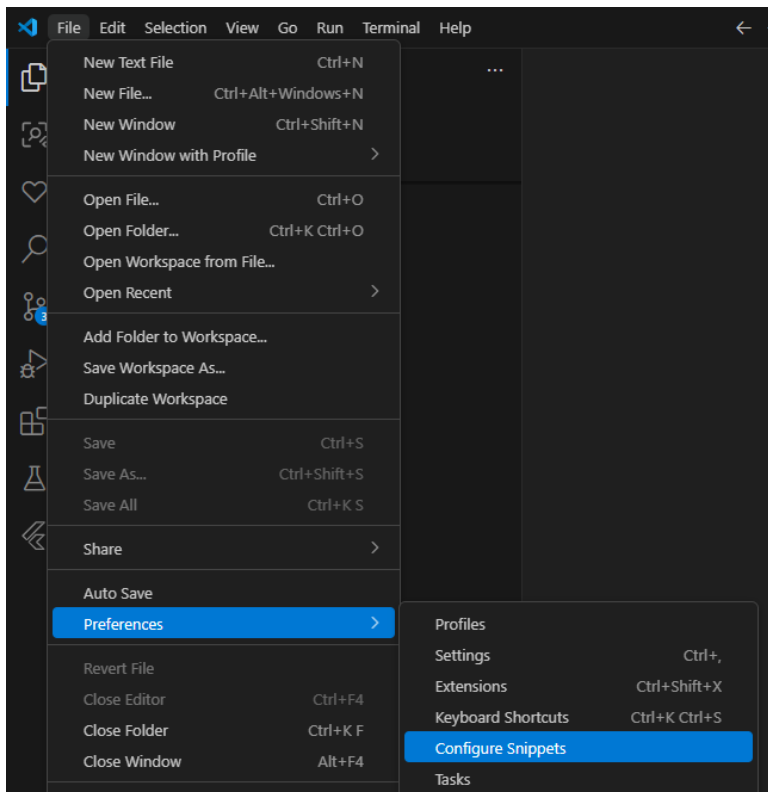
HOW TO USE CODE TEMPLATE

Although any text editor can be used for writing source code, **Visual Studio Code (VS Code)** has become the **industry-standard** development environment due to its rich feature set and extensibility.

One particularly useful feature is its support for **custom code snippets**, which help automate boilerplate and repetitive patterns. This is especially beneficial when developing **Flutter applications using the Application Layers Framework**, where recurring code segments such as **import statements**, **view definitions**, and **controller classes** can be scaffolded with ease.

To define custom Dart snippets in VS Code:

1. Navigate to **File → Preferences → Configure User Snippets**.
2. Select **Dart** from the list of available languages.
3. Define your templates in the generated JSON file by specifying:
 - **prefix**: the shortcut keyword to trigger the snippet.
 - **body**: the actual lines of code.
 - **description**: an optional label to describe the snippet.



This action will create a new file named `dart.json` within your **VS Code user profile**, where you can define and manage custom Dart snippets.

You can use the example templates below as a **starting point** and **tailor them** to suit your project's specific needs. Feel free to adjust prefixes, formatting, and structure to align with your development workflow and coding conventions.

```
{
  "ALF Import": {
    "prefix": "import",
    "body": [
      "import '/common/import.dart';",
    ]
  }
}
```

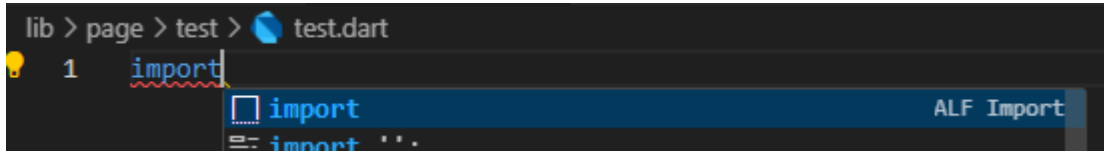
```

    ""
  ]
},
"ALF Page View": {
  "prefix": "page",
  "body": [
    "class ${1:PageName}Page extends AppPageView {",
    "  ${1:PageName}Page({super.key})",
    "    : super(",
    "      controller: ${1:PageName}Controller(),",
    "      event: ${1:PageName}Event.initialize,",
    "      repositories: [",
    "        AppRepositoryProvider<${1:PageName}Repository>(create: (context) =>",
    "${1:PageName}Repository()),",
    "      ],",
    "    );",
    "",
    "    @override",
    "    void init() {",
    "      super.init();",
    "      // your code goes here",
    "    }",
    "",
    "    @override",
    "    Widget build(BuildContext context) {",
    "      return Container();",
    "    }",
    "",
    "    @override",
    "    void dispose() {",
    "      // your code goes here",
    "      super.dispose();",
    "    }",
    "  }"
  ]
},
"ALF Page Controller": {
  "prefix": "controller",
  "body": [
    "enum ${1:PageName}Event { initialize, refreshView }",
    "",
    "enum ${1:PageName}State { field }",
    "",
    "class ${1:PageName}Controller extends AppPageController {",
    "  ${1:PageName}Controller()",
    "    : super(state: {",
    "      ${1:PageName}State.field: 0,",
    "    }) {",
    "    register(event: ${1:PageName}Event.initialize, trigger: initialize);",
    "    register(event: ${1:PageName}Event.refreshView, trigger: refreshView);",
    "  }",
    "",
    "    void initialize({dynamic params}) {",
    "      update(state: { ${1:PageName}State.field: params });",
    "    }",
    "",
    "    @override",
    "    void init() {",
    "      super.init();",
    "      // your code goes here",
    "      subscribeRepository<${1:PageName}Repository>(event: ${1:PageName}Event.refreshView);",
    "    }",
    "",
    "    @override",
    "    void dispose() {",
    "      // your code goes here",
    "      super.dispose();",
    "    }",
    "  }"
  ]
}
}

```

After saving the `dart.json` file, you can **test your custom templates** by creating a new **Dart file**. Begin typing `import` and Visual Studio Code will automatically display **snippet suggestions**, including your newly added template labeled as **“ALF Import”**.

This enables quick insertion of commonly used patterns, helping you maintain consistency and speed throughout your development workflow.



Select the desired snippet, and the corresponding **import statement** will be automatically inserted into your Dart file.



Repeat the same process for the **controller snippet**, and you will see the following generated code. Simply **enter the actual page name**, and placeholder values will be automatically replaced allowing for quick and consistent setup of new pages.

```

lib > page > test > test.dart > PageNameEvent
1  import '/common/import.dart';
2
3  enum PageNameEvent { initialize, refreshView }
4
5  enum PageNameState { field }
6
7  class PageNameController extends AppPageController {
8      PageNameController()
9          : super(state: {
10              PageNameState.field: 0,
11          }) {
12          register(event: PageNameEvent.initialize, trigger: initialize);
13          register(event: PageNameEvent.refreshView, trigger: refreshView);
14      }
15
16      void initialize({dynamic params}) {
17          update(state: { PageNameState.field: params });
18      }
19
20      @override
21      void init() {
22          super.init();
23          // your code goes here
24          subscribeRepository<PageNameRepository>(event: PageNameEvent.refreshView);
25      }
26
27      @override
28      void dispose() {
29          // your code goes here
30          super.dispose();
31      }
32  }

```

Test the final snippet **page** and observe the generated code structure. Once inserted, simply **enter the desired page name**, and all corresponding placeholder references will be automatically replaced throughout the template for consistency and ease of setup.

```
lib > page > test > test.dart > PageNamePage
34 class PageNamePage extends AppPageView {
35   PageNamePage({super.key})
36   : super(
37     controller: PageNameController(),
38     event: PageNameEvent.initialize,
39     repositories: [
40       AppRepositoryProvider<PageNameRepository>(create: (context) => PageNameRepository()),
41     ],
42   );
43
44   @override
45   void init() {
46     super.init();
47     // your code goes here
48   }
49
50   @override
51   Widget build(BuildContext context) {
52     return Container();
53   }
54
55   @override
56   void dispose() {
57     // your code goes here
58     super.dispose();
59   }
60 }
```

For more comprehensive documentation on the **Visual Studio Code snippet tool**, refer to the official guide: Snippets in Visual Studio Code (<https://code.visualstudio.com/docs/editing/userdefinedsnippets>). It covers everything from creating basic snippets to advanced customization techniques.