# Clean Framework

**Clean architecture library inspired by Bob Martin's guidelines.**

- Layered architecture
- Way to structure our apps
- Each layer is testable
- Each layer has one simple job
- The framework is evolving and will get updated as needed when needed

# Architecture Layers

UI
Presenter
ViewModel
Input
Entity
UseCase
Output
Request
External Interface
Gateway
Response
Service/firebase/graphql

# Additional Resources

FeatureProvider

FeatureMapper

Feature

FeatureState

*FeatureTester*

ProvidersContext

AppProvidersContainer

RestGateWay

RestExternalInterface

*Connectivity*

*Logger*

FirebaseGateway

FirebaseService

CFRouterScope

GraphQLGateway

GraphQLService

CFRouteInformationParser

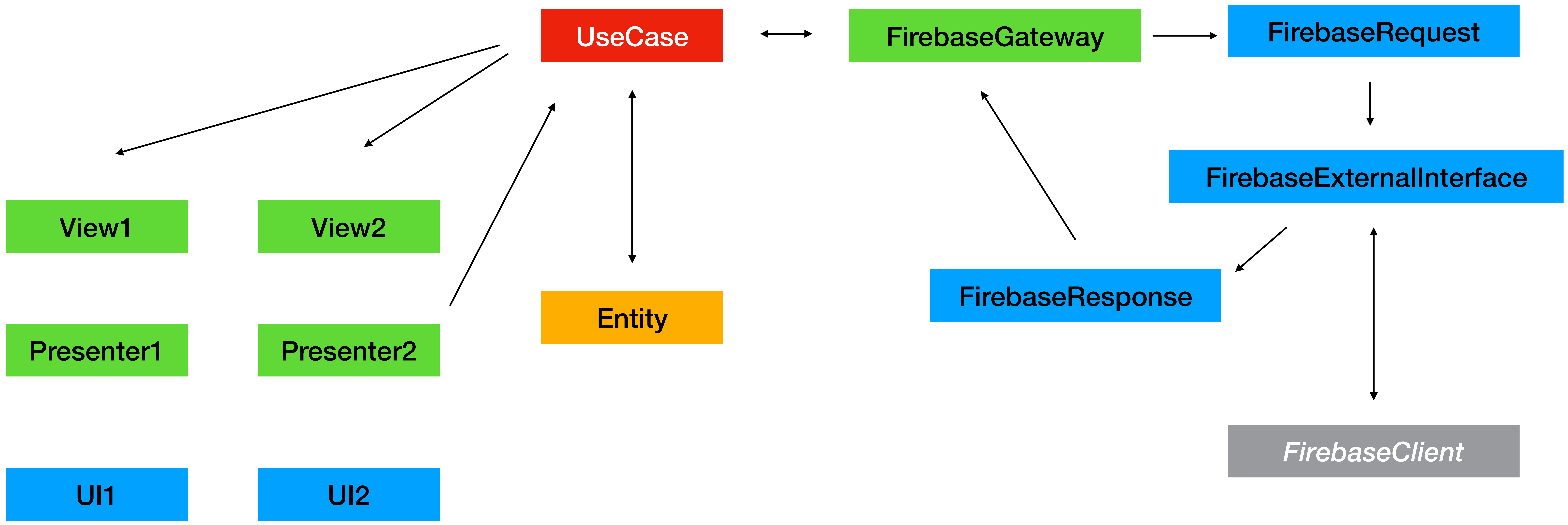
ThemeProvider

RedisGateway

LocalRedis

CFRouterDelegate

## Example of Implementation



## External Interface

The external interface provides a medium between a feature use case and an external dependency such as firebase. It receives a request from a gateway. Then calls the external dependency using the data in the Gateway request. Using Either<Left, Right> it returns either a Right if it was successful or a Left if it was unsuccessful. The goal is to wrap/hide any data that comes from the outside of the app to protect the app from changes. For example a plugin or service could change and only the external interface needs updated. The external interface might be used by more than one use case. One use case could use multiple external interfaces.

## Gateway

The gateway is an intermediary between the external interface and the use case.  It builds the request for the external interface. It reacts to success vs failure. It support synchronous and asynchronous call. Synchronous is supported by DirectGateway using onSuccess. Asynchronous is supported by the WatcherGateway with onYield. A use case may use many gateways. The gateway shields the use case from changes in the external systems. The external system can change but the input/output will typically stay the same.

## Use case

### The domain layer of the app

The use case contains 100% of the business logic of the feature. If the use case tests pass all of the requirements are met. They are independent of the UI of the app, they could run as a flutter app or even a console app. Entities only interact with UseCases.

The flow of data between the use case and components from the outer layers is controlled via Inputs and Outputs.

All data flowing from an external dependency comes through an Input, and all data leaving the use case
to an external dependency goes through an output. This works in the same way for data used on the UI, where the Output is processed by Presenters.

## Entity

Any data that needs to be preserved to enable all the business logic flows, should exist within the Entity class. Any state data should be stored in the Entity and modified by the UseCase. Entities can also be enforcing Domain rules.

A feature that has a form for user data would send the data through UI events to the UseCase, which in turn sets the data into the Entity fields. The UseCase can also be doing this when querying data from any External Interface.

## Presenter

Translates the data of any Output from an UseCase and creates ViewModels, which are "snapshots" of the Entity that can be used by any UI component that wants to show the business logic data.

Presenters are Stateless Widgets with a build method that already shares the generated ViewModel. Everytime the Entity changes and could produce a different Output, the build will be invoked with such changes.

This component serves as an intermediary between user actions and the UseCase events. The UI uses callbacks that the Presenter links with methods from the UseCase.

Each feature has multiple Presenters, one Presenter works with one specific Output. There could exist multiple ViewModels per Output as well.

## UI

This component takes a ViewModel and combines the data with Flutter UI code to create the screens or widgets where the User can see the feature data and interact with it.

A feature uses UI components as an entry point, being the first components to be built after a navigation event happens. They should be building the Presenters and receiving the updates on ViewModels, as well as defining the callbacks that would be actions on the Presenter (and UseCase by consequence).

Since ViewModels are just PODOs, these are useful for testing, any developer can mock ViewModels and complete UI implementations without the need to code UseCases.