POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Development of a Large-Scale Flutter App

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Alejandro Ferrero**

Student ID: 10731020
Advisor: Prof. Luciano Baresi
Co-advisors:
Academic Year: 2021-22

Abstract

The sustained growth of the Flutter framework since its first stable release has drawn the attention of companies from all sectors and sizes, enabling them to build cross-platform applications from a single codebase. This thesis aims at providing tangible insights derived from the development of large-scale Flutter applications. In particular, its purpose is to analyze essential architectural choices, patterns, and opinionated best practices affecting its maintainability, testability, and scalability.

This thesis work introduces a research work about the relevant software techniques, concepts, background, and decisions applied within the scope of the empirical part of the thesis. Said research includes, but is not limited to, knowledge of Cross-platform App Development, Application State Management, Software Architectural Patterns, Software Modularization, and Software Testing.

Furthermore, this thesis work leverages a large-scale Flutter project developed in a distributed software environment for a world-leading company in the domotics and home automation industry. Thus, readers will find detailed descriptions of the choices taken throughout its development, discussions of production-ready code samples, contributions to the existing Flutter literature from an empirical perspective, and carefully argued solutions to non-trivial problems.

Ultimately, this document presents a thorough study and conclusions focused on a specific project. However, its contents are sufficiently general and valuable for other developers to incorporate its applicability into their own by extrapolating the gathered information and adjusting it to their particular requirements.

Keywords: App, Cross-Platform, Dart, DependencyInjection, Flutter, Maintainability, Mock, Modularization, Scalability, Software Architecture, State Management, Testability, Testing

Abstract in lingua italiana

La crescita sostenuta del framework Flutter dalla sua prima versione stabile ha attirato l'attenzione di aziende di tutti i settori e dimensioni, consentendo loro di creare applicazioni multipiattaforma da un'unica base di codice. Questa tesi mira a fornire approfondimenti tangibili derivati dallo sviluppo di applicazioni Flutter su larga scala. In particolare, il suo scopo è analizzare le scelte architettoniche essenziali, i modelli e le migliori pratiche supponenti che ne influenzano la manutenibilità, la testabilità e la scalabilità.

Questo lavoro di tesi introduce un lavoro di ricerca sulle tecniche software rilevanti, i concetti, il background e le decisioni applicate nell'ambito della parte empirica della tesi. Tale ricerca include, a titolo esemplificativo, la conoscenza dello sviluppo di app multipiattaforma, della gestione dello stato delle applicazioni, dei modelli architettonici del software, della modularizzazione del software e del test del software.

Inoltre, questo lavoro di tesi sfrutta un progetto Flutter su larga scala sviluppato in un ambiente software distribuito per un'azienda leader a livello mondiale nel settore della domotica. Pertanto, i lettori troveranno descrizioni dettagliate delle scelte effettuate durante il suo sviluppo, discussioni su campioni di codice pronti per la produzione, contributi alla letteratura Flutter esistente da una prospettiva empirica e soluzioni attentamente argomentate a problemi non banali.

In definitiva, questo documento presenta uno studio approfondito e conclusioni incentrate su un progetto specifico. Tuttavia, i suoi contenuti sono sufficientemente generali e preziosi da consentire ad altri sviluppatori di incorporare la sua applicabilità nella loro estrapolando le informazioni raccolte e adattandole ai loro requisiti particolari.

Parole chiave: App, Cross-Platform, Dart, Dependency Injection, Flutter, Manutenibilità, Mock, Modularizzazione, Scalabilità, Architettura del software, Gestione dello stato, Testabilità, Test

Contents

Abstract i

Abstract in lingua italiana iii

Contents v

Introduction 1
- 0.1 Native vs. Cross-Platform . . . 1
- 0.2 Flutter & App Architecture . . . 3
- 0.3 Document structure . . . 5

1 Background and Motivations 7
- 1.1 Flutter . . . 7
  - 1.1.1 Architectural overview . . . 8
  - 1.1.2 State Management . . . 16
- 1.2 Dart . . . 22
  - 1.2.1 Overview . . . 22
  - 1.2.2 Sound Null Safety . . . 23
  - 1.2.3 Asynchronous Programming . . . 23
  - 1.2.4 Packages . . . 24
- 1.3 Software Architecture . . . 25
  - 1.3.1 Overview . . . 25
  - 1.3.2 Layered . . . 26
  - 1.3.3 Feature-oriented . . . 27
- 1.4 Software Testing . . . 28
  - 1.4.1 Overview . . . 29
  - 1.4.2 Test Coverage . . . 31
  - 1.4.3 Dependency Injection . . . 32
  - 1.4.4 Mock . . . 33

1.5 Motivations . . . 34

2 Implementation 35

2.1 Context . . . 35

2.1.1 Technical aspects . . . 36

2.1.2 Development Process . . . 38

2.2 Hybrid Architecture . . . 39

2.2.1 Layered . . . 40

2.2.2 Feature-Oriented . . . 41

2.2.3 Packaging . . . 52

2.3 State Management . . . 55

2.3.1 Selection Criteria . . . 55

2.3.2 Code Samples . . . 56

2.4 Testing . . . 65

2.4.1 Preamble & Considerations . . . 65

2.4.2 Package Testing . . . 66

2.4.3 Bloc Testing . . . 71

2.4.4 Widget Testing . . . 73

2.4.5 Remarks . . . 80

3 Results 83

3.1 Quantitative Data . . . 83

3.2 Qualitative Data . . . 84

4 Related Work 91

5 Conclusions 93

Bibliography 95

A Appendix - Source Codes 101

B Appendix - App Screenshots 133

List of Figures 145

List of Tables 147

List of Source Codes 149

Introduction

The software application (App) industry has undergone a notable transformation in software development technologies in the last decade. Companies from all sectors and sizes have shifted from native-platform programming languages to their modern, cross-platform, typically-open-sourced counterparts to develop their suite of software products. This transformation responds to the rising need to address the problems of maintaining different native codebases. However, embracing youthful cross-platform technologies also comes at a cost, requiring further evaluation when considering this trade-off.

0.1. Native vs. Cross-Platform

Native development refers to creating Apps on a per-platform basis, meaning that deploying an App to *foreign* platforms is not feasible [3]. This approach encompasses a set of specific tools and languages designed to work within a native environment and benefits from full access and control over the platform's APIs. However, this development approach incurs high costs and requires advanced technical knowledge to reach multi-platform audiences [36]. Additionally, device fragmentation and vendor-specific modifications accentuate this problem, particularly in the realm of mobile devices [76]. Thus, the inability to reuse code across multiple platforms leads to redundant App implementations and inefficient development practices [16], urging companies and developers alike to seek solutions on other technologies.

Cross-platform development refers to the approach that does not consider a concrete software implementation but rather a general solution to running an App on several platforms from a single codebase [16]. These versatile and flexible characteristics have compelled top companies to develop their suite of software products or a part of it using cross-platform technologies [70] [32]. Furthermore, these technologies' open-sourced nature poses yet another significant factor in their adoption, leading to relevant academic research on the evolution of mobile software [35]. Figure 1 below illustrates the taxonomy of the main cross-platform approaches. Among the most popular and widely used tools and

frameworks derived from this categorization, we find Angular[1], React[2], and Vue.js[3] (Progressive Web Apps), Cordova[4] (Hybrid App), React Native[5], NativeScript[6], Xamarin[7], and Qt[8] (Self-contained runtime), DSLs[9], and UML[10] (MDSD), and lastly, XMLVM[11], J2ObjC[12], and Flutter[13] (Transpiling). Moreover, Flutter, React Native, Xamaring, Qt, and Cordova represent about 80% of the most used frameworks among developers worldwide [45], with Flutter and React alone representing roughly 80% of the cross-platform mobile framework market share [44].

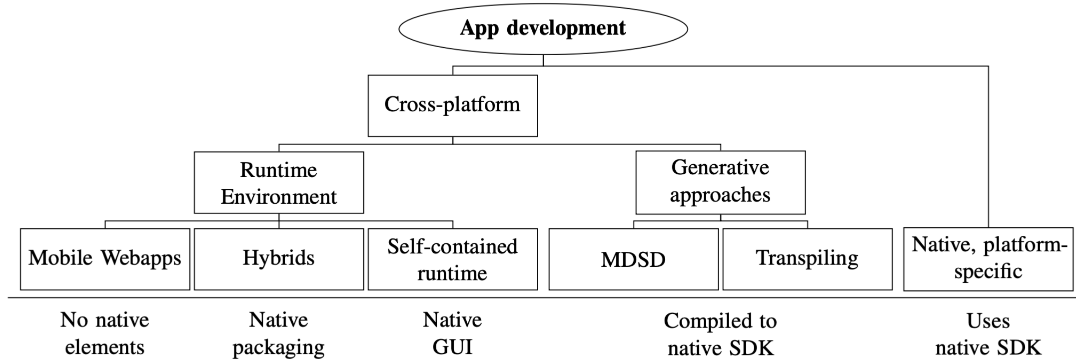

Figure 1: Categorization of Cross-Platform Approaches. [48]

Nonetheless, some of these solutions may suffer from limited functionality and performance issues compared to native development approaches. An all-too-common topic for discussion is the inferior performance exhibited by Apps developed with cross-platform technologies compared to natively-developed Apps. However, some studies demonstrate that, in particular cases, using frameworks that generate native apps might yield code that outperforms hand-written code due to optimization; interpreted apps could undergo runtime optimization that leads to better performance than apps optimized at compile-time [3].

[1]https://angular.io/
[2]https://reactjs.org/
[3]https://vuejs.org/
[4]https://cordova.apache.org/
[5]https://reactnative.dev/
[6]https://nativescript.org/
[7]https://dotnet.microsoft.com/en-us/apps/xamarin
[8]https://www.qt.io/
[9]https://en.wikipedia.org/wiki/Domain-specific_language
[10]https://www.uml.org/
[11]http://www.xmlvm.org/overview/
[12]https://developers.google.com/j2objc
[13]https://flutter.dev/

0.2. Flutter & App Architecture

Among the numerous cross-platform development options, Flutter has experienced the fastest growth in terms of mass adoption. This growth is remarkably striking in mobile app development, replacing React Native in the first position for most used mobile framework by software developers with a 42% market share - see Figure 2. Google released Flutter's first stable release on December 4th, 2018 [18], and, at the time of writing this document, it supports production-ready applications for Android, iOS, Web [37] [57], and Windows [60] - macOS and Linux stable support expected by the end of 2022. Moreover, its open-source repository on GitHub[14] has gathered over 135k stars - roughly 35k more stars than its closest cross-platform framework *competitor*, React Native[15] - which shows its substantial influence in the open-source software community.

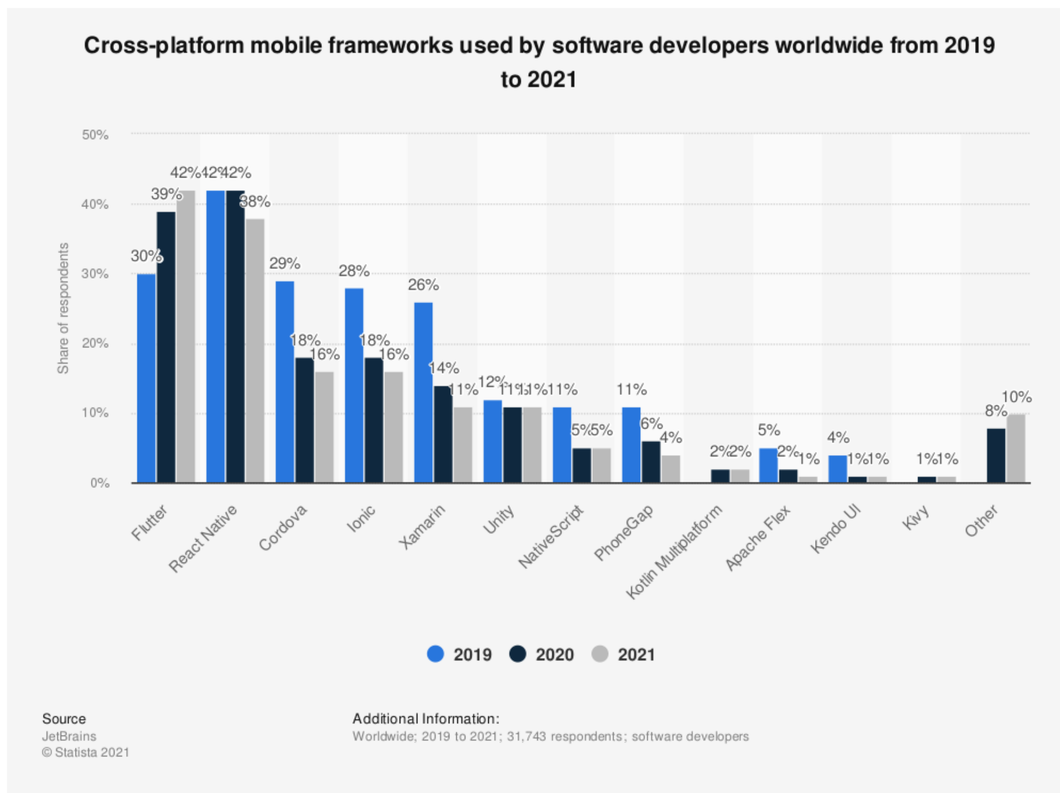

Figure 2: Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021 [44]

Flutter is an open-source framework supported by Google for building beautiful, natively compiled, multi-platform applications from a single codebase. Dart is the client-optimized programming language for any-platform fast apps powering this cross-platform framework.

[14] `https://github.com/flutter/flutter`
[15] `https://github.com/facebook/react-native`

Flutter code compiles to ARM or Intel machine code and JavaScript for fast performance on any device. Additionally, it supports Hot Reload [21] for enhanced productivity and allows developers to control every pixel to create customized, adaptive designs.

However, due to the young nature of this framework, there is no official consensus on how to architect Flutter Apps [64] [59]. Software architecture for mobile applications mainly focuses on the decisions and patterns applied to handle the app state and code modularization, which affects the app's testability support. Regarding state management, the Flutter team provides developers with numerous approaches [22] but does not enforce their usage or suggest which option is more fitting for a given scenario. As far as code modularization, Dart offers powerful capabilities to break up the codebase into collections of well-organized, independent, and reusable units called packages [23]. Nevertheless, there is no formal recommendation on which software architecture facilitates better compartmentalization means or a precise procedure to organize these packages. Lastly, despite the phenomenal tooling and resources available for testing Flutter applications [24], there exists a lack of best practices enabling a systematic methodology for thoroughly testing a Flutter App.

These factors motivated our decision to contribute to the current growing Flutter literature by proposing a solution that serves as a guide to navigating these ambiguous decisions when working on large-scale Flutter Apps. This proposal introduces a hybrid architecture, which leverages the strengths of layered and featured-oriented architectures. It implements the BLoC architectural pattern to manage the App's state. It provides precise modularization criteria to arrange the App's codebase into more manageable code units. Lastly, it offers valuable insights on the systematic testing techniques applied to obtain one hundred percent code coverage through unit and widget testing. Moreover, the analysis and development of this proposal took place within the context of a distributed software environment at a world-leading company in the domotics and home automation industry working on a cross-platform mobile App. Ultimately, the results of this analysis and implementation met the stakeholders' common high expectations and demands for the quality outcome of the project, who were professionals with different backgrounds and interests (software engineers, QA specialists, project managers, product owners, and beta testers).

0.3. Document structure

The rest of the document is organized as follows:

- **Chapter 1: Background and Motivations -** This chapter presents the reader with the central and most fundamental theoretical concepts required to comprehend the practical section of this document, including a comprehensive vision of Flutter, an introduction to Dart, a study of relevant software architecture approaches, and an in-depth evaluation of testing techniques, dependency injection, and mocking.
- **Chapter 2: Implementation -** This chapter introduces the core aspects of the large-scale Flutter App implemented during this thesis endeavor. This implementation serves as a tangible and practical solution built on top of the theoretical concepts and motivations presented in the previous chapter and the weaknesses found therein.
- **Chapter 3: Results -** This chapter navigates the reader through the results derived from the implementation endeavors presented in the previous chapter, providing quantitative and qualitative data supporting the objectives of this thesis.
- **Chapter 4: Related Work -** This chapter wraps up the work presented in this thesis by providing the reader with additional insights into other authors' academic endeavors related to the knowledge presented in this document. It reviews a series of research and thesis papers focused on software architecture, state management solutions, and testability in Flutter applications.
- **Chapter 5: Conclusions -** This chapter presents the final conclusions derived from the implementation and obtained results, and concludes by providing a series of future study lines for the proposed work.

1 | Background and Motivations

This chapter presents the reader with the central and most fundamental theoretical concepts required to comprehend the practical section of this document. Section 1.1 provides a comprehensive vision of Flutter, including an architectural overview 1.1.1 and an analysis of state management in this framework while focusing on the BLoC pattern 1.1.2. Moreover, section 1.2 introduces Dart as the underlying technology and programming language powering Flutter. Furthermore, we offer a study of the relevant software architecture approaches considered during the development of this project in section 1.3. The background research culminates with an in-depth evaluation of testing techniques, dependency injection, and mocking 1.4. Lastly, we use this background knowledge to argue the motivations behind our contributions, emphasizing the principal objectives of this endeavor in section 1.5.

1.1. Flutter

Flutter [25] is Google's portable framework for crafting cross-platform, natively compiled applications from a single codebase. Furthermore, it is open-sourced and, hence, free to use. Flutter consists of two essential parts [72]:

- An SDK (Software Development Kit): It is a collection of tools used by developers to create applications. It enables code compilation into native machine code.
- A widget-based framework (UI library): It is a set of reusable UI elements. They allow developers to design applications according to their specific needs.

"A powerful, general-purpose, open UI toolkit for building stunning experiences on any device, embedded, mobile, desktop, or beyond."

— **Tim Sneath**, Announcing Flutter 1.0 Keynote

Developers building Flutter Apps use the Dart programming language, further analyzed in subsection A.2. Google created this typed, object programming language that focuses on front-end development and released it in October 2021. Overall, Flutter combines ease

of development with similar-to-native performance while maintaining visual consistency across platforms [69].

1.1.1. Architectural overview

This subsection follows the official documentation and resources provided by the Flutter team [26] while including some complementary information from other reputable sources to achieve the utmost accuracy in the description of the core principles and concepts that form Flutter's design.

"The goal is to enable developers to deliver high-performance apps that feel natural on different platforms, embracing differences where they exist while sharing as much code as possible."

— **Flutter Team**, Flutter architectural overview [26]

Layers

An extensible and layered system underpins Flutter's design. These layers behave as independent libraries and mount one on top of the other, creating a bottom-up structure where each layer relies on the layer right below. It is worth noting that this architecture does not allow for privileged access from one layer to the one below and treats each framework level as optional and replaceable pieces.

A platform-specific **embedder** placed at the lowest level of this layered architecture facilitates an entry point to the underlying operating system. This embedder leverages an appropriate language for the platform (Android, iOS, MacOS, Windows, and Linux) and coordinates low-level operations and tasks such as rendering surfaces, accessibility, input, and message event loop management. Thereby, the underlying operating system perceives Flutter applications as any other native application enabling Flutter code integration as a module (package) or as the entire application content.

Moving up the layered structure, we find the Flutter **engine** placed at the core of this architecture. Written in C++, this engine supports the necessary primitives to build all Flutter applications and is responsible for frame rendering. **Skia**[1], an open-source 2D graphics library that provides APIs that work across a variety of hardware and software platforms, powers the graphics capabilities of this engine. Furthermore, the low-level implementation of Flutter's core API includes text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain.

[1] `https://skia.org/`

Lastly, we find the Flutter **framework** at the top of this architecture, which is essentially the layer developers interact with the most. The following subsections contain further descriptions about the most relevant sub-layers composing this modern, reactive, and Dart-written framework.

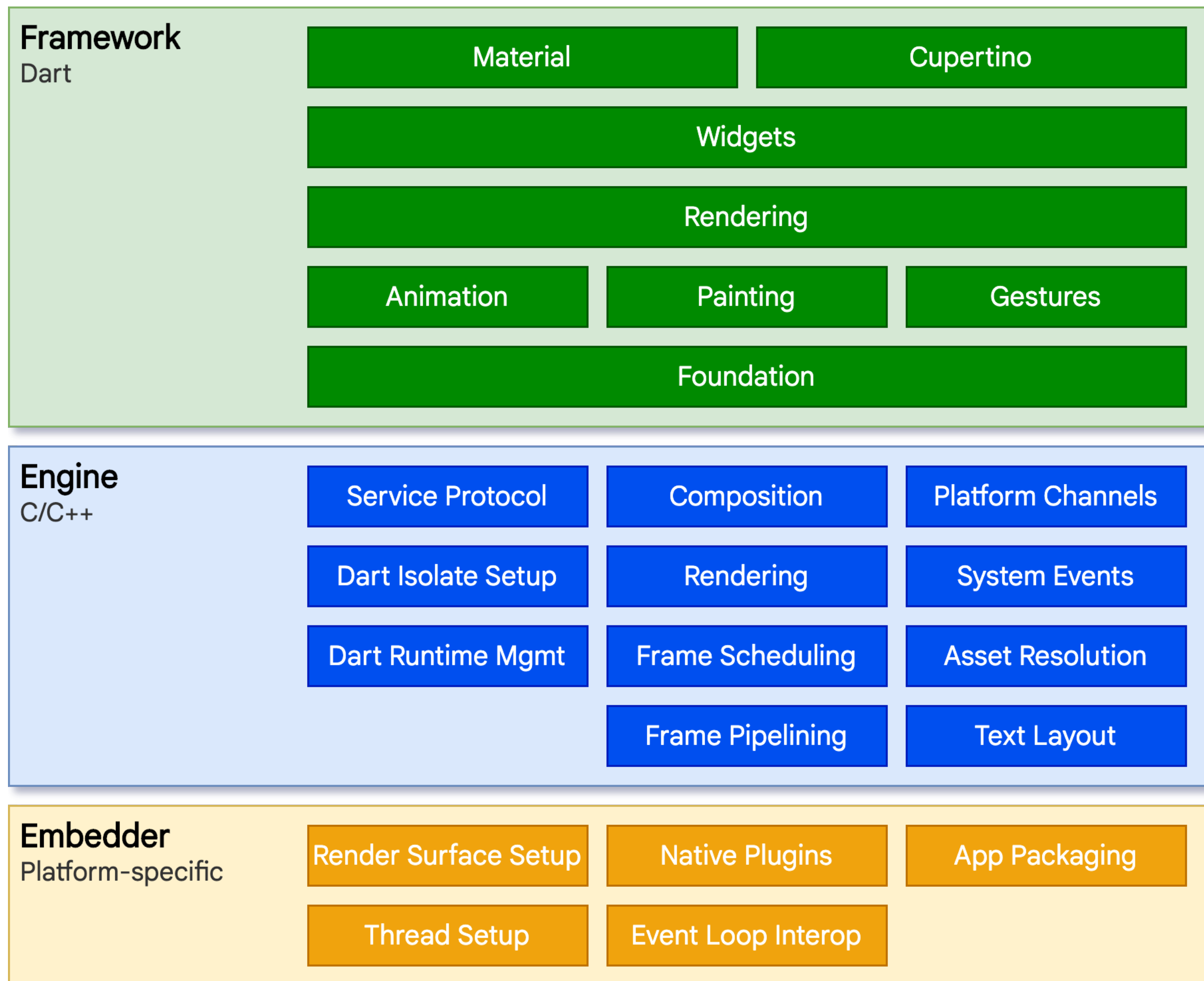

Figure 1.1: Flutter architectural layers [26]

It is worth noting that the Flutter framework is relatively lightweight and small, leaving the implementation of many higher-level features to developers through the use of packages - further explanations included in section 1.2.

Reactive User Interfaces

The Flutter team claims that "Flutter is a reactive, pseudo-declarative UI framework." Combining these two programming paradigms enables developers to map the application to the interface state, prompting the framework to update the interface at run time as the application state changes [40].

Flutter leverages this alternative-to-traditional-principles approach to overcome challenges such as state changes cascading throughout the entire User Interface (UI) or the out-of-sync problem exhibited by the Model View Controller approach. Thus, an explicit decoupling of the UI from its underlying state facilitates the creation of only the UI description, while the framework creates and updates the UI as needed.

Flutter uses immutable classes to configure a tree of objects, also known as widgets. These widgets allow for the management of layout and compositing separately through two different object trees where the former manages the latter.

"Flutter is, at its core, a series of mechanisms for efficiently walking the modified parts of trees, converting trees of objects into lower-level trees of objects, and propagating changes across these trees."

— **Flutter Team**, Flutter architectural overview [26]

Widgets override the **build()** method to declare their user interface. This function essentially converts state to UI: **UI = f(state)**.

Thanks to the fast object instantiation and deletion provided by Dart [63], the Flutter framework calls this method as many times as needed without negatively impacting the app's performance.

Widgets

As mentioned before, a **widget** is an immutable declaration part and the building block of a Flutter App's UI. The combination of small, single-purpose widgets is a powerful composition technique emphasized by Flutter. Furthermore, this composition creates a hierarchy of nested widgets where the root is typically a *MaterialApp* or a *CupertinoApp*. The code snippet below shows a simple example where all the instantiated classes are widgets forming this described structure.

Listing 1: Flutter Hello World

```
1  import 'package:flutter/material.dart';
2
3  void main() => runApp(const MyApp());
4
5  class MyApp extends StatelessWidget {
6    const MyApp({Key? key}) : super(key: key);
7
8    @override
9    Widget build(BuildContext context) {
```

```
10      return MaterialApp(
11        home: Scaffold(
12          appBar: AppBar(
13            title: const Text('My Home Page'),
14          ),
15          body: Center(
16            child: Builder(
17              builder: (BuildContext context) {
18                return Column(
19                  children: [
20                    const Text('Hello World'),
21                    const SizedBox(height: 20),
22                    ElevatedButton(
23                      onPressed: () {
24                        print('Click!');
25                      },
26                      child: const Text('A button'),
27                    ),
28                  ],
29                );
30              },
31            ),
32          ),
33        ),
34      );
35    }
36  }
```

Moreover, events prompt the framework to update the App's UI. This update derives from an efficient comparison between the old and new widgets, which updates the widget hierarchy. Most importantly, Flutter features unique widget implementations, rather than deferring to those provided by the system, leading to substantial benefits:

- Provides unlimited extensibility
- Avoids a significant performance bottleneck
- Decouples the application behavior from any operating system dependencies

Recall that overriding the **build()**[2] method determines a widget's visual representation. This function must be free of side effects and return a new tree of widgets. Then, the framework calls determined build methods based on the render object tree. Notice that

[2] https://api.flutter.dev/flutter/widgets/StatelessWidget/build.html

this building process relies on returning quickly from the method and the asynchronous execution of the computational work. Ultimately, Flutter rerenders only the widgets whose state has changed by calling their build method thanks to this effective automated comparison, enabling high-performance, interactive apps.

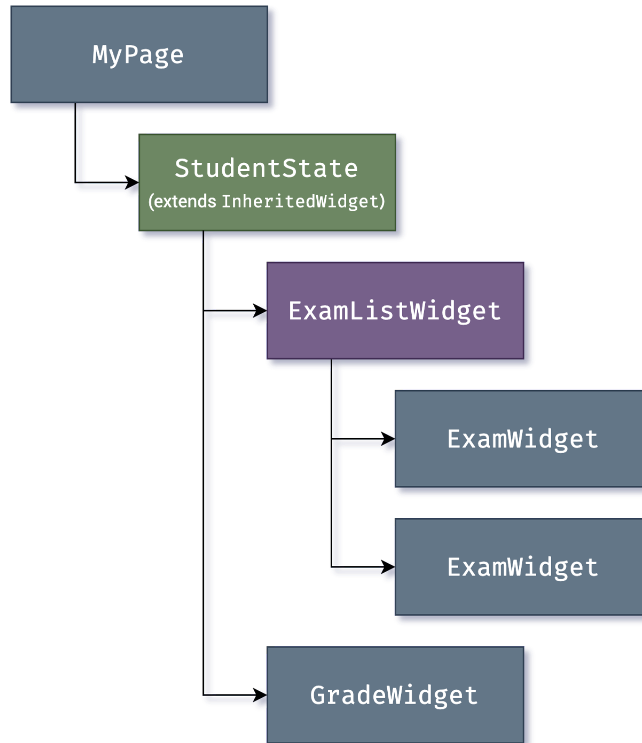

Figure 1.2: InhertitedWidget state example [26]

Furthermore, Flutter classifies widgets as *stateful* or *stateless*. A **StatelessWidget**[3] has no mutable state, and hence, none of its properties change over time. On the other hand, a **StatefulWidget**[4] has unique characteristics that may need to change based on user interactions or other factors. Thereby, developers can mutate a widget's State[5] by calling setState()[6], prompting the framework to update the UI by calling the State's build method. Nevertheless, managing state data up and down the tree hierarchy becomes troublesome and inconvenient, especially as the tree grows larger and deeper. Flutter introduces a third type of widget that facilitates data access from a shared ancestor, **InheritedWidget**[7]. Figure 1.2 above exemplifies this behavior.
Lastly, Flutter employs InheritedWidget broadly across the framework to manage shared

[3] `https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html`
[4] `https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html`
[5] `https://api.flutter.dev/flutter/widgets/State-class.html`
[6] `https://api.flutter.dev/flutter/widgets/State/setState.html`
[7] `https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html`

states, such as in the application's visual theme, Navigator[8], or MediaQuery[9]. However, scaling applications requires more advanced state management approaches, as described in subsection 1.1.2.

Rendering

To better understand Flutter's rendering pipeline shown in Figure 1.3 below, we introduce a high-level comparison to other common approaches [43] [62] [31].

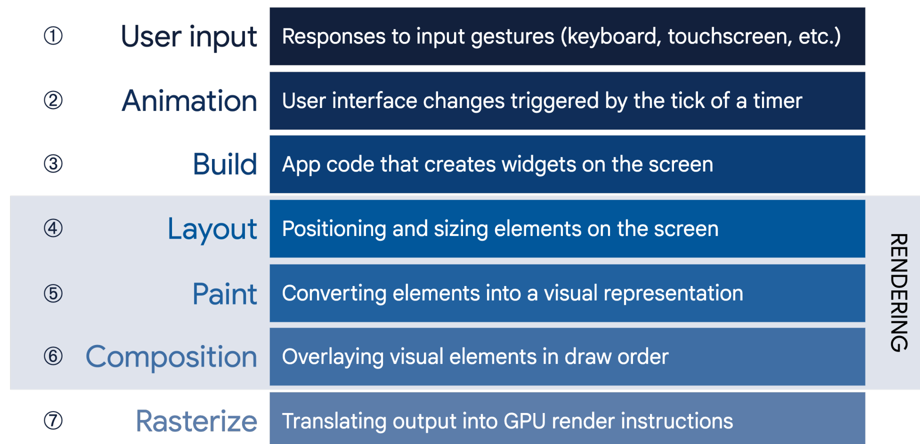

Figure 1.3: Flutter render pipeline [26]

Figure 1.4 below depicts the **native** approach to building mobile Apps where developers must use a platform-specific programming language. Thereby, Apps communicate with the platform to create widgets or access other services, like Bluetooth. The canvas then renders these widgets, which in turn receive the events. The main limitation of this approach is that, despite this straightforward and efficient architecture, developers must create different applications to target distinct platforms since widgets and native languages differ considerably.

[8] https://api.flutter.dev/flutter/widgets/Navigator-class.html
[9] https://api.flutter.dev/flutter/widgets/MediaQuery-class.html

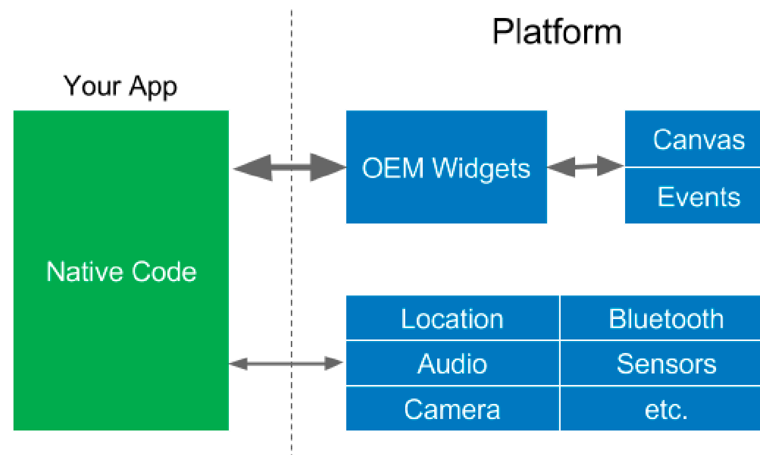

Figure 1.4: Native rendering approach [43]

Figure 1.5 displays the **WebViews** approach, an early effort at enabling cross-platform development. Apps based on this approach render a bundle of HTML, CSS, and JavaScript onto a Web View much as it occurs on a mobile browser. This approach's main drawbacks are the web technology stack's limitations and the need to use a bridge to communicate with the platform's services. The latter consumes resources and time, worsening the application's performance.

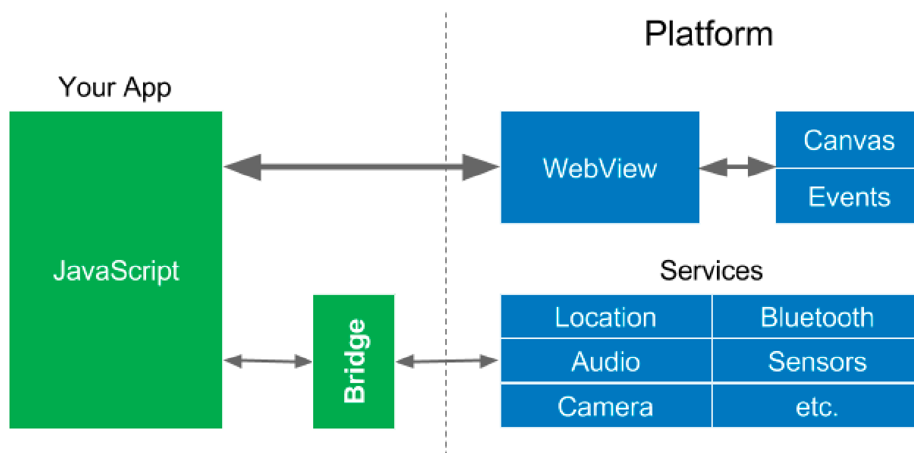

Figure 1.5: WebView rendering approach [43]

Reactive Views, shown in Figure 1.8, also known as Interpreted, attempt to create cross-platform solutions leveraging JavaScript. Frameworks like ReactJS and React Native use this approach where the creation of web views relies on reactive programming patterns [54]. This approach's main limitation is, once again, the need to use a bridge to communicate with the platform, negatively impacting the application's performance.

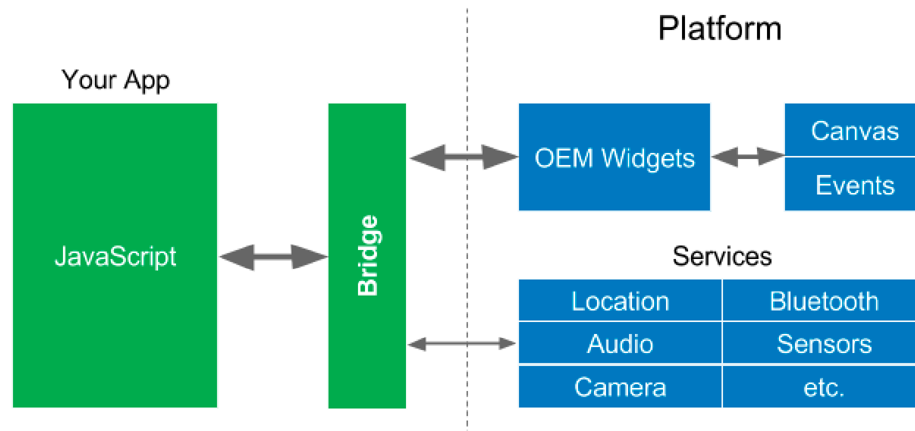

Figure 1.6: Reactive Views rendering approach [43]

Overall, cross-platform frameworks leverage an abstraction layer above the underlying native libraries. Hence, resulting in significant overheads, especially when there is a non-negligible number of interactions between the UI and the application's logic.

"The overriding principle that Flutter applies to its rendering pipeline is that ***simple is fast****."*

— **Flutter Team**, Flutter architectural overview [26]

By contrast, Flutter bypasses the system UI widget libraries in favor of its unique widget set, minimizing said abstractions. The first thing to notice from Figure 1.7 is that Flutter uses native ARM binary code, which is compiled ahead of time and does not require a JVM. Furthermore, Flutter uses Skia to render its visuals and also embeds a copy of this graphic engine as part of its engine. Hence, allowing applications to remain updated with the latest performance improvements regardless of the native platform's updates.

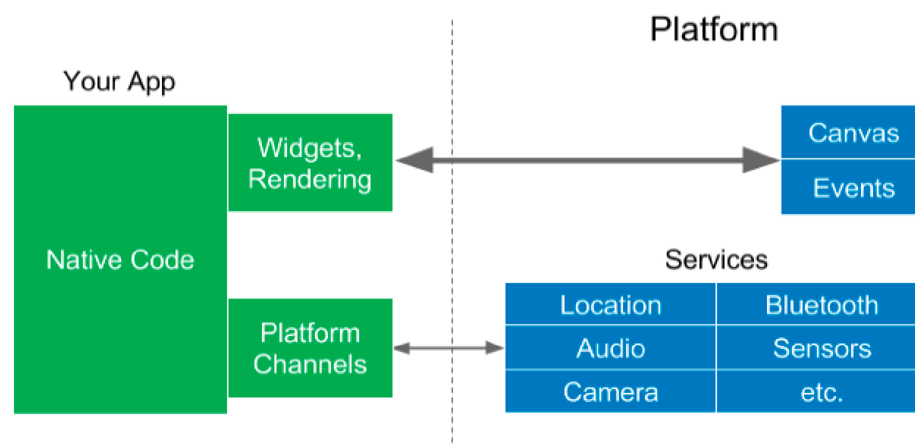

Figure 1.7: Flutter rendering approach [43]

1.1.2. State Management

From the Flutter API documentation [20], state is information that (1) can be read synchronously when the widget is built and (2) might change during the lifetime of the widget. In a broader sense, the state of an application is its condition or quality of being at a given moment in time [38]. Thus, in the context of Flutter, state management refers to how an application handles the distribution of state throughout the widget tree [31].
Furthermore, Flutter is a declarative framework, meaning that it builds the UI based on the App's current state. Other imperative frameworks' performance, like Android SDK or iOS UIKit, would suffer if they continuously rebuilt their UIs from scratch on state updates. However, Flutter manages to do so at a remarkable speed, even on every frame if needed. The most notable benefit from this approach is the single-code path for any state of the UI, as developers only need to describe what the UI should look like for any given state, once.

Figure 1.8: Flutter declarative formula [27]

The Flutter team emphasizes that broad definitions for the state of an application, though valid, are not suitable when architecting an App as developers do not manage everything that fits these definitions [28]. Therefore, they focus on the state that developers do manage and make the following differentiation:

- **Ephemeral** state, also known as UI or Local state, refers to the state efficiently contained within a widget. It does not perform complex changes, does not require serialization, and other parts of the widget tree hardly ever need to access it.
- **App** state, or shared state, is not ephemeral. There is a need to share this state across different parts of the App and maintain it between user sessions. Managing this state is more involved and typically requires a state management solution to deal with its complexity within the context of an application's nature.

Nonetheless, the Flutter team considers "there is no clear-cut rule" to distinguishing between ephemeral and App states. Indeed, they do not even enforce a systematic approach to choosing a state management solution for a large-scale application. Thus, they provide

a list of state management approaches [22], though they leave it up to developers which option to choose. Therefore, we picked the BLoC pattern to manage the state of the Flutter application evaluated and discussed in this document. We based the outcome of our decision on a thorough analysis of the BLoC pattern, and the bloc library packages [4]. Moreover, we also considered previous research studies that examined the most relevant state management solutions for Flutter applications [64] [59] [31] [39]. These studies determined that the BLoC pattern made Flutter Apps more scalable and testable, making it the most suitable solution for our project needs.

BLoC Pattern

Paolo Soares introduced BLoC in 2018, a state management solution based on a clear architectural pattern [61]. He aimed at sharing as much code as possible for an application implemented with two different Dart-based frameworks by reusing the code in charge of the business logic. Hence, he placed all the business logic into independent **B**usiness **Lo**gic **C**omponents (BLoCs) away from the framework-specific UI components. Lastly, he presented the following strict rules for BLoCs:

1. Inputs and outputs are simple Streams/Sinks only
2. Dependencies must be injectable and platform agnostic
3. No platform branching allowed
4. Implementation can be whatever you want if you follow the previous rules
5. Though, he suggests reactive programming

As for the UI design guidelines:

1. Each "complex enough" component has a corresponding BLoC
2. Components should send inputs "as is"
3. Components should show outputs as close as possible to "as is"
4. All branching should be based on simple BLoC boolean outputs

"Design rules are not negotiable, and that's for the sanity of everyone."

— **Paolo Soares**, BLoC presentation at DartConf 2018 [61]

By following this pattern and its rules, an App will feature the following architectural advantages [31]:

1. Dedicated business logic components
2. Simple, logic-agnostic UI components
3. Testable business logic
4. BLoCs control widget rebuilding
5. Predictable state changes which must come from a single place

bloc library

Felix Angelov released the bloc package's first version in October 2018 [5]. This package is the centerpiece of a library of seven other packages whose focus is to facilitate the implementation of the BLoC design pattern in a simple, lightweight, and highly-testable manner. Thus, we introduce the reader to the subset of packages implemented in the Flutter App discussed in section 2.3, where we will further demonstrate and analyze their implementation and use-cases.

"A predictable state management library that helps implement the BLoC design pattern."

— **Felix Angelov**, bloc library [4]

The **bloc** package represents a high-level abstraction of the BLoC pattern. It enables developers to focus on writing the business logic rather than on the complex reactive aspects and boilerplate required to implement this pattern from scratch. Thereby, this package's public API exposes two classes to implement the bloc pattern, Cubit[10] and Bloc[11], both extending the base class BlocBase[12]. Cubits and Blocs are reasonably similar, though the former relies on methods to emit new states, while the latter leverages Streams and Events as inputs to output a Stream of States.

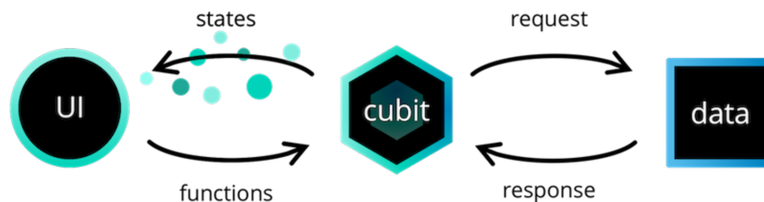

Figure 1.9: Cubit architecture [5]

[10] https://pub.dev/documentation/bloc/latest/bloc/Cubit-class.html
[11] https://pub.dev/documentation/bloc/latest/bloc/Bloc-class.html
[12] https://pub.dev/documentation/bloc/latest/bloc/BlocBase-class.html

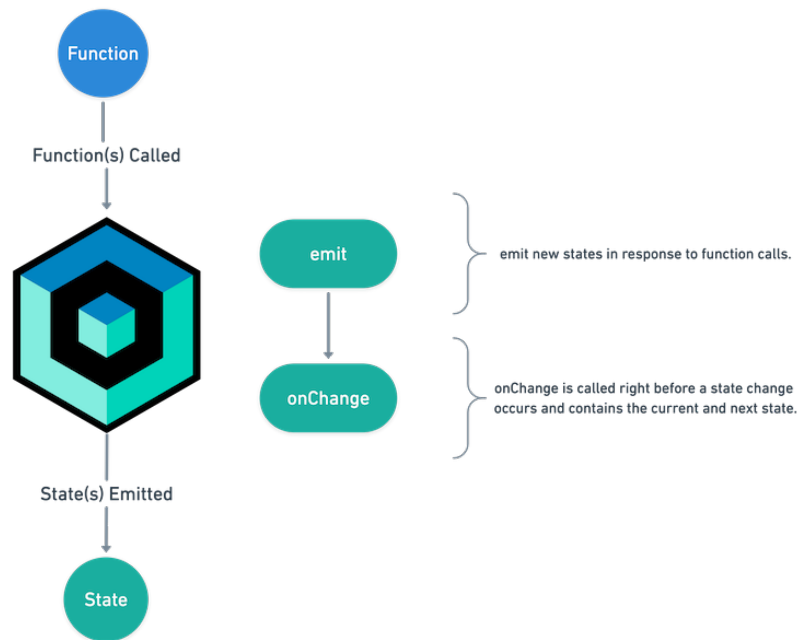

Figure 1.10: Cubit flow [5]

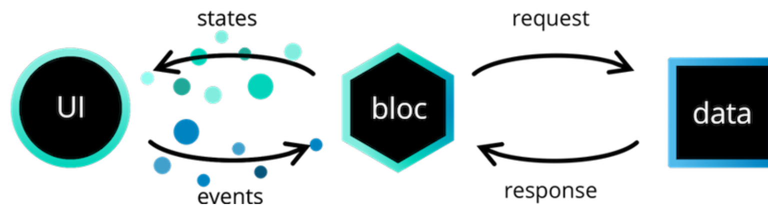

Figure 1.11: Bloc architecture [5]

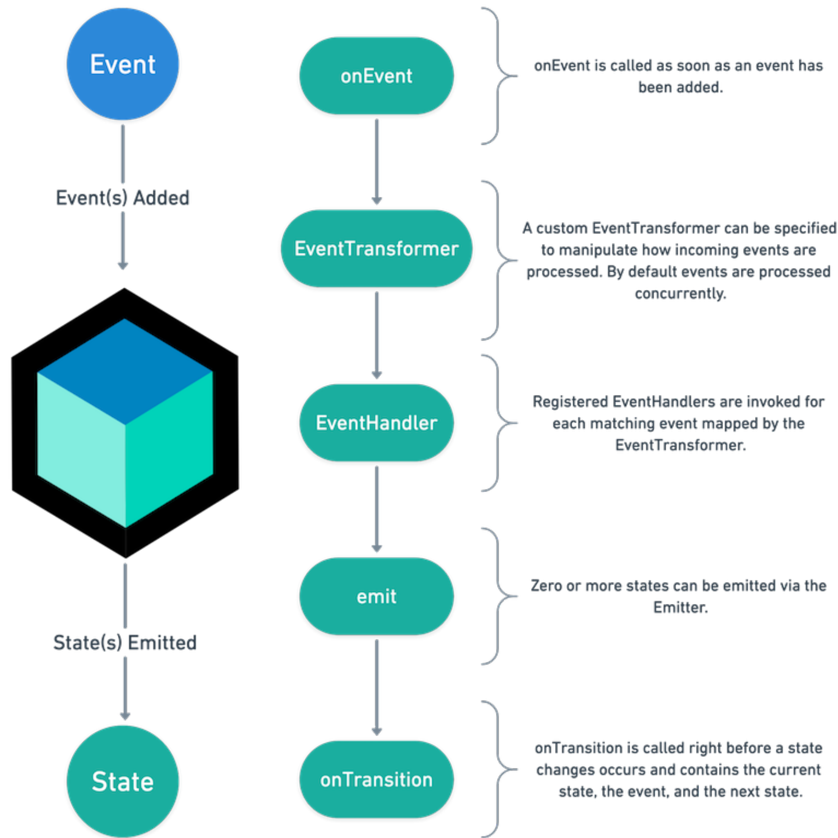

Figure 1.12: Bloc flow [5]

Moreover, the **flutter_bloc** package [6], built to work with the previously reviewed bloc package, provides developers with a collection of Flutter Widgets that facilitate the implementation of the BLoC design pattern. Among its most relevant Widgets, we find

- BlocBuilder[13], which handles building a widget in response to new states
- BlocSelector[14], which allows developers to filter updates by selecting a new value based on the bloc state and prevents unnecessary builds if the selected value does not change
- BlocProvider[15], which is responsible for creating the Bloc or Cubit and the child having access to either instance via *BlocProvider.of(context)*
- MultiBlocProvider[16], which merges multiple BlocProvider widgets into one widget

[13]https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/BlocBuilder-class.html

[14]https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/BlocSelector-class.html

[15]https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/BlocProvider-class.html

[16]https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/MultiBlocProvider-class.html

tree, improving readability and eliminating BlocProvider nesting

- BlocListener[17], which guarantees to invoke the listener in response to a state change only once
- MultiBlocListener[18], which merges multiple BlocListener widgets into one widget tree, improving readability and eliminating BlocListener nesting
- BlocConsumer[19], which exposes a builder and a listener to react to new states analogously to a nested BlocListener and BlocBuilder but reducing the required boilerplate
- RepositoryProvider[20], which is responsible for creating the repository and the child having access to the repository via *RepositoryProvider.of(context)*
- MultiRepositoryProvider[21], which merges multiple RepositoryProvider widgets into one widget tree, improving readability and eliminating RepositoryProvider nesting

Then, the **hydrated_bloc** package [7], also built to work with the bloc package, is an extension on this package that automatically persists and restores bloc and cubit states. It exports a Storage interface enabling developers to work with any storage provider, though it also provides a built-in, hive[22]-based implementation through HydratedStorage[23].
Lastly, the **bloc_test** package[8] is a Dart-based library that simplifies the testing of blocs and cubits.

[17] https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/BlocListener-class.html
[18] https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/MultiBlocListener-class.html
[19] https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/BlocConsumer-class.html
[20] https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/RepositoryProvider-class.html
[21] https://pub.dev/documentation/flutter_bloc/latest/flutter_bloc/MultiRepositoryProvider-class.html
[22] https://pub.dev/packages/hive
[23] https://pub.dev/documentation/hydrated_bloc/latest/hydrated_bloc/HydratedStorage-class.html

1.2. Dart

This section presents the reader with an overview of Dart and its main characteristics and powerful features.

1.2.1. Overview

Dart [65] forms Flutter's foundation by providing the language and runtimes to power this framework's Apps. Its technical envelope's[24] design is particularly well-suited for client development, prioritizing both development and high-quality production experiences across a wide variety of compilation targets (web, mobile, desktop, and embedded).
Moreover, Dart uses static type checking to ensure that a variable's value always matches the variable's static type, making it type-safe. It also supports *dynamic* types combined with runtime checks, offering further flexibility to the language's typing system. Overall, Dart is a client-optimized language for developing fast apps on any platform. Thereby, Dart allows code compiling into these different platforms:

- **Native** - For applications targeting mobile and desktop devices, Dart includes both a Dart VM with just-in-time (JIT) compilation and an ahead-of-time (AOT) compiler for producing machine code
- **Web** - For apps targeting the web, Dart includes a development time compiler (dartdevc) and a production time compiler (dart2js). Both compilers translate Dart into JavaScript.

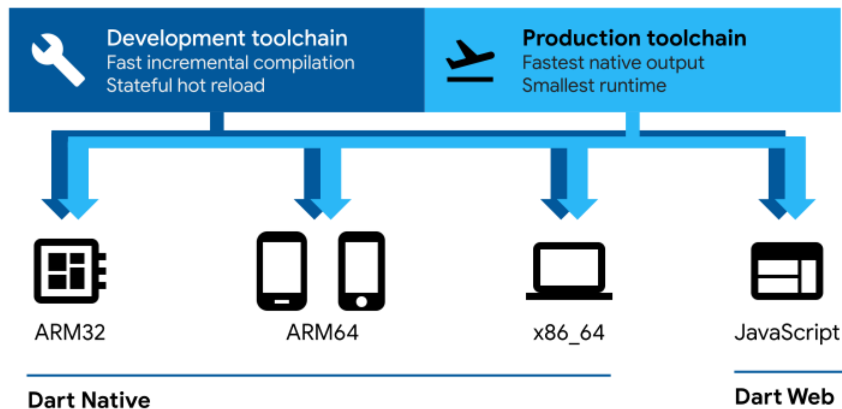

Figure 1.13: Dart platforms [65]

[24]Choices made during development that shape the capabilities and strengths of a language

1.2.2. Sound Null Safety

Dart's null safety support deserves special attention as it forces variables to be non-nullable by default unless developers define it otherwise in their code. Additionally, null safety turns runtime null-deference errors into edit-time analysis errors enhancing the development experience while minimizing runtime exceptions and bugs. Ultimately, Dart's null safety support builds upon the following three core design principles:

- **Non-nullable by default** - Variables are non-nullable unless explicitly declared otherwise
- **Incrementally adoptable** - Developers decide what and when to migrate to null safety, allowing for incremental migrations and mixing null-safe with non-null-safe code
- **Fully sound** - Dart's sound null safety allows for compiler optimizations as non-nullable types can never become nullable

"Null safety is the largest change we've made to Dart since we replaced the original unsound optional type system with a sound static type system in Dart 2.0."

— **Bob Nystrom**, Understanding Null Safety [51]

1.2.3. Asynchronous Programming

Asynchronous programming is a notably relevant subject in Dart programming and Flutter. Leveraging its powerful features allows to architect an App and its code in a more organized and efficient manner. Furthermore, the Future[25] and Stream[26] classes characterize asynchronous programming in Dart.

Future

A future is the result of an asynchronous computation, which is a computation that cannot return an immediate result after being executed [66]. Thereby, this result may have two states, uncompleted or completed. The former refers to a future waiting for a function's asynchronous operation to finish or throw an error, while the latter refers to a successful or failed completed computation.

Stream

A stream is a sequence of asynchronous events, distinguished as data or error events [67]. To simplify this concept, Didier Boelens [13] considers a pipe with two ends, where data

[25] https://api.dart.dev/stable/2.16.1/dart-async/Future-class.html
[26] https://api.dart.dev/stable/2.16.1/dart-async/Stream-class.html

and events always flow in the same direction from one end to the other, never vice versa. "The pipe is called a Stream. To control the Stream, we typically use a StreamController. To insert something into the Stream, the StreamController exposes the entrance, called a StreamSink, accessible via the sink property. The Stream's way out is exposed by the StreamController[27] via the stream property."
Notice Streams may convey any type of data, such as a value, an event, an object, a collection, a map, an error, or even another Stream. Moreover, there are two types of Streams:

- **Single-subscription** - Only allows a *single* listener throughout the Stream's lifetime. Hence, it is impossible to listen twice on such a Stream, even after canceling the first subscription.
- **Broadcast** - Allows *any* number of listeners. It is possible to add new listeners to a Broadcast Stream at any given point, having new listeners receive the events as soon as they start listening to the Stream.

1.2.4. Packages

Understanding the parts that compose more complex and larger entities enable software engineers to architect applications leveraging functional and behavioral modularity principles. Functional modularity refers to the composition of smaller independent components with clear boundaries and functions. On the other hand, behavioral modularity addresses traits and attributes that can evolve independently. Thus, complex software applications may be broken into functional parts called modules, which can be created, changed, tested, used, and replaced separately. Software modularity brings the following benefits to software systems [2]:

- More lightweight modules with less code
- Introduction of new features or changes to modules in isolation, separate from the other modules
- Easy identification and fixing of module-specific errors
- Modules can be built and tested independently
- Enhanced collaboration for developers working on different modules for the same application
- Reusability of modules across various applications
- Modules kept in a versioning system can be easily maintained and tested
- Module fixes and noninfrastructural changes do not affect other modules

Thereby, Dart favors modularity and provides an ecosystem based on packages to manage

[27] https://api.dart.dev/stable/2.16.1/dart-async/StreamController-class.html

shared software such as libraries and tools [68]. At a minimum, a Dart package is a directory containing a pubspec[28] file, a yaml[29]-based file containing metadata about the package. Notice packages may contain dependencies[30], libraries, applications, resources, tests, images, and examples. Moreover, the concept of separation of concerns between objects in object-oriented programming (OOP) resembles a Dart library, which exposes functionality as a set of interfaces and hides the implementation from the rest of the world. Libraries allow for a better application structure, tight coupling minimization, and more maintainable code. Ultimately, a Dart application is a library, as well.

1.3. Software Architecture

This section aims at reviewing the concept and implications of Software Architecture. Moreover, it presents the reader with the two designs leveraged during the implementation part of the project, layered and featured-oriented architectures. Notice this chapter does not intend to be a comprehensive evaluation of Software Architecture which includes detailed information about the vast literature destined to cover this topic, but rather a focused and precise analysis of a subset of software-architecture-related matters having direct implications in the development of the Flutter application described in section 2.2.

1.3.1. Overview

Shlaer and Mellor might have been the first to use the expression *software architecture* in academic literature [58]. Software architecture involves a series of decisions based on many factors in a wide range of software development [30], representing the highest abstraction level [14] at which we construct and design software systems. Moreover, it enhances the traceability between requirements and technical solutions, reducing risks associated with building the technical solution and facilitating the satisfaction to systemic qualities. In other words, the software architecture sets the boundaries for the quality levels resulting systems can achieve and represents an early opportunity to design for software quality requirements such as reusability, performance, safety, and reliability [15].

"To design the software architecture to meet the quality requirements is to reduce the risks of not achieving the required quality levels."

— **PerOlof Bengtsson**, Software Architecture - Design and Evaluation [15]

M. Kassab et al. provided an empirical study where they conducted a comprehensive

[28] `https://dart.dev/tools/pub/pubspec`
[29] `https://yaml.org/`
[30] `https://dart.dev/tools/pub/glossary#dependency`

survey to document the practices used by software professionals when selecting and incorporating architectural patterns for their projects in the industry [30]. Based on the survey results and the quality requirement criteria that led to the most used approaches, we considered that a combination of flexible yet powerful architectural patterns would ensure and improve a given software application's maintainability, testability, and scalability. Thus, we decided to employ a hybrid architecture combining layered and feature modularization to implement our Flutter App, among other patterns further discussed in section 2.2.

"The architecture of a software system is almost never limited to a single architectural pattern."

— **Microsoft**, Microsoft Application Architecture Guide: Patterns & Practices [1]

1.3.2. Layered

The layered software architecture pattern, also known as the n-tier architecture style or the multi-layered architecture style, is arguably the most commonly used architectural pattern in software engineering. In essence, the layered architecture's goal is to organize the components of an application into horizontal logical layers and physical tiers [71].

Layers are role-and-responsibility separated logical units within an application which manage their specific software dependencies. Higher layers may exploit services from a lower layer, but never the other way around. Moreover, tiers are physical units whose fundamental purpose is to run code, like a web server or a database. Notice layers may be hosted in dedicated and exclusive tiers, though this is not required. More importantly, the physical separation of tiers enables better scalability, maintainability, and resiliency. However, it may increase latency due to network communication overhead.

A traditional layered software architecture features three tiers and four layers. Notice that introducing multiple layers for different software components enhances the separation of concerns, and hence, it improves the components' simplicity, maintainability, and testability.

Regarding the four common layers,

- **Presentation** - This layer contains the user-exposed UIs.
- **Business Logic** - This layer handles all business logic, validations, and business processes.
- **Data Access** - Also known as the persistence layer, this layer is responsible for interacting with a database.
- **Data Store** - This layer is the actual data store for the application.

As for the three tiers,

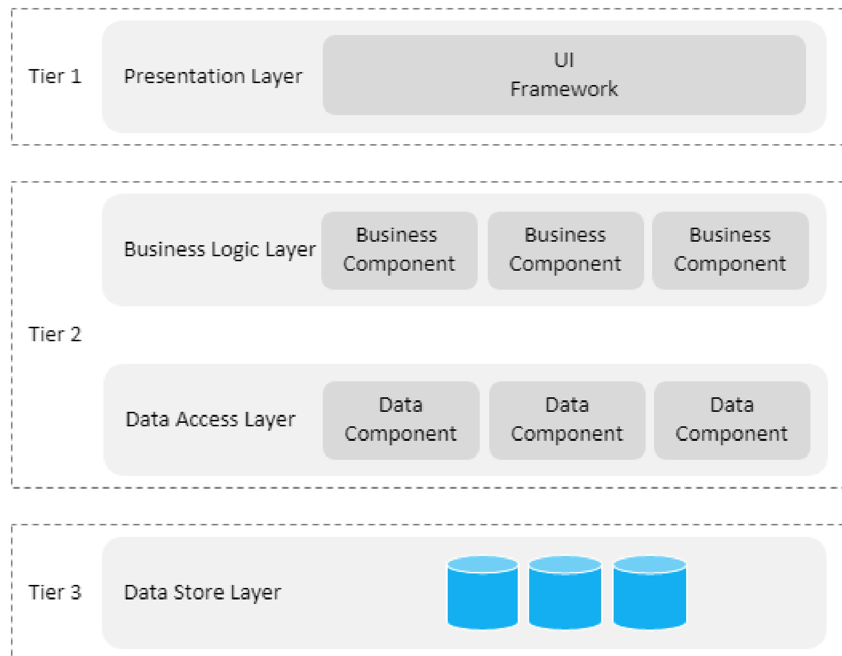

Figure 1.14: Layered architecture [71]

- **Presentation** - This tier hosts the front-end codebase. It is the application's top-most level, and it enables user interaction.
- **Application** - This tier hosts the business logic layer and data access layer. It is also known as the middle tier.
- **Data** - This tier hosts the data store layer. Databases, file systems, blob storage, or document databases are examples of resources found in a data store.

1.3.3. Feature-oriented

The number of notions and interpretations of a feature is as broad and abstract as definitions may have. We propose a broad and fundamental-concept-encompassing definition combining the descriptions provided by Kang et al. [42] and S. Apel and C. Kästner [9]. A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems representing a functionality entity that satisfies a requirement, serves a design decision, and provides a potential configuration option.
"The feature-oriented concept is based on the emphasis placed by the method on identifying those features a user commonly expects in applications in a domain."

— **Kyo C. Kang et al**, Feature-Oriented Domain Analysis (FODA) Feasibility Study [42]

Moreover, we find Feature-Oriented Software Development (FOSD) particularly relevant to observe and comprehend the reasoning behind the feature-related decisions supporting our application architecture. FOSD is a paradigm for the construction, customization, and synthesis of large-scale software systems aiming at decomposing such a system based on the features it provides [9]. Decomposition enables the creation of well-structured and user-needs-tailored software systems while facilitating the reuse of shared common features to generate different software systems. Ultimately, FOSD aims at three core properties: structure, reuse, and variation; and comprises four phases: domain analysis, domain design and specification, domain implementation, and product configuration and generation.

- **Domain analysis** - Determines which features are part of the software system or product line, including a domain scoping step that defines its dimension and having feature modeling as its core activity.
- **Domain design and implementation** - Developers typically create a set of first-class feature artifacts that encapsulate design and implementation information.
- **Product configuration and generation** - A generator can pick the corresponding feature artifacts and create an efficient and reliable software product based on the user-desired features and domain knowledge.

It is worth noting that, despite the numerous existing languages and tools for feature implementation, they often do not interoperate, causing integration problems during other phases of the FOSD process. Moreover, specifying, verifying, and testing the correctness of generated software products presents a key challenge. Thus, the work presented in this study leverages the main feature-related concepts from FOSD but does not leverage automated software generation tools during the project implementation and configuration, avoiding unnecessary and out-of-scope complexities and overheads. Conversely, we exclusively relied on human-based software implementation and configuration built upon the domain analysis, design, and implementation previously worked out by the engineering and requirement teams of the domotics company.

Ultimately, feature-oriented architectures promote the creation of manageable projects constructed upon highly maintainable, scalable, and testable units.

1.4. Software Testing

This chapter presents a comprehensive overview of testing, and then it introduces the reader to test coverage, dependency injection, and mocks. Notice this content leverages the brilliant research performed by M. Veng in his Master's Thesis [73]. It then concludes with a review of testing within the Flutter framework. These topics and contents are

imperative to thoroughly understand the testing part of the project found in section 2.4.

1.4.1. Overview

Software literature dates back to the early 1970s when Hetzel Hetzel organized the first conference about software testing. Back then, engineers viewed software testing as destructive, meaning they aimed at finding errors within a given problem rather than constructive, which would have urged engineers to have working and fault-free software systems. With the gain in popularity, software testing's view shifted towards error prevention, maintainability, and capability measuring during the 1980s. Most importantly, due to its importance in assuring software quality, software safety, and customers satisfaction, software development has integrated software testing into its life cycle in recent years [73]. Moreover, we find two terms that define the nature of a given test. *Black-box* testing refers to those tests where the tester does not know the internal structure and algorithms of the software. On the other hand, *white-box* testing assumes testers know the internal structure and understand the detailed process of the software. Thus, we find the following types of tests:

- **Unit test** - assesses the smallest unit of a system, meaning a method, function, or procedure. Theoretically, it does not involve any other components, though avoiding connection to external resources is not always feasible in practice. As we will see in section 2.4, mocking techniques may assist developers in complying with unit testing principles by keeping each unit isolated.
- **Module test** - is a class or package test concerning many programmers or a whole team. However, its differences from a unit test are ambiguous and even subjective, as testing all units within a module is equivalent to testing the module. Therefore, we will treat module and unit testing as the same procedure and refer to both as unit testing here forth.
- **Integration test** - combines all components which build up the whole system into a test targeting the system workflow, ensuring that the system's functionalities meet the user requirements. Notice integration testing usually takes longer than unit testing due to connection to external resources and dependencies.
- **Acceptance test** - corresponds to the user acceptance of the system. Users perform the required acceptance test to ensure the working order and the correctness of the system.

It is worth acknowledging that software engineers and developers perform unit and integration testing. Hence, we can consider these tests as white-box testing since they are aware of the implementation and inner workings of the system. Conversely, the

acceptance test is a type of black-box testing since the users who carry out these tasks are unaware of the underlying system dependencies.

Furthermore, Figure 1.15 illustrates fundamental concepts, which will allow the reader to dive deeper into more specific and involved testing aspects after the following review.

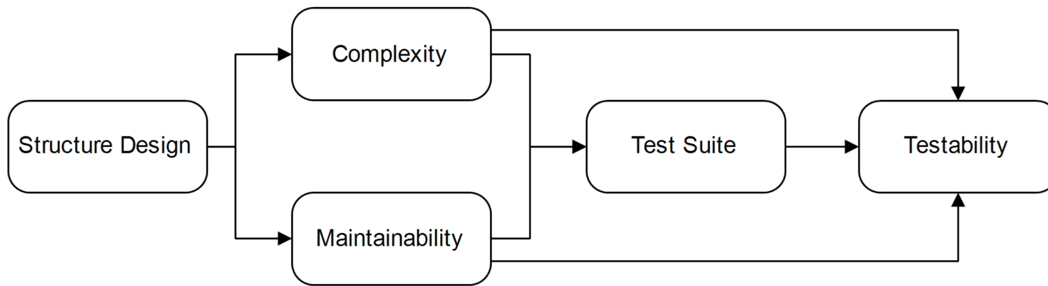

Figure 1.15: Testing conceptual framework [73]

The structure design corresponds to the software system's location where component organization and interaction occurs. Furthermore, it indicates the internal code structure found in a software system. This code structure carries an associated complexity and maintainability, impacting the overall system.

"The ease of modifying software components to include or remove capability, to tweak performance, and to fix defects."

— **Steve McConnell**, Code complete: A practical handbook of software construction [50]

From the definition of maintainability, we can observe the close relationship between complexity and maintainability. Moreover, the collection of test cases defined for a given software system composes the system's test suite. The test suite's objective is threefold, as it reveals system faults, assesses the quality, and ensures the correctness of its functionalities. Thus, maintainability and complexity are tightly related to the number of test cases found in a system's test suite. In other words, more complex systems require higher maintainability efforts and a larger number of test cases.

"The ease of a system, component or requirement which allows test criteria to be created and the ease of performing the test execution to verify the satisfaction of the test criteria."

— **IEEE**, IEEE Standard Glossary of Software Engineering Terminology [41]

In simple terms, testability addresses the ease with which a system can be tested. Notice that the test suite, complexity, and maintainability affect a given system's testability. Additionally, there are two essential facets of testability [12] [33]:

- **Controllability** - Degree of control software testers have over the system's input enabling testers to predict the system's output value.
- **Observability** - Degree of verification software testers have over the system's output value provided a given input value enabling testers to verify the test creation and execution correctness.

Ultimately, testability is one of the core features under maintainability. Thus, studying testability relates to maintainability and vice versa.

1.4.2. Test Coverage

Zhu et al. [77] contributed to the testing literature by establishing the standard for analyzing the test criteria behind test case considerations, also known as test adequacy. The outcome of this analysis includes:

- **Statement coverage** - testers generate test cases to assert the execution of every code statement at least once during the test process.
- **Branch coverage** - testers design test cases based on branch conditions (if-then-else, switch-case, and loop). It relies on an adequacy criterion resulting from the percentage coverage of the program execution paths.
- **Path coverage** - the program flow from the entry point to the exit point determines this test adequacy criterion.
- **Mutation adequacy** - testers aim to detect program faults by injecting artificial faults or mutants into the system. This aim dates back to the origins of software testing and its destructive view.

One relevant consideration worth noting is the glaring similarity between branch and path coverage, as both depend on the program's control. Thus, M. Veng concluded that the distinction between branch coverage and path coverage emerges with the existence of the loop statement as path coverage may encounter problems with infinite applicability [33]. Furthermore, test adequacy provides a twofold service within software testing roles. Firstly, it helps determine the stopping rule for creating test cases. For instance, testers following statement coverage could deem adequate the tests exercising all the tests inside a program. Secondly, it helps determine the test quality by quantifying it into a percentage, enabling testers to assess whether the overall test meets the acceptable quality criteria. For instance, we strived for one hundred percent coverage as the quality criteria for the Flutter application discussed in section 2.4. Thereby, we successfully met the determined quality criteria by having our test cases cover all branches, which gave developers tangible confidence about the completeness of their tests.

Zhu et al. [77] used the information from the software requirements and its internal

structure to categorize test adequacy into three criteria-based groups:

- **Specification-based** - leverages software specifications to drive testers into designing test cases that ensure meeting the requirements.
- **Program-based** - follows the white-box testing concept to define test cases based on the program's internal structure prompting testers to consider statement coverage, branch coverage, and path coverage when generating said test cases
- **Interface-based** - applies testing criteria not based on internal program information or specifications to the user interface to verify whether the input values are correct.

1.4.3. Dependency Injection

Dependency Injection (DI) is a design practice that enforces the supply of service classes to another class that depends on them. Furthermore, it claims to make code more loosely coupled, hence, making code extensible, maintainable and testable [?] [49] [53] [56]. This pattern has specific practices on injecting said services summarized below.

- **Constructor injection** - It is the simplest form of the DI pattern. Developers supply a class as a dependency to the constructor of a class that requires services or functionalities from the injected class. However, this approach may lead to a typical code smell where developers inject multiple dependencies into a given class [50] [56]. Thus, property injection arises as a potential trade-off solution when injecting more than three or four dependencies.
- **Property injection** - It is similar to constructor injection. However, this approach injects dependencies via the setter property, rather than through the constructor. The dependency is optional as the class needing its services has a local default, allowing users to inject different dependencies. This approach also presents some weaknesses, such as obscuring the internal structure of the classes implementing this DI variant or repeatedly injecting the same dependencies in different parts of the program, complicating debugging tasks. Therefore, software engineers such as Bent Kent recommend constructor over property injection to enable programmers to see the dependencies during instantiation and avoid object instantiation without passing dependencies either [10].
- **Method and Interface injection** - Both forms perform DI by passing a dependency as the argument to a method. These approaches provide dynamic dependencies during execution time, and in the case of interface injection, it enforces the class implementing its interface to supply the dependency.

Overall, proponents of DI patterns defend that they enable developers to produce more

loosely-coupled code, which increases the controllability and observability of test cases, enhances software extensibility and reusability, and ultimately affecting positively software quality attributes such as maintainability and testability.

1.4.4. Mock

Due to the remarkable weight unit testing has in this Thesis work, it is worth focusing on *mocks*. Mocks allow programmers to use mock objects as fake components or services instead of the real ones to ensure the correct functioning of unit tests. This practice reinforces the fundamental principle of unit tests, which is testing the smallest unit in isolation. However, real software applications exhibit classes that communicate with other components and services, precluding isolated method testing and breaking the original goal of unit testing. Thus, mocking said components and services allows developers to decouple external dependencies and execute unit tests in isolation, facilitating the construction and execution of a system's test suite.
Moreover, mock is a general term encompassing a family of similar implementations to replace real external resources during unit testing [11]. There are other similar terms, such as dummy, stub, fake, and mock, causing confusion for developers and readers due to their vague differences. Therefore, M. Veng discerns them into the stub group, including dummy, fake, and stub, and the mock group, including itself [73]. To clarify this separation, he argues that stubs are stand-in resources providing only the necessary data, while mocks extend this concept with object behavior. Hence, developers use stubs to create state verification tests and mocks to build tests that verify method calls, calling frequency, and calling order. Thereby, Mackinnon et al. [47] concluded that the mock pattern includes the following five steps:

1. Create mock object
2. Set up mock object state
3. Set up mock object expectation
4. Supply mock object to domain code
5. Verify behavior on the mock object

Notice that this pattern highlights how steps one through four are common to both stubs and mocks, while step five only applies to mocks as it includes behavior verification.

Ultimately, mocks provide developers with numerous valuable benefits, such as fast execution of unit tests due to external resource decoupling, error reproduction, unit test localization, ensured controllability and observability, and consistent predictability [11] [47].

1.5. Motivations

The fundamental motivation for our contribution resides in the lack of consensus for architecting large-scale Flutter applications observed in ??. The team behind this technology does not provide an official approach for architecting a Flutter App, leaving these decisions up to the implementers. Therefore, the powerful flexibility offered by Flutter, along with the absence of a widely agreed solution for a clean architecture, leads to significant problems when maintaining, scaling, and testing a codebase. Thus, we identified the need for a standardized, systematic, and consistent approach to architecting Flutter apps focused on separation of concerns and modularization, state management, and testability.

Furthermore, Dart's built-in modularization capabilities allow the extraction of functionality into separate packages. We encourage taking full advantage of this feature to compartmentalize an application into libraries having distinct responsibilities. This approach, along with proper dependency-injection techniques, enables the creation of easily-testable dedicated packages for clients, services, repositories, and plugins. Moreover, we propose an architecture built upon hybrid modularization combining layered and featured-oriented architectures, overcoming team distribution and inter-dependable features shortcomings.

Given its critical implications when developing applications, particularly at a large scale, we paid significant attention to state management. Hence, we chose the BLoC architectural pattern and implemented it with the bloc library packages created by Felix Angelov. Among other benefits, this implementation provided our application with a clear separation of concerns for business logic, explicit dependency-injection mechanisms, and robust testing capabilities. We believe that leveraging the BLoC pattern offers applications a coherent architecture for state management while keeping state changes predictable.

Lastly, we consider testability the cornerstone of software development in general and even more so in the context of a large-scale application. Thereby, we advocate for hundred percent code coverage through unit and widget testing to ensure the project's maintainability and scalability. Thus, incorporating testing during the development process is a critical practice that avoids low-quality applications and minimizes the appearance of bugs at a production stage.

2 | Implementation

This chapter introduces the core aspects of the large-scale Flutter App implemented during this thesis endeavor. This implementation serves as a tangible and practical solution built on top of the theoretical concepts and motivations presented in chapter 1 and the weaknesses found therein. Hence, we firstly provide the reader with a section covering the description and details of the application and its development, to then introduce the core aspects of the proposed application, such as the architectural patterns underpinning said implementation, the techniques and code arrangements applied to incorporate an effective state management system, and the testing approaches leveraged throughout the application.

2.1. Context

This section reviews the most relevant aspects of the implemented large-scale Flutter application, allowing the reader to learn essential knowledge which shall help them understand the upcoming content.

The development of the proposed application responds to a domotics company's need to have an accessible, capable, and intuitive system to manage a given smart home and the devices inside. A **smart home** is a home setup where a networked device enables users to remotely and automatically control internet-enabled appliances and devices. The company's goal is to implement this system in a fully automated home, featuring home security, attribute control and monitoring, access control, and alarm systems. Furthermore, the desired system is a cross-platform (Android and iOS), cross-device mobile application built with Flutter enabling users to interact with a given smart home, its devices, and data. Notice these devices belong to the realm of the Internet of Things, a system of interrelated computing devices uniquely identified facilitating data transfer over a network without requiring human-to-human or human-to-computer interaction. It is worth noting that the company requesting this system had already developed a Flutter App aiming at fulfilling the previously mentioned desired functionality. However, this application's maintainability was below any acceptable standards as any of its functionality included proper testing, and its codebase lacked consistency on critical aspects such as

architectural patterns, state management, and separation of concerns. Therefore, scaling this application required expensive and time-consuming code refactorings, which led to building the whole application from scratch.

2.1.1. Technical aspects

We now navigate the reader through the most relevant technical aspects featured in the proposed large-scale Flutter application. Notice that the upcoming content does not cover every single small technical detail as it would require this section to include trivial and uninteresting information that does not bring any value to the reader. Thus, it focuses on features, components, and services that highlight the application's complexity and size while giving a holistic vision of its functionality.
Firstly, it is worth mentioning that the system's infrastructure uses AWS Amplify to provide essential capabilities to the front end (Flutter App) and back end (in-home system not reviewed in this thesis) such as:

- **Authentication**[1] - APIs and building blocks for developers who want to create user authentication experiences.
- **Analytics**[2] - Easily collect analytics data for a given app. Analytics data includes user sessions and other custom events worth tracking in a given app.
- **API**[3] - Provides a simple solution when making HTTP requests. It provides an automatic, lightweight signing process that complies with AWS Signature Version 4.
- **GraphQL Client**[4] - Interact with a given GraphQL server or AWS AppSync API with an easy-to-use & configured GraphQL client.
- **Storage**[5] - Provides a simple mechanism for managing user content for your app in public, protected, or private storage buckets.
- **Push Notifications**[6] - Allow developers to integrate push notifications in your app with Amazon Pinpoint targeting and campaign management support.
- **Interactions**[7] - Create conversational bots powered by deep learning technologies.
- **PubSub**[8] - Provides connectivity with cloud-based message-oriented middleware.
- **Internationalization**[9] - A lightweight internationalization solution.

[1] https://aws.github.io/aws-amplify/media/authentication_guide
[2] https://aws.github.io/aws-amplify/media/analytics_guide
[3] https://aws.github.io/aws-amplify/media/api_guide
[4] https://aws.github.io/aws-amplify/media/api_guide#configuration-for-graphql-server
[5] https://aws.github.io/aws-amplify/media/storage_guide
[6] https://aws.github.io/aws-amplify/media/push_notifications_setup
[7] https://aws.github.io/aws-amplify/media/interactions_guide
[8] https://aws.github.io/aws-amplify/media/pub_sub_guide
[9] https://aws.github.io/aws-amplify/media/i18n_guide

- **Cache**[10] - Provides a generic LRU cache for JavaScript developers to store data with priority and expiration settings.
- **Predictions**[11] - Provides a solution for using AI and ML cloud services to enhance your application.

More specifically, the proposed large-scale application leverages the following packages:

- **amplify_flutter**[12] - The top-level module for Amplify Flutter.
- **amplify_api**[13] - The Amplify Flutter API category plugin.
- **amplify_auth_cognito**[14] - The Amplify Flutter Auth category plugin using the AWS Cognito provider.
- **amplify_storage_s3**[15] - The Amplify Flutter Storage category plugin using the AWS S3 provider.

Let us now discuss the application's core functionality. The App provides users with authentication functionality such as sign-up, sign-in, sign-out, and unregistering. Signed-up users can create one or many homes, which behave as virtual representations of a given physical smart home. Notice that these homes include multiple attributes that allow users to customize and modify specific details for a particular home, such as its address, apartment and unit number, type of home, size, internal systems (cooling, heating), number of people, etc. More importantly, the application allows users to switch from one home to another and access their specific IoT devices and data. Regarding IoT devices, the App includes virtual representations of any given physical device registered to a particular home. The application enables users to connect to a physical Hub, a specific type of device providing access to the rest of IoT devices, manually, via WiFi, or leveraging native software such as Apple HomeKit[16] for iOS users or Easy Connect[17] for Android users. Notice that the App integrates proper permission handling that allows access to this functionality and services provided by native low-level components and software. As far as the other IoT devices, the application features device-specific flows to pair any supported device to a particular Hub by leveraging Bluetooth, QR Code Scanning, or manual triggers. Furthermore, the application includes complex yet intuitive views enabling users to control paired devices via device-specific commands, which manifest the relevance of reactiveness in the proposed application. Moreover, this application leverages Dart streams to support alarm notifications which inform

[10] https://aws.github.io/aws-amplify/media/cache_guide
[11] https://aws-amplify.github.io/docs/js/predictions
[12] https://pub.dev/packages/amplify_flutter
[13] https://pub.dev/packages/amplify_api
[14] https://pub.dev/packages/amplify_auth_cognito
[15] https://pub.dev/packages/amplify_storage_s3
[16] https://www.apple.com/ios/home/
[17] https://source.android.com/devices/tech/connect/wifi-easy-connect

the users about critical activities and statuses of a given home and its devices. Notice the App leverages Dart streams across the entire application to implement different features and functionality, but alarm notifications are a notably interesting and complex case as the application may be idle or in background mode, yet the user still needs to receive such notifications. Lastly, the application also provides virtual representations of rooms to allow users to place devices in the desired location, creating a fully immersive smart-home experience through a mobile application. It is worth observing that the application enables users to add, remove, or edit any given virtual entity, providing a greater sense of ownership and customization while ensuring a sound and safe experience through meticulous and accurate error handling.

Lastly, Appendix B includes screenshots of the most relevant screens and processes included in the developed Flutter application. Notice these screenshots are not a complete collection of all the different screens implemented in the App, but rather a careful and deliberate selection of those screens highlighting the core functionality and aspects described before. Moreover, although this is the UI implemented by the thesis' author, the proposed architecture, modularization techniques, and state management solution allow for total flexibility regarding UI design since the presentation layer is decoupled from the other lower layers, as we will see in the upcoming sections. Thus, any given Flutter developer could provide a different look and feel to our proposed application, exhibiting the powerful general concepts and practices introduced in this thesis work that can be transferred to another large-scale Flutter.

2.1.2. Development Process

Furthermore, this thesis builds upon a project developed in a distributed software environment for a world-leading company in the domotics and home automation industry needing a robust large-scale Flutter application, as previously mentioned. This environment was fully remote and relied on a git[18]-based version control system featuring advanced Code Integration / Code Deployment mechanisms. Notice that the approach and architectural decisions discussed in the following sections minimized the code-related conflicts, enabling developers to distribute and develop coding tasks efficiently and effectively. Accordingly, this environment included various teams formed by back-end engineers, QA testers, designers, embedded and IoT devices engineers, project and product managers, and a front-end team with some of the world's most skilled and talented Flutter developers and engineers, including the thesis' author. It is worth noting that the thesis' author

[18] https://git-scm.com/

contributed in no small manner to developing and shaping the project by implementing complex features, enhancing the test suite for the entire application within any given layer and any given feature, participating in technical meetings and backlog grooming sessions, reviewing pull requests, reporting and fixing bugs, and researching unknown knowledge. However, he did not implement the entirety of the codebase, which resulted from a collaborative team effort. Ultimately, this project served as an ideal case study to support and motivate the proposed work, solely created by the thesis' author, including all its contents: the months-long research introduced in chapter 1, the analysis, demonstrations, and example selection illustrated in this chapter, the aggregation of the results presented in the upcoming chapter 3, the related work covered in chapter 4 and the ultimate conclusions in the final chapter 5.

2.2. Hybrid Architecture

This section provides a detailed description of the software architecture and related facets affecting the structure of our proposed project. Thus, we introduce a hybrid approach combining the layered (Subsection 2.2.1) and feature-oriented (Subsection 2.2.2) architectural patterns and emphasize the relevance of modularization through packaging in the context of Flutter applications (Subsection 2.2.3).

In the context of this thesis, we refer to hybrid architecture as the combination of layered and feature-oriented modularization. We provided an overview of what these two patterns aim to accomplish and their principal characteristics in Section 1.3. Thus, we propose an architecture that leverages the strengths of both approaches leading to:

- Better separation of concern
- Enhanced maintainability
- Enhanced testability
- Code flexibility (Add, remove, or change features easily)
- Less unused code
- Feature and module reusability
- Consistent behavior and standards across different applications

Nonetheless, this architecture is more involved than any of the other two approaches separately, and thus, it also has some drawbacks that are worth noting, such as

- Slower development
- Potential overkill for simple features
- May increase code complexity

Ultimately, the drawbacks are somewhat subjective as they depend on the ability of

the implementers, while the advantages of this approach are more tangible and directly address two of our main objectives: high maintainability and testability. Following this overview, we present the reader with the fundamental characteristics taken from each approach and applied to our implementation.

2.2.1. Layered

Edsger W. Dijkstra [19] was the first to propose the fragmentation of software programs into responsibility-based hierarchical levels, called layers, and has since become a common standard in large-scale applications [55]. Thereby, we leveraged this industry-tested approach to propose a layered architecture based on four distinct levels: User Interface Layer, Business Logic Layer, Repository Layer, and Data Layer.

- **Data Layer** - It is the lowest layer of our four-tier architecture. It is responsible for communicating with external sources (databases or APIs) to retrieve raw data. Moreover, this layer exposes clients free of any UI-specific dependencies. Lastly, we can consider this layer the *engineering* layer since it serves the purpose of efficiently processing and transforming data.
- **Repository Layer** - Decouples the business logic and data layers and composes one or more clients from the data layer to apply domain-specific business rules to the retrieved data. Domain-based repositories compose this layer whose principal role is centralizing shared data access functionality, acting as a middleman between the business logic and data layers. Furthermore, there should only be one repository per domain, which must be free of any Flutter dependencies, and it can only interact with the data layer. Lastly, we can consider this layer the *product* layer since it brings value to the user through the exposure of refined data.
- **Business Logic Layer** - Vessels the business logic of the application. Blocs and Cubits compose one or more repositories and include the logic to surface the business rules via a specific feature or use-case. Moreover, this layer employs the bloc library to manage the logic associated with each feature. Hence, we can consider this layer the *feature* layer as it determines the correct functioning of any given feature. The state management section 2.3 provides more comprehensive information about this layer and its implementation.
- **Presentation Layer** - It is the top-most layer of our four-tier architecture. It serves as the UI of the application, allowing users to interact directly with it. These interactions generate events, which are then forwarded to the business logic layer. Moreover, it reacts to state changes from the business logic layer, prompting the UI to trigger rendering modifications via Flutter Widgets. Furthermore, this layer

should only interact with the business logic layer. Lastly, we can consider this layer the *design* layer since it aims to provide the best possible experience for users through visual components and effects.

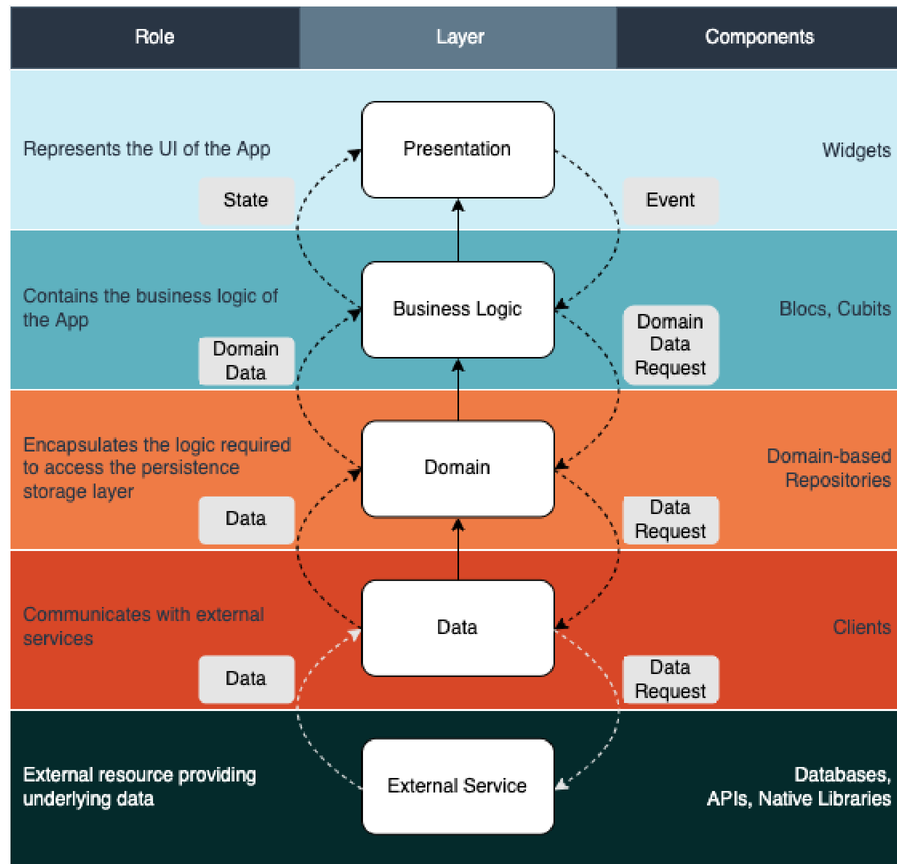

Figure 2.1: Four-tier layered architecture underpinning the implementation's structure

Notice that the external service layer, the bottom-most layer, depicted in Figure 2.1 does not depend on us, and hence, we do not account for it when describing this layered architecture. Additionally, the repository layer adheres to the essential premises of the *Repository Pattern* [34]. However, our proposed implementation does not rely on agnostic interfaces that enable its inheritance at the data layer, but we leverage client-oriented composability to craft these domain-based repositories.

2.2.2. Feature-Oriented

Before diving into the details of how our architecture implements the feature-oriented approach, it is necessary to distinguish the concept of feature by layer:

- **Infrastructure Feature** - It is a feature found in the data layer and presented as a client module that adheres to the role of this layer by communicating with external

sources and fetching data. It adds functionality at a low level within the application structure, and hence, it does not provide direct value to the application users as they perceive it as a black box.

- **Domain Feature** - It is a feature found in the domain layer and presented as a domain-based repository that adheres to the role of this layer by applying domain-specific business rules to the retrieved data. It adds functionality at a middle level within the application structure, and hence, it does not provide direct value to the application users as they still perceive it as a black box.
- **Application Feature** - It is a feature found within the business logic layer (logic component), the presentation layer (design component), or both layers (combined component) that adheres to the role of this layer by providing either functionality or visual value, or both. Notice that this feature sits at the highest level within the application structure, and hence, it exhibits direct value to the application users as they can interact with this feature.

Infrastructure Features

We find infrastructure features at the bottom-most architectural layer of the application. As previously mentioned, we use modules called clients to implement these features, which we differentiate between pure or involved clients depending on their intrinsic complexity. Notice that said complexity refers to the client's internal functionality and data manipulation. Thus, we present the reader with one example of an involved client, which is paramount to the proper functionality of the proposed application, and another example of a simple client, which serves as a basis and reference for other simple clients implemented in the application.

Firstly, we proceed to discuss our API Client. This module features the most involved functionality, data manipulation, and thus, code structure among all the implemented clients. Its essential role is to communicate with a specific API to fetch, create, and update data and subscribe to data changes. In this particular case, our application must interact with a GraphQL API[19] that accepts queries to fetch data, mutations to create and update data, and subscriptions to subscribe to data changes in the back-end. It is worth noting that the structure of this client would require minor modifications to serve the same purpose should the back-end expose a REST API instead, although the libraries and methods employed to communicate with one or the other API would change significantly since the interaction with GraphQL and REST APIs differs significantly. However, none of these changes should affect the layers above since the implementation is abstracted

[19] https://graphql.org/

from other layers and modules.

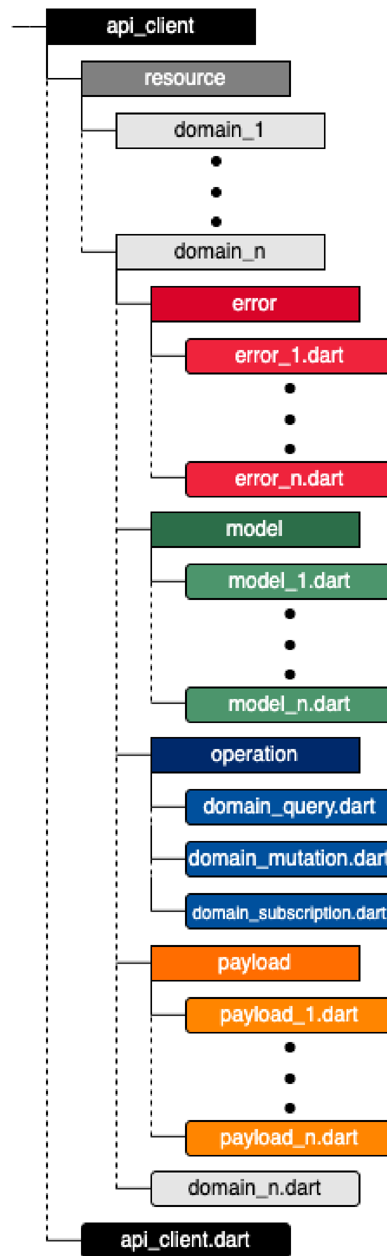

Figure 2.2: API Client structure

From Figure 2.2 above, we observe that domains influence how we structure this client module. These domains correspond to a representation of a meaningful concept or object retrieved from the API. Notice that the analysis, definition, and design of these domains are outside the scope of this thesis work as they were the responsibility of the requirements and embedded teams, and we solely needed to incorporate them into the application for further manipulation. Moreover, these domains may introduce specific errors in the system, which we need to address and handle accordingly. Thereby, the error submodule

and the different error files therein allow mapping said errors, providing effective error handling within this client, and avoiding exception leakage to upper layers. Additionally, the model submodule contains the files that allow the mapping of raw data into more structured and valuable data objects. These models need not match precisely the representation of the domain models in the back-end but rather provide a solid basis for upper layers to consume and further manipulate this data. Furthermore, the operation submodule contains files including the different operations the API Client can perform against the GraphQL API, while the payload submodule contains the different objects (payloads) returned by any of these given operations. Notice that these payloads include the raw data to be manipulated and mapped into their corresponding model and error objects. Lastly, the domain_n.dart file leverages all the submodules previously described and adds the necessary functionality to interact with the external source through dedicated class methods.

Listing 2: Api Client

```
1  /// {@template api_client}
2  /// An API client to communicate with GraphQL API
3  /// {@endtemplate}
4  class ApiClient {
5    /// {@macro api_client}
6    ApiClient({
7      required GraphQLCategory graphQLCategory,
8      required Client client,
9    })  : _accountResource = AccountResource(graphQLCategory: graphQLCategory),
10         _homeResource = HomeResource(
11           graphQLCategory: graphQLCategory,
12           http: client,
13         ),
14         _deviceResource = DeviceResource(graphQLCategory: graphQLCategory),
15         _deviceCommandResource = DeviceCommandResource(
16           graphQLCategory: graphQLCategory,
17         ),
18         _hubResource = HubResource(graphQLCategory: graphQLCategory),
19         _areaResource = AreaResource(graphQLCategory: graphQLCategory),
20         _alarmResource = AlarmResource(graphQLCategory: graphQLCategory),
21         _homeMemberResource = HomeMemberResource(
22           graphQLCategory: graphQLCategory,
23         );
24
```

```
25    final AccountResource _accountResource;
26    final HomeResource _homeResource;
27    final DeviceResource _deviceResource;
28    final DeviceCommandResource _deviceCommandResource;
29    final HubResource _hubResource;
30    final AreaResource _areaResource;
31    final AlarmResource _alarmResource;
32    final HomeMemberResource _homeMemberResource;
33
34    /// {@macro account_resource}
35    AccountResource get accountResource => _accountResource;
36
37    /// {@macro home_resource}
38    HomeResource get homeResource => _homeResource;
39
40    /// {@macro device_resource}
41    DeviceResource get deviceResource => _deviceResource;
42
43    /// {@macro device_command_resource}
44    DeviceCommandResource get deviceCommandResource => _deviceCommandResource;
45
46    /// {@macro hub_resource}
47    HubResource get hubResource => _hubResource;
48
49    /// {@macro area_resource}
50    AreaResource get areaResource => _areaResource;
51
52    /// {@macro alarm_resource}
53    AlarmResource get alarmResource => _alarmResource;
54
55    /// {@macro home_member_resource}
56    HomeMemberResource get homeMemberResource => _homeMemberResource;
57  }
```

The code presented above corresponds to the api_client.dart file. This file exposes every domain as a dedicated resource, allowing the repository layer to leverage all their functionality.

Regarding the pure client example, we provide the reader with the structure and code implementation of the Permission Client in Figure 2.3. Notice that the model submodule is optional and depends on the data needs of a given client. Lastly, the code snippet

22 shown in the Appendix chapter corresponds to the permission_client.dart file and includes all the permission requests required by the Flutter implementation to access specific native resources.

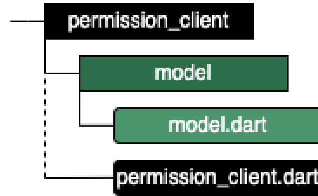

Figure 2.3: Permission Client structure

Domain Features

We find domain features within the domain-architectural layer of the application. As previously stated, we use modules known as repositories to implement these features. Much like the clients in the data layer, repositories do not provide direct value to application users. However, they represent a fundamental piece of the proposed software architecture, insulating the application from changes in the persistence store (service layer) and facilitating automated unit testing, as we will see in section 2.4. Furthermore, we created a component that allows emitting events from a Future response or another Stream of events without blocking the data flow. We called this component *AsyncBehaviorSubject*, enabling repositories to become reactive components that provide the business logic layer with continuous data streams. We consider this feature a novel approach that allows us to implement reactive principles leveraged by the blocs at the business logic layer instead of implementing a typical imperative behavior. Thus, we present the reader with the considerably straightforward structures of said *AsyncBehaviorSubject* and repositories, and a code snippet for the *AreaRepository*, while the code snippet for *AsyncBehaviorSubject* is listed under the List of Source Codes 2.4.

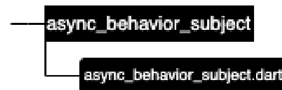

Figure 2.4: AsyncBehaviorSubject structure

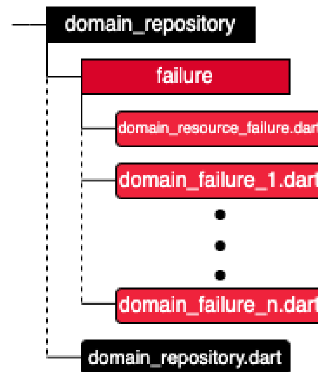

Figure 2.5: Repository structure

Listing 3: Area Repository

```
/// A [Map] of [Area]s by their [Area.id]
typedef AreaMapById = Map<String, Area>;

/// {@template area_repository}
/// Repository to manage information about [Area]s.
/// {@endtemplate}
class AreaRepository {
  /// {@macro area_repository}
  AreaRepository(this._apiClient, this._homeId);

  final ApiClient _apiClient;
  final String _homeId;

  late final AsyncBehaviorSubject<AreaMapById> _areasController =
      AsyncBehaviorSubject<AreaMapById>(
    onStart: () {
      _areasController.addFuture(_getAreas(_homeId));
    },
  );

  /// Stream the list of areas
  Stream<AreaMapById> get areas => _areasController.stream;

  Future<AreaMapById> _getAreas(String homeId) async {
    try {
      final areas = await _apiClient.areaResource.getAreas(homeId);
      return {for (final area in areas) area.id: area};
```

```
28      } catch (e) {
29        throw AreaResourceFailure(e);
30      }
31    }
32
33    // Specific domain methods below
34  }
```

Notice we did not include the full implementation of the Area Repository as the file contains numerous methods to interact with the GraphQL API through the Area Resource. Nonetheless, we covered the initial constructor, the properties initialization, and the primary use of the AsyncBehaviorSubject within a domain repository. Lastly, it is worth observing that the structure of a domain repository includes a failure submodule which allows to classify and handle errors produced within a given domain. More importantly, since these errors should arise from the exceptions thrown at the data layer, the domain_resource_failure.dart file handles resource-related errors, including a list of specific errors as parts of this class.

Listing 4: Area Resource Failure

```
1  part 'add_device_failure.dart';
2  part 'delete_area_failure.dart';
3  part 'create_area_failure.dart';
4  part 'update_area_failure.dart';
5  part 'area_remove_hub_failure.dart';
6  part 'area_add_hub_failure.dart';
7  part 'area_remove_device_failure.dart';
8
9  /// Failure thrown if any [AreaResource] method fails.
10 class AreaResourceFailure implements Exception {
11   /// Default constructor
12   AreaResourceFailure(this.originalException);
13
14   /// Original [Exception]
15   final Object? originalException;
16
17   @override
18   String toString() => 'AreaResourceFailure - $originalException';
19 }
```

Application Features

We find applications features at the top-most architectural layer of the application. As previously described, these features add either functional or visual value, or both, directly to the application user. Thus, we introduce the reader with three different examples of a feature solely adding visual value, another one solely adding functional value, and lastly, one providing both visual and functional value to the end-user.

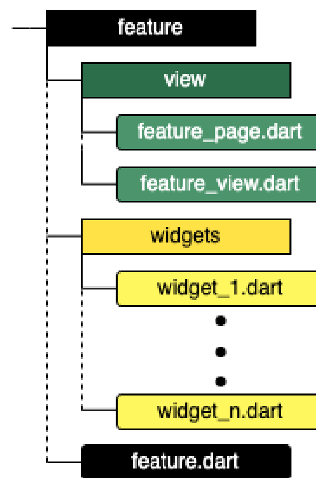

Figure 2.6: Visual application feature structure

Listing 5: Welcome Page

```
1  class WelcomePage extends StatelessWidget {
2    const WelcomePage({Key? key}) : super(key: key);
3
4    @override
5    Widget build(BuildContext context) {
6      return const Scaffold(
7        extendBodyBehindAppBar: true,
8        appBar: CustomAppBar(
9          automaticallyImplyLeading: false,
10       ),
11       body: SafeArea(
12         child: WelcomeView(),
13       ),
14     );
15   }
16 }
```

The code presented above shows the code employed to create a visual feature. The main

characteristic of this type of feature is that it only uses Flutter Widgets to provide its value, and hence, it does not require state management or additional complex functionality. In this particular case, we observe the top-most widget of the feature, WelcomePage, which provides the initial Scaffold and body structure for nested Widgets down the Widget Tree to complete the visual feature. As shown in Figure 2.6, developers can implement as many Widgets as they may need to implement a visual feature.

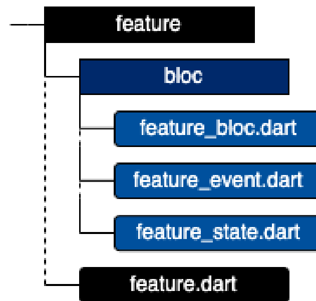

Figure 2.7: Visual application feature structure

Listing 6: Home Bloc

```
part 'home_event.dart';
part 'home_state.dart';

class HomeBloc extends Bloc<HomeEvent, HomeState> {
  HomeBloc({required this.homeId, required this.homeRepository})
      : super(HomeLoading()) {
    _sub = homeRepository.homes.listen((homes) {
      final home = homes.values.where((element) => element.id == homeId);
      if (home.isNotEmpty) {
        add(HomeUpdated(home.first));
      } else {
        add(const HomeNotFound());
      }
    });

    on<HomeUpdated>(_onHomeUpdated);
    on<HomeNotFound>(_onHomeNotFound);
  }

  final HomeRepository homeRepository;
  final String homeId;
  late StreamSubscription<Map<String, Home>> _sub;
```

```
23
24    FutureOr<void> _onHomeUpdated(HomeUpdated event, Emitter<HomeState> emit) {
25      emit(
26        HomeData(
27          address: event.home.address,
28          unitNumber: event.home.unitNumber,
29          homeId: event.home.id,
30        ),
31      );
32    }
33
34    FutureOr<void> _onHomeNotFound(HomeNotFound event, Emitter<HomeState> emit) {
35      emit(HomeMissing(homeId));
36    }
37
38    @override
39    Future<void> close() {
40      _sub.cancel();
41      return super.close();
42    }
43  }
```

Notice that the functional feature shown in Figure 2.7 and the code above corresponds to a bloc handling the events and state performed on a Home, one of the domains included in the application. Furthermore, this type of feature usually manages the business logic above other nested features, either visual or functional, or both. Notice it is a standalone component, and hence, we can reuse or leverage it in other parts of the application, ensuring that the behavior of this functional feature stays consistent across the application. Lastly, we provide further insights on functional features in section 2.3, presenting the reader with a comprehensive overview of state management patterns and components.

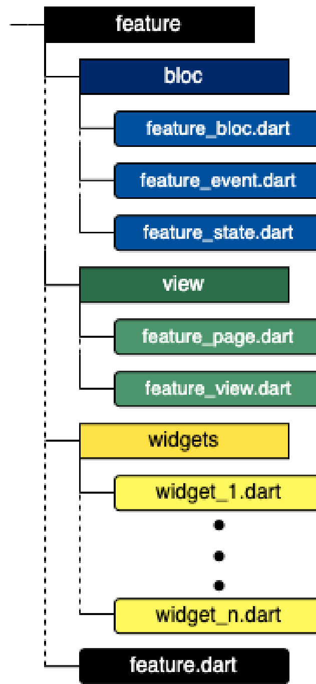

Figure 2.8: Combined application feature structure

Figure 2.8 above, the *HomeDetailsBloc*, and *HomeDetailsPage* displayed in code listings 24 and 25, respectively, show a feature's structure and implementation combining both visual and functional values. More specifically, this feature provides the end-user with a Home Details Page, including a nested view and Widgets, which leverages a HomeBloc and a HomeDetailsBloc to handle user interactions and state changes throughout this feature. Furthermore, this feature includes both a bloc and view subcomponents, enabling more complex features. Ultimately, these are the most widely used features within the application as they allow us to enhance the presentation layer with the state changes handled by blocs at the business logic layer. Once again, we take a more in-depth look at these features in section 2.3.

2.2.3. Packaging

In addition to the advantages provided by combining the layered and feature-oriented architectural patterns, we consider fundamental complementing our architecture with an effective modularization approach. This approach leverages the full power of dart packages reviewed in Section 1.2.4, enabling the compartmentalization of features by layer and feature. We propose a project structure that adheres to the **Multimodule Monorepo** [46] [52], an approach that allows maintaining a project as a single repository with multiple submodules for each of the features included in each layer. Furthermore, the taxonomy of this enhanced architecture introduces two higher grouping layers: lib and packages.

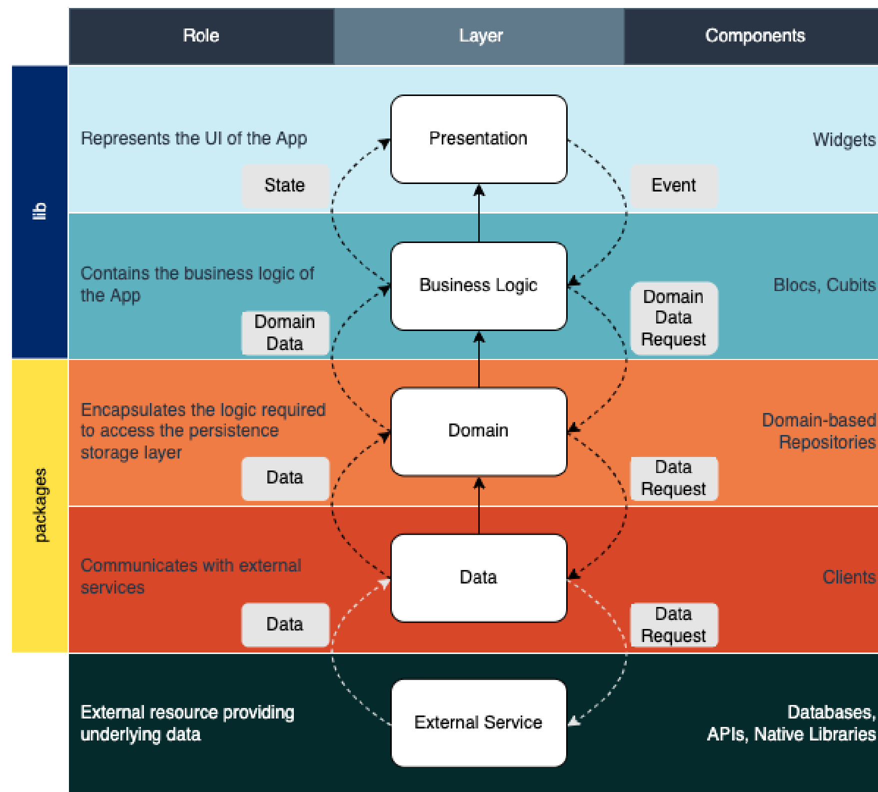

Figure 2.9: Hybrid Architecture with multimodule monorepo

As shown in Figure 2.9, the lib layer is effectively a directory encompassing the two higher layers of our architecture, presentation and business logic, and the application features therein. Moreover, we find another encompassing layer called packages, whose principal function is to group the repositories and clients within the domain and service layers under a single directory. Ultimately, we can group all our application features as submodules under the lib directory while repositories and clients become standalone packages listed under the packages directory. Notice this approach provides our project with several benefits, such as:

- **Discoverability** - Access to all packages and codes from the IDE view
- **Separation of concern** - Each package has a single purpose and becomes a composing part of a given layer
- **Testability** and composition: Each layer has distinct rules enabling independent and isolated testing by layer
- **Reusability**: Different projects may share the same package keeping functionality consistent across applications
- **Clarity**: Explicit understanding of the project's dependency graph (tools like pub-

viz[20] facilitate its visualization)

Moreover, we suggest creating the application project and the leveraged internal packages using **very_good_cli**[21]. This Dart package provides an enhanced Command Line Interface, which uses the **Very Good Core**[22] template by default, adhering any project and package to the industry-tested best practices implemented during the development of our proposed Flutter Application [74].

In the end, following these architectural patterns and decisions enables engineers to easily add, modify, remove, and test features and address bugs and fixes without affecting other project mates' work, enhancing the overall maintainability and scalability of a given software application. Refer to Figure 2.10 below for an explicit, though simplified, visualization of the proposed project's directory structure.

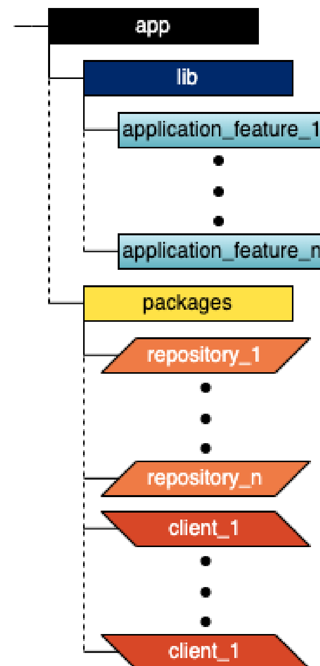

Figure 2.10: Simplified project directory structure

[20] https://pub.dev/packages/pubviz
[21] https://pub.dev/packages/very_good_cli
[22] https://github.com/VeryGoodOpenSource/very_good_cli/blob/main/doc/very_good_core.md

2.3. State Management

State management is arguably one of the most controversial, debated-about, and critical decisions software architects and engineers must make when implementing any software App. It is also one of the earliest decisions, thoroughly influencing the implementation, particularly the application features found at the higher architectural layers. Thereby, choosing the *right* state management solution is a decision that incurs weighing the advantages and disadvantages provided by the analyzed options, accounting for the non-trivial problems that shall arise should the chosen solution be changed during the development process [31]. Thus, this chapter introduces the reader to the selection criteria employed to select **BLoC** and **flutter_bloc** as our state management solution and then provides an ample and comprehensive demonstration about implementing this pattern in our proposed large-scale application.

2.3.1. Selection Criteria

Our selection criteria do not intend to become the ultimate model to select a state management solution in any given circumstances but rather allow any engineer seeking to build a maintainable, scalable, and testable Flutter application to find the most suitable option. Hence, the chosen state management solution must be predictable, simple, and highly testable and increase the engineers' comfortability and confidence towards building and maintaining a robust product [17]. Thus, we chose the BLoC pattern and its Flutter implementation through the flutter_bloc package, both reviewed in Section 1.1.2.

"The best state management solution is the one that works the best for you."

— **Jorge Coca**, Why we use flutter_bloc for state management [17]

Predictability

Engineers often face significant challenges when accurately determining the state of the developed application at any given point in time. As we reviewed in Section [26], Flutter introduces the Widget Tree to address this challenging ambiguity, allowing developers to modify the state of the Widgets, and hence, the Widget Tree's structure. However, managing widget states vertically and horizontally across a complex Widget Tree is a tortuous and intricate endeavor. Therefore, flutter_bloc allows developers to decompose the application's state into smaller, well-defined, and deterministic state machines to address this complexity and ambiguity. Ultimately, these state machines transform events into zero, one, or multiple predictable states.

Simplicity

Notice Apps are reactive by nature, and thus, developers must program them to be reactive. However, this natural *reactiveness* is the cause of non-deterministic user interactions that may occur at any point in time, if at all. Therefore, developers traditionally relied on powerful yet complex APIs to manage said reactiveness and produce interactive and engaging applications. On the other hand, flutter_bloc proposes a simplified API that abstracts the complexity of streams while still honoring the natural reactiveness of applications. Thereby, developers need not maintain non-trivial stream subscriptions and lifecycles, allowing them to focus on predictable interactions by handling incoming events and outputting new states.

High testability

As previously stated numerous times throughout this Thesis work, we consider testability is a crucial feature of any high-quality software application. Furthermore, we seek and deliver one hundred percent coverage through unit testing, as shown in section 1.4, to boost developers' confidence in delivering reliable and quality products. Moreover, the flutter_library makes code testing one of its principal values and provides a dedicated package for such a purpose, **bloc_test**. This package is a utility library that eliminates the complexity of testing reactive code while enabling developers to unit test their code and validate the product behavior at any point in time with barely any setup required.

However, we had to address and clarify some concerns about flutter_bloc being too complex requiring too much boilerplate. Regarding the package's complexity, it is necessary to highlight that Flutter already provides a capable solution to manage state: *setState* on *StatefulWidgets*. However, this simplistic approach can rapidly lead to state mismanagement, and at scale, it may suffer from known problems such as prop drilling [29]. As for the required boilerplate, we consider benefits introduced by flutter_bloc such as truly immutable and independent events and states outweigh having fewer lines of code. Thus, the package's complexity and required boilerplate remain justified and do not suggest that any other analyzed solution should become more suitable for our purposes.

2.3.2. Code Samples

This subsection offers comprehensive insights into the state management components implementation by introducing the reader to representative use cases extracted from the

developed large-scale Flutter application. As discussed previously, the flutter_bloc package provides these state management components as Blocs and Cubits. Observe that we made a limited selection from all the business logic components implemented across the application. Nonetheless, this selection presents and covers the most relevant scenarios faced when designing and implementing Blocs and Cubits while focusing on general and extrapolatable concepts rather than on the specifics of any particular feature. Ultimately, the approaches exemplified in the following examples allowed developers to exclusively use **StatelessWidgets** in favor of **StatefulWidgets** regardless of the implemented feature, except in the case of UIs including animations, effectively avoiding the use of *setState*.

Code Sample I

This first code sample presents a straightforward Bloc, allowing the introduction of more complex examples building on top of the concepts learned from this initial use case. These types of blocs represent a functional feature and favor reusability by other design features. Hence, we start by analyzing the events shaping the behavior of the proposed bloc.

Listing 7: Delete Device Event

```
1  partof 'delete_device_bloc.dart';
2
3  abstract class DeleteDeviceEvent extends Equatable {
4    const DeleteDeviceEvent();
5
6    @override
7    List<Object> get props => [];
8  }
9
10 class DeleteDeviceRequested extends DeleteDeviceEvent {}
```

Notice how we start by defining an abstract class, *DeleteDeviceEvent*, which acts as a base class for the event class defined afterward, *DeleteDeviceRequested*. *DeleteDeviceRequested* is immutable by default and allows triggering state changes by prompting the bloc to react to a specific interaction. Lastly, even though this example only includes one event class that takes no parameters, we may have multiple events accepting any number of parameters, as we will see in the following more-involved examples. Moreover, we proceed to analyze the state of the proposed bloc.

Listing 8: Delete Device State

```
part of 'delete_device_bloc.dart';

class DeleteDeviceState extends Equatable {
  const DeleteDeviceState({
    this.status = DeleteDeviceStatus.initial,
  });

  final DeleteDeviceStatus status;

  @override
  List<Object?> get props => [status];

  DeleteDeviceState copyWith({
    Device? device,
    DeleteDeviceStatus? status,
  }) =>
      DeleteDeviceState(
        status: status ?? this.status,
      );
}

enum DeleteDeviceStatus {
  initial,
  updated,
  loading,
  success,
  failed,
}
```

The state file includes a single immutable and instantiable class, *DeleteDeviceState*. It is worth mentioning that other implementations and even the official flutter_bloc documentation declare the main Bloc state class as an abstract and then include multiple state classes to represent the different states a given feature may exhibit. However, we introduce a cleaner and leaner approach that provides enhanced flexibility when leveraging this state within the presentation layer without jeopardizing the immutability principles. This approach relies on a status **Enum** encompassing the different states otherwise declared as immutable classes and a **copyWith** method that returns a new instance of the state class.

We can employ this approach thanks to the **Equatable**[23] class that allows comparing two objects, including their properties, and thus states too, to determine whether they are equal or not. Notice how the overridden props getter includes the status property, which effectively allows differentiating between states by checking the value of this property. Ultimately, we broadly use this pattern in the application, keeping bloc state design and implementation consistent across features. Moreover, we proceed to analyze the proposed bloc.

Listing 9: Delete Device Bloc

```
1  part 'delete_device_event.dart';
2  part 'delete_device_state.dart';
3
4  class DeleteDeviceBloc extends
5  Bloc<DeleteDeviceEvent, DeleteDeviceState> {
6    DeleteDeviceBloc({
7      required DeviceRepository deviceRepository,
8      required String deviceId,
9    })  : _deviceId = deviceId,
10         _deviceRepository = deviceRepository,
11         super(const DeleteDeviceState()) {
12     on<DeleteDeviceRequested>(_deleteDeviceRequested);
13   }
14
15   final String _deviceId;
16   final DeviceRepository _deviceRepository;
17
18   FutureOr<void> _deleteDeviceRequested(
19     DeleteDeviceRequested event,
20     Emitter<DeleteDeviceState> emit,
21   ) async {
22     try {
23       emit(state.copyWith(
24           status: DeleteDeviceStatus.loading,
25         ),
26       );
27       await _deviceRepository.unregisterDevice(
28         deviceId: _deviceId,
29       );
```

[23] https://pub.dev/packages/equatable

```
30         emit(state.copyWith(
31               status: DeleteDeviceStatus.success,
32             ),
33           );
34       } catch (e) {
35         addError(e);
36         emit(state.copyWith(
37             status: DeleteDeviceStatus.failed,
38           ),
39         );
40       }
41     }
42   }
```

The bloc described in the code above, *DeleteDeviceBloc*, extends the base class **Bloc<Event, Class>**, having *DeleteDeviceEvent* and *DeleteDeviceState* as the Event and State types, respectively. Notice we must inject all the dependencies leveraged by the bloc as parameters. Thus, employing this DI approach enables developers to thoroughly test blocs, as we will see in subsection 2.4.3. In this case, the described bloc requires an instance of a *DeviceRepository* and a parameter of type String. Furthermore, the **on**<*DeleteDeviceRequested*> function enables the registration of an event handler for an event of type *DeleteDeviceRequested*. Additionally, the flutter_bloc documentation indicates "there should only ever be one event handler per event type E," ensuring event handling in a systematic and deterministic manner. Moreover, the described bloc includes a private method to handle the reaction to the *DeleteDeviceRequested* event. This private method leverages an Emitter of the same type as the bloc state, *DeleteDeviceState*, to emit state changes exploited by the presentation layer. Notice how we employ the previously described copyWith method to emit a new state that includes the corresponding status. Lastly, since the private method includes an asynchronous operation, we must use the **async-await** keywords to ensure a correct state sequence and surround it in a **try-catch** block to prevent unhandled exceptions, including an **addError** function that allows logging caught errors.

Code Sample II

We now review a more involved use case that refers to a feature combining design and functional value. Thus, we aim to demonstrate how these two parts are integrated and walk the reader through the thinking process and implications behind the code referenced in the List of Source Codes . Much like we did in the previous code sample, we will start

by analyzing the proposed bloc and its different parts, to then move onto the feature's View and how the state affects the widgets therein.
From the code snippet 26, we observe multiple immutable classes extending the base class *EditDeviceEvent*. These immutable classes correspond to a particular event that the bloc will handle independently. Moreover, this example emphasizes the importance of the Equatable class, rendering each immutable event class and its properties independent, thus, enabling value comparison. Let us now analyze the state of the proposed bloc.
The code presented in 27 corresponds to *EditDeviceState*, the state shaping the behavior of the proposed bloc. Notice how this state's strategy and structure mirrors the state presented in the previous example by using a single instantiable state class that includes a *copyWith* method and relies on a particular enum to determine the state's status. The main differences from the previous simpler example are the number of properties included in this state and the use of **boolean getters**. The latter is a common practice used throughout the application to simplify the condition statements in the presentation layer. Moreover, we proceed to analyze the proposed.
Regarding the proposed bloc 28, *EditDeviceBloc*, we observe that, once again, it features evident similarities to the bloc presented previously. However, this bloc features more parameters required by the bloc constructor and events. Regarding the parameters, it stands out the use of two repositories, *DeviceRepository* and *AreaDepository*, which shall allow the bloc to interact with the domain layer. As for the events, we have omitted the specific code of each of the listed private methods for brevity and generality shake. Nonetheless, notice the one-to-one correlation between the number of events extending the *EditDeviceEvent* base class and the number of private bloc methods that include the business logic to handle each event. Ultimately, this correlation indicates the need for a bloc to manage each event independently and on a case-by-case basis. Let us now review the presentation components leveraging the analyzed bloc.

Listing 10: Edit Device Page

```
1  class EditDevicePage extends StatelessWidget {
2    const EditDevicePage({Key? key, required this.device})
3      : super(key: key);
4
5    final Device device;
6
7    static const routeName = '/edit-device';
8
9    static Route<String> route({required Device device}) {
10     return MaterialPageRoute<String>(
```

```
11        settings: const RouteSettings(name: routeName),
12        builder: (context) =>
13          EditDevicePage(device: device),
14      );
15    }
16
17    @override
18    Widget build(BuildContext context) {
19      return BlocProvider(
20        create: (context) => EditDeviceBloc(
21          homeId: context.read<HomeBloc>().homeId,
22          device: device,
23          deviceRepository: context.read<DeviceRepository>(),
24          areaRepository: context.read<AreaRepository>(),
25        )..add(EditDeviceAreaRequested()),
26        child: const EditDeviceView(),
27      );
28    }
29  }
```

Firstly, we need to address the technique employed to inject the bloc within the presentation layer so that widgets further down the Widget Tree have access to it. We call this technique the **Page-View pattern** and it prevents developers from facing a common issue where the code *calls a bloc that is not within the context it is being used*. Thus, the Page-View pattern enables developers to inject a given bloc at the top of a feature's Widget Tree allowing the **View** Widget and its nested Widgets to use said bloc through the **BuildContext**. Thereby, we inject the *EditDeviceBloc* via the **BlocProvider** and pass as a child Widget the *EditDeviceView*. It is worth noting that we can pass the necessary dependencies, such as repository instances or data from other blocs' states, to the bloc by leveraging **context.read<T>**. This data access and DI mechanism are deeply influenced by how we inject dependencies into the application structure (Widget Tree), further reviewed in the Testing section. Furthermore, observe how we **add** an event, *EditDeviceAreaRequested*, as soon as we instantiate the bloc, allowing the *EditDeviceView* to access the initial necessary data right away.

Let us now take a look at the *EditDeviceView* Widget presented in source code 29. The *EditDeviceView* Widget's purpose is to provide a Scaffold for rendering nested widgets, and more importantly, listen to state changes to show notification widgets and perform navigation actions. Regarding the listening aspect, widgets may employ a **BlocListener<Bloc, State>**, which guarantees to invoke the listener in response to a state

change only once. Thereby, *EditDeviceView* implements a *BlocListener<EditDeviceBloc, EditDeviceState>* to listen to state changes from the *EditDeviceBloc* and react to such changes within the scope of the function passed to its listener parameter, leaving its child widget unaffected by re-rendering actions. Additionally, the parameter **listenWhen** facilitates more granular control over the implemented listener by comparing the previous and current states, which enhances the efficiency of the widget by avoiding unnecessary **listener** calls. Lastly, let us analyze the structure of *_DeviceNameTextField*, one of the nested widgets within the *EditDeviceView*.

The nested widget introduced in code snippet 30, *_DeviceNameTextField*, directly accesses the *EditDeviceBloc* through the *BuildContext* object. Furthermore, it leverages the **select** function to retrieve part of the state and react to changes only when the selected part changes, enhancing the Widget's efficiency by avoiding unnecessary rebuilds. Lastly, we can also notify the bloc about occurred events by, once again, accessing the bloc instance via the BuildContext and adding the desired event, *EditDeviceNameUpdated(name)*, in this case. Notice *_DeviceNameTextField* is a text field allowing user input and displaying back such input value as the result of state changes.

Code Sample III

This third code sample navigates the user through another scenario where a given bloc helps to control the state of a particular view and the rendered contents or Widgets. Having already thoroughly reviewed two different bloc use cases and their code implementation, we now focus solely on the implementation and effects of the proposed bloc within the presentation layer to avoid redundant code and explanations.

Listing 11: Plug Pairing Body

```
1  class PlugPairingBody extends StatelessWidget {
2    const PlugPairingBody({Key? key}) : super(key: key);
3
4    @override
5    Widget build(BuildContext context) {
6      return BlocProvider(
7        create: (context) => PlugPairingBloc(
8          hubRepository: context.read<HubRepository>(),
9          deviceRepository: context.read<DeviceRepository>(),
10       )..add(const PlugPairingCableChecked()),
11       child: const PlugPairingBodySwitcher(),
12     );
13   }
```

```
14 }
```

The *PlugPairingBody* Widget presented in code above adheres to the Page-View pattern previously mentioned. Notice this pattern does not enforce a fixed naming convention as the Page and View terms are concepts rather than specific types of widgets. Therefore, in this case, *PlugPairingBody* corresponds to the Page concept, and *PlugPairingBodySwitcher* corresponds to the View concept. Thus, we inject the *PlugPairingBloc* with its required dependencies using the **create** function parameter exposed by the *BlocProvider* and pass *PlugPairingBodySwitcher* as a **child** having access to the injected bloc. Let us now look further into this child widget shown in code snippet 31.

PlugPairingBodySwitcher serves as a general example of a widget that allows developers to address a typical situation where the contents of a View must dynamically change based on a given state. More specifically, we are not interested in rerendering some of the displayed widgets but rather in changing the entire Widget Tree contents from below a specific point, effectively rendering a new View. Moreover, these kinds of views often require developers to handle showing notifications in the form of dialogs or snack bars and navigation control. Thereby, **BlocConsumer<Bloc, State>** enables developers to address this situation neatly. This flutter-provided class, analogous to a nested *BlocListener* and *BlocBuilder*, exposes a **builder** and **listener** to react to new states while reducing the boilerplate needed. Notice how *BlocConsumer<PlugPairingBloc, PlugPairingState>* relies on **buildWhen** for more granular control avoiding unnecessary builder calls, enhancing the widget's efficiency.

Code Sample IV

Lastly, we introduce the reader to an example of the only cubit implemented in the proposed Flutter application. The reason behind using blocs in favor of cubits is that the latter limit some relevant functionality, like automating logs and analytic events, which become convenient and valuable as the application grows. Nonetheless, this code sample exemplifies the usefulness of cubits for particular situations, rendering our review of state management implementations and use cases complete.

Listing 12: Bottom Navigation Cubit

```
1 class BottomNavigationCubit
2 extends Cubit<BottomNavigationPage> {
3   BottomNavigationCubit()
4   : super(BottomNavigationPage.home);
5
6   void switchTo(int index) {
```

```
7       if (index >= 0
8         && index <= BottomNavigationPage.values.length) {
9         emit(BottomNavigationPage.values[index]);
10      }
11    }
12  }
13
14  enum BottomNavigationPage {
15    home,
16    areas,
17    stats,
18    routines,
19    settings,
20  }
```

The code presented above refers to a cubit used to manage the navigation across the main tabs or pages of the implemented application. Thus, *BottomNavigationCubit* implements a single public method that manages the navigation flow, **switchTo**, and leverages an enum as the state shaping the cubit's behavior, *BottomNavigationPage*. Notice the evident similarities with its bloc counterpart, but most importantly, the reduction in boilerplate employed to implement a cubit.

2.4. Testing

This section presents the last and arguably, most critical content related to the proposed large-scale Flutter application's implementation. It provides the reader with a preamble that covers substantial insights into the motivations and criteria behind the proposed testing practices. Moreover, it introduces an exhaustive list of the most characteristic tests found in the application's test suite. Lastly, it culminates with some final remarks about the considered test cases and software testing in the context of this implementation.

2.4.1. Preamble & Considerations

"Ideally, your product should be shippable at any point, and tests can help you get there."

—

textbfScarlett Wardrop, Flutter testing: A very good guide [10 Insights] [75]

We consider testing a key area essential to delivering high-quality, maintainable, and scalable applications. Moreover, verifying all code behaves as intended allows reducing

risk, increasing confidence in a given codebase, and keeping current expectations and assumptions aligned. Ultimately, a well-structured test will always output the same result for any given input, ensuring the long-term functionality of code regardless of functionality and features added in the future.
Furthermore, Section 1.4 offered an extensive software testing analysis, touching upon some fundamental aspects, such as testing criteria and code coverage. Accordingly, we strive for one hundred percent code coverage for our entire codebase as a standard for code quality and test adequacy, enforcing the exercise of every line of code at least once. Additionally, we integrate these standards into a work methodology requiring developers to build features and corresponding tests as part of the same engineering effort. This approach encourages code ownership and responsibility while potentially boosting productivity and predictable behaviors across an engineering team.
Lastly, a comprehensive test suite requires extra time to write and maintain, which may hinder the initial progress of pure development tasks. Nonetheless, as a codebase grows, tests serve to avoid requirements ambiguity, communicate intended behavior, and identify and fix unwanted functionality or bugs. Thus, investing time in writing tests alongside feature implementation saves time in the long run by avoiding code rewrites and helps to deliver more stable and reliable products.

2.4.2. Package Testing

This subsection reviews relevant test cases structure and implementation for a client and a repository package. It serves as an example of the approach followed to test other similar packages, and it demonstrates the importance of software testing and test coverage at the infrastructure and domain layers. Moreover, notice that all the tests carried out belong to the unit testing category.

api_client

Let us now consider the **api_client** package to illustrate the testing practices employed to analyze its internal functionality. Before diving into the code, it is worth noting that this package is of utmost importance for our codebase as it interacts directly with the external GraphQL API, and hence, numerous repository packages rely on its functionality to access domain data. Thus, selecting this specific package to exemplify testing practices in client packages is not a coincidence. Instead, we deliberately chose this package to enable the reader to understand the implications of comprehensively testing this specific package and its effects on the broader scheme of packages and features included in the application. Moreover, we present the initial setup for the tests cases addressing one of the

resources exposed by the *ApiClient* class, the *AlarmResource*, in the code sample below.

Listing 13: Alarm Resource - Mocks and Fakes

```
class MockGraphQLCategory extends Mock
  implements GraphQLCategory {}

class FakeGraphQLRequest extends Fake
  implements GraphQLRequest<String> {}

void main() {
  group('AlarmResource', () {
    late GraphQLCategory graphQLCategory;
    late AlarmResource alarmResource;

    setUp(() {
      graphQLCategory = MockGraphQLCategory();
      alarmResource = AlarmResource(
        graphQLCategory: graphQLCategory,
      );
    });

    setUpAll(() {
      registerFallbackValue(FakeGraphQLRequest());
    });
}
```

Firstly, we must declare the **mock** and **fake** classes representing external dependencies leveraged by the *ApiClient* and the *AlarmResource* classes. Mocking or faking a dependency enables developers to isolate and focus on the tests instead of on the behavior or state of external dependencies. As stated in section 1.4.4, fakes are stand-in resources that provide only the necessary data, while mocks extend this concept with object behavior. In this case, *MockGraphQLCategory* allows simulating the behavior of queries, mutations, and subscriptions without interacting with the external API, while *FakeGraphQLRequest* allows to fake the request data included in the mocked object. Furthermore, gathering all the test setup and cases under a descriptive **group** under **main()** is a best practice that boosts clarity, organization, and maintainability as the test suite grows. Then, observe how we use the **setUp** function, a helper function that runs before executing every test, to instantiate the mocked *GraphQLCategory* and then inject it as a dependency of the *AlarmResource*, effectively allowing developers to control the behavior of queries, mutations, and subscriptions within this class object. Lastly, **setUpAll** registers a function

to be run once before all tests, in this case, **registerFallbackValue**. Passing an instance of *FakeGraphQLRequest* to *registerFallbackValue()* allows using **any()** as a parameter to mock objects, which is crucial for taking full advantage of mocking practices. Let us now introduce a simple test that serves as a basis for more complex examples.

Listing 14: Alarm Resource can be instantiated

```
1     test('can be instantiated', () {
2       expect(
3         () =>
4           AlarmResource(graphQLCategory: graphQLCategory),
5         returnsNormally,
6       );
7     });
```

The test included in the code above checks whether the code can successfully instantiate the *AlarmResource*. Notice that developers can effortlessly infer the intent of this test case by reading the test description, which must be coherent and descriptive. Furthermore, the **expect** function asserts that the first parameter, called *actual*, matches the second parameter, called *matcher*. In this case, we use **returnsNormally** as a matcher that matches a function call against no exception. Let us now take a look at more complex test cases shown in code snippet 32.

We nest the *alarms* group under the previously declared *AlarmResource* group for further organization. Notice this group refers to a method exposed by the *AlarmResource* class, and hence, all the test cases grouped under *alarms* address the functionality exhibited by this method. It is worth observing that well-structured and defined tests allow developers to understand the intent of the tested code without addressing or reviewing the code directly. Thereby, readily acknowledge two sets of tests cases to analyze when the *alarms* method **throws** an exception and when it returns successfully. Moreover, notice how the structure of all the tests shares evident similarities making testing the method's behavior almost systematic. Firstly, we include the **when** helper function to create a stub method response, allowing to call a method on a mock object and then a canned response method on the result. Notice we use **thenThrow** to throw a given exception and **thenAnswer** to return an asynchronous response, while *alarmsMalformedResponse*, *alarmsValidResponse*, and *alarmCollectionPayload* are variables holding String values. Lastly, we include an **expect** function to validate the test as we did in the previous example. In this case, the first argument corresponds to the method we are testing, alarms, through an instance of *MockAlarmResource*, while the second argument corresponds to the matcher we expect to validate the test against. Ultimately, we did not include all the tests required to cover all the behavior exposed by this given resource. However, this sample of test cases and

their previous setup are general enough to showcase how this approach shaped the rest of the test suite employed to analyze the remaining functionality exhibited by this resource, and most importantly, the rest of the resources exposed by the *ApiClient* class.

alarm_repository

Let us now introduce a series of insightful tests for the **alarm_repository**, which allow the reader to continue building knowledge about unit testing upon the previously analyzed *api_client* package. Much like in the example before, the following code and descriptions do not cover all the behavior exhibited by the *AlarmRepository* class and do not represent all the tests included in this package's test suite, but they provide the necessary information to comprehend how and why we applied these generic tests across different repositories within the domain layer. Once again, we start by reviewing the setup preceding the test cases.

Listing 15: Alarm Repository - Mocks and Fakes

```
class MockApiClient extends Mock
  implements ApiClient {}

class MockAlarmResource extends Mock
  implements AlarmResource {}

class FakeAlarmUpdatedPayload extends Fake
  implements AlarmUpdatedPayload {
    @override
    String get homeId => 'homeId';
    @override
    String get alarmId => 'id';
    @override
    Alarm get alarm => updatedAlarm;
}

class FakeAlarmCollection extends
  Fake implements AlarmCollection {
    @override
    List<Alarm> get alarms => [alarm];
}
```

From the code above, we notice that this test suite requires two mock objects, *MockApiClient* and *MockAlarmResource*, which will mock the behavior of *ApiClient* and *Alarm-*

Resource, respectively. Furthermore, we introduce two fake objects, *FakeAlarmUpdated-Payload* and *FakeAlarmCollection*, which will provide each test case with the necessary data. It is worth noting the use of overridden properties within the fake objects allowing them to access specific data and preventing tests from failing due to accessing null values. Moreover, as shown in source code 33 we perform a similar **setUp** as we did in the *api_client*, where we instantiate the mock objects and then inject them as an external dependency to *AlarmRepository*. Observe how we also include a series of when functions within this prior setUp as they represent shared desired functionality across the upcoming test cases. Most importantly, we need an instance of *MockAlarmResource* to control the behavior of this class within any given test case. Thus, we use **thenReturn** to return the said instance every time a test invokes *apiClient.alarmResource*.
Once more, we observe from the tests presented in code snippet 34 how we rely on nested test groups and detailed test descriptions to enhance the readability, structure, and thus, the maintainability of this given test suite. Firstly, we test that we can successfully instantiate the *AlarmRepository* class. Then, we validate the functionality of the alarms method exhibited by *AlarmRepository*, which internally leverages the *AlarmResource* and its exposed class methods alarms and *alarmUpdated*. We now focus on the relevant, different parts of the test cases not included in the *api_client* example to avoid explanation redundancy. Thus, the main distinction relies on dedicated *matchers* that validate Stream responses, such as **emits** and **emitsInOrder**. Aside from this characteristic, the rest of the test body follows the same structure introduced in the unit tests previously reviewed, facilitating a methodical and standardized approach to designing test cases. Ultimately, we use **when** to control the internal behavior of external dependencies and provide the desired data object, and **expect** to validate the results at the end of the test body.

One final comment about the exemplified test cases shown in this subsection is that implementing the underlying code that powers the helper methods used to create and manage mock and fake objects would be extremely time-consuming and inefficient. Therefore, our entire application's test suite relies on **mocktail**[24], a Dart package that focuses on providing a familiar, simple API for creating mocks in Dart (with null-safety) without the need for manual mocks or code generation.

[24] `https://pub.dev/packages/mocktail`

2.4.3. Bloc Testing

This subsection focuses on the unit tests employed to test the business logic components managing the state of our application, also known as blocs. However, despite these tests being unit tests conceptually and effectively, their nature and structure are distinct enough to address them as bloc tests. Thus, the test suite covering our entire business logic layer leverages **bloc_test**, a Dart package that simplifies testing blocs and cubits. Moreover, the code samples presented below illustrate our codebase's most representative use cases addressing bloc testing while covering the typical setup required to perform this process effectively. Let us now start by analyzing the initial bloc test setup shown in code snippet 35.

Firstly, as we learned in the previous testing subsection, we define the mock and fake objects that will allow us to acquire the necessary controllability over any given test case. Notice that we only need to mock the classes and data objects that belong to the layer located right below the business logic layer. Hence, *MockDeviceRepository* and *MockAlarmRepository* mock the behavior exhibited by *DeviceRepository* and *AlarmRepository*, respectively, both belonging to the Domain layer. Furthermore, *FakeAlarmResolvePayload* represents a data object faking the contents of an *AlarmResolvePayload* instance. Regarding the initial **setUp**, observe the evident similarities to the structure shown in the package testing section as we instantiate the mock objects to enforce controlled behavior on all test cases. Moreover, we also employ a grouping strategy to maintain test cases organized and semantically coherent. Let us now take a look at the first test presented in code snippet 36.

Although this first test is not a bloc Test, it is necessary to test the initial state produced by the bloc after instantiating it. It is worth observing the role proper dependency injection plays when testing blocs. Notice how developers can easily inject the mock repositories into the *WaterLeakAlarmBloc* to control any side effects or functionality produced by these external dependencies. Once again, we use **expect** to validate *WaterLeakAlarmBloc()*.state matches the expected *WaterLeakAlarmState()*. Let us now address the behavior produced by a given bloc event by introducing dedicated bloc tests.

Listing 16: Water Leak Alarm Data Fetched group

```
1 group('WaterLeakAlarmDataFetched', () {
2   WaterLeakAlarmBloc buildBloc() {
3     return WaterLeakAlarmBloc(
4       homeId: 'homeId',
5       deviceRepository: deviceRepository,
6       alarmRepository: alarmRepository,
```

```
7     );
8   }
9
10  blocTest<WaterLeakAlarmBloc, WaterLeakAlarmState>(
11    'emits correct state when  alarmRepository.alarms '
12    'returns Stream with expected alarms',
13    setUp: () {
14      when(() => alarmRepository.alarms(any()))
15          .thenAnswer((_) => Stream.value(alarms));
16    },
17    build: buildBloc,
18    act: (bloc) => bloc.add(const WaterLeakAlarmDataFetched()),
19    expect: () => <WaterLeakAlarmState>[
20      const WaterLeakAlarmState(
21        fetchedAlarmStatus: FetchedDevicesStatus.loading,
22      ),
23      WaterLeakAlarmState(
24        alarm: alarm,
25        fetchedAlarmStatus: FetchedDevicesStatus.success,
26      ),
27    ],
28  );
29
30  blocTest<WaterLeakAlarmBloc, WaterLeakAlarmState>(
31    'emits error',
32    setUp: () {
33      when(() => alarmRepository.alarms(any()))
34          .thenAnswer((_) => Stream.error('Error'));
35    },
36    build: () => WaterLeakAlarmBloc(
37      homeId: 'homeId',
38      deviceRepository: deviceRepository,
39      alarmRepository: alarmRepository,
40    ),
41    act: (bloc) => bloc.add(const WaterLeakAlarmDataFetched()),
42    expect: () => const <WaterLeakAlarmState>[
43      WaterLeakAlarmState(
44        fetchedAlarmStatus: FetchedDevicesStatus.loading,
45      ),
```

```
46        WaterLeakAlarmState(
47          fetchedAlarmStatus: FetchedDevicesStatus.error,
48        ),
49      ],
50    );
51  });
```

Observe how we gather all the test cases related to the *WaterLeakAlarmDataFetched* bloc event under the same group. This grouping approach allows developers to tackle functionality on an event-by-event basis in an orderly manner. Furthermore, a common practice is to create custom helper functions, such as **buildBloc()**, which developers can reuse across multiple test cases to avoid code redundancy. Moreover, we introduce a detailed dissection of the underlying functionality of a **blocTest** to understand the two examples presented in the code above:

- **blocTest** - Creates a new *bloc*-specific test case with the given description. It handles asserting the orderly emission of the expected bloc after executing the act function. Additionally, it ensures that no additional states are emitted by closing the bloc stream before evaluating the expectation.
- **setUp** - It is an optional parameter used to set up any dependencies before initializing the bloc under test. Ultimately, it sets up the necessary state for a particular test case. build - Constructs and returns the bloc under test.
- **act** - It is an optional callback parameter invoked with the bloc under test and used to interact with the bloc.
- **expect** - It is an optional *Function* parameter that allows verifying that the bloc under test emits the expected returned *Matcher* after executing the act function. Notice that said *Matcher* is often an ordered list of the emitted bloc states, as shown in the test cases above.

Moreover, we introduce one final bloc test in code snippet 37 to complete the bloc testing process review. Notice that this code addresses a new bloc event, *WaterLeakAlarmOkStatusBannerClosed*. This test case features the same structure as the previous bloc tests, but it presents a subtle yet powerful variation. This variation refers to the **seed** parameter, an optional Function that returns a state used to seed the bloc before calling act.

2.4.4. Widget Testing

Building a large-scale Flutter application requires developing numerous features which heavily rely on a non-trivial combination of Widgets and states to deliver the intended product value to end-users. Manually testing all these features would incur tremendous

and often unaffordable human efforts. Therefore, automated tests mitigate these efforts by ensuring a given app performs correctly before publishing it while retaining features and bug-fix velocity. Furthermore, the Flutter framework supports a variety of automated tests summarized in the table below 2.1.

Test Type	Definition	When Best Used
Unit	A function/method/variable in isolation Is a pure Dart test, no Flutter dependency	Testing contract and business logic Quick feedback for developers to improve code confidence
Widget	A single UI Component The output is the widget sub-tree, not the rendered widget itself	Testing UI in isolation Can include a broader feature but need to be contained
Integration	An end-to-end experience with mocked dependencies	To find business logic breaks in a client
End-to-End	An end-to-end experience with a real backend and/or hardware Is a "black box"	Attempting to validate the full end-user experience
Golden	The "pixel-by-pixel" spec	Once you have widget tests and UI is locked down

Table 2.1: Flutter test classification [75]

This thesis work focuses on widget testing to validate the behavior of the presentation layer, and hence, this subsection introduces the reader to this new testing approach. A **widget test**, also known as component test in other UI frameworks, tests a single widget. Its goal is to verify that a widget's UI looks and interacts as expected. Furthermore, widget testing involves multiple classes and requires a test environment that provides the appropriate widget lifecycle context [24]. It is worth mentioning that a widget test is more comprehensive than a unit test, though it also requires a simplified test environment to validate the results. Therefore, it is critical to properly architect the Flutter application, and most importantly, its Widget Tree to provide a widget test suite with such a testing environment. We consider that the approach illustrated in code snippet 38 provides the most effective and least error-prone architecture for widget-based feature integration and testing. Notice how the App widget, placed at the top of the Widget Tree, requires injecting all the necessary repositories as already-instantiated injectable dependencies. Furthermore, we use *MultiRepositoryProvider* to merge multiple *RepositoryProvider* widgets into a single widget tree, which improves the readability and eliminates the need to nest multiple *RepositoryProviders*, while ensuring child widgets have access to the injected repositories. The implications of this DI approach are tremendously relevant as it allows developers to have access to these repositories anywhere down the Widget Tree via *BuildContext*, enabling them to inject any repository into any given bloc that may require its exposed functionality. Lastly, observe that we, once again, use the Page-View pattern allowing us to pass the *AppView* widget as the child funneling down this functionality.
Moreover, this approach facilitates the creation of a helper extension, *AppTester*, on **Wid-**

getTester, a class that programmatically interacts with widgets and the test environment. Thus, the sole purpose of this helper extension is to provide a flexible environment for a *widgetUnderTest*. This environment leverages a *MaterialApp* widget as a wrapper for *widgetUnderTest* and defaults blocs and repositories to mock instances unless developers inject custom mock objects to achieve further controllability. The invaluable benefit of using this helper will become evident once we review the widget test cases introduced in the upcoming code samples. Hence, we present the reader with our proposed setup for widget testing in code snippet 39 and three different examples addressing the most common use cases for testing widgets in a large-scale Flutter application.

Setup

Firstly, we declare the mock objects leveraged across the different test cases, as we did in previous testing examples. In this case, we are testing the presentation layer, and hence, the external dependencies or components producing side effects within test cases are typically blocs. Therefore, *bloc_test* provides a helper class, **MockBloc**, enabling developers to create mock blocs that implement all the necessary fields and methods and allow further customization at *runtime* to define how the bloc may behave. Furthermore, we employ the same **group**ing and **setUp** approach employed in previous test examples by providing an explicit description of the widget under test, *WaterLeakAlarmDetailsView*, and instantiating the mock object, *MockWaterLeakAlarmBloc*. Notice the **setUp** function also defines the behavior of *MockWaterLeakAlarmBloc* when accessing its state by returning an instance of its initial state, *WaterLeakAlarmState*.

Listing 17: Mock Water Leak Alarm Bloc - setup

```
1  class MockWaterLeakAlarmBloc
2      extends MockBloc<WaterLeakAlarmEvent, WaterLeakAlarmState>
3      implements WaterLeakAlarmBloc {}
4
5  void main() {
6    group('WaterLeakAlarmDetailsView', () {
7      late WaterLeakAlarmBloc waterLeakAlarmBloc;
8
9      setUp(() {
10       waterLeakAlarmBloc = MockWaterLeakAlarmBloc();
11       when(() => waterLeakAlarmBloc.state).thenReturn(
12         const WaterLeakAlarmState(),
13       );
14     });
```

```
15    )};
16  }
```

Lastly, we introduce a common practice implemented across the application widget-testing files, which relies on a **WidgetTester** extension, *pumpWaterLeakAlarmDetailsView*, that allows developers to inject a mock bloc with custom behavior into the previously analyzed helper method, **pumpApp**. Ultimately, this technique saves time, improves test readability, and avoids potentially unintended behavior.

Listing 18: Custom extension on WidgetTester

```
1  extension on WidgetTester {
2    Future<void> pumpWaterLeakAlarmDetailsView({
3      required WaterLeakAlarmBloc waterLeakAlarmBloc,
4    }) =>
5        pumpApp(
6          BlocProvider.value(
7            value: waterLeakAlarmBloc,
8            child: WaterLeakAlarmDetailsView(),
9          ),
10       );
11 }
```

renders

Testing whether and when a Flutter application renders a given widget is arguably the most fundamental widget test. Thus, we provide the reader with two examples addressing this test case. Notice that both tests adhere to the same structure using the **testWidgets** function to access the **WidgetTester** object. Once inside the test body, we control the behavior of the mock bloc by returning the desired state, which will determine the rendered widgets. Moreover, we *pump* the application by leveraging the previously described extension method, *pumpWaterLeakAlarmDetailsView*, providing the widget under test, *WaterLeakAlarmDetailsView*, access to the mock bloc, *MockWaterLeakAlarmBloc*. Lastly, the **expect** function validates the test's result by leveraging **find.byType** (Finder) and **findsOneWidget** (Matcher), allowing developers to assert that the Finder locates exactly one widget in the widget tree.

Listing 19: Mock Water Leak Alarm Bloc - renders group

```
1  group('renders', () {
2        testWidgets(
3          'SomethingWrongWithDevice on ValveStatus.fault',
4          (WidgetTester tester) async {
```

```
5          when(() => waterLeakAlarmBloc.state).thenReturn(
6            const WaterLeakAlarmState(
7              valve: valve,
8              sensors: sensors,
9              fetchedDevicesStatus: FetchedDevicesStatus.success,
10             valveStatus: ValveStatus.fault,
11           ),
12         );
13         await tester.pumpWaterLeakAlarmDetailsView(
14           waterLeakAlarmBloc: waterLeakAlarmBloc,
15         );
16         expect(
17           find.byType(SomethingWrongWithDevice),
18           findsOneWidget,
19         );
20       },
21     );
22
23     testWidgets(
24       'CircularProgressIndicator when FetchedDevicesStatus.loading',
25       (WidgetTester tester) async {
26         when(() => waterLeakAlarmBloc.state).thenReturn(
27           const WaterLeakAlarmState(
28             valve: valve,
29             sensors: sensors,
30             fetchedDevicesStatus: FetchedDevicesStatus.loading,
31           ),
32         );
33         await tester.pumpWaterLeakAlarmDetailsView(
34           waterLeakAlarmBloc: waterLeakAlarmBloc,
35         );
36         expect(
37           find.byType(CircularProgressIndicator),
38           findsOneWidget,
39         );
40       },
41     );
42   });
```

shows

Another typical use case for widget testing is to check whether the app **shows** a specific widget. For this purpose, we use a similar approach to the one presented before, but we introduce a new helper function provided by *bloc_test*, **whenListen**. This function creates a stub response for the **listen** method on a given bloc, allowing to return a canned Stream of states for a bloc instance. Moreover, **whenListen** also handles stubbing the bloc's state, keeping it in sync with the emitted state. Lastly, attempt to **find** the desired widget **byKey**, this time.

Listing 20: Mock Water Leak Alarm Bloc - shows group

```
1  group('shows', () {
2        testWidgets(
3          'error snack bar if FetchedDevicesStatus.error',
4          (tester) async {
5            whenListen(
6              waterLeakAlarmBloc,
7              Stream.value(
8                const WaterLeakAlarmState(
9                  valve: valve,
10                 sensors: sensors,
11                 fetchedDevicesStatus: FetchedDevicesStatus.error,
12               ),
13             ),
14           );
15           await tester.pumpWaterLeakAlarmDetailsView(
16             waterLeakAlarmBloc: waterLeakAlarmBloc,
17           );
18           expect(
19             find.byKey(const Key('fetchAlarm_error_snackBar')),
20             findsOneWidget,
21           );
22         },
23       );
24     });
```

adds

This third example covers a test case where developers need to verify whether a specific function has been called. For this purpose, we introduce another widget, *WaterSystemOk-*

Banner. Thereby, we find two essential differences when compared to the previous widget tests. Firstly, we use the instance of **WidgetTester**, tester, to act on a specific widget, found **byKey**, by **tap**ping on it. Secondly, once the action completes successfully, we must verify that the function under test, *waterLeakAlarmBloc.add(const WaterLeakAlarmOk-StatusBannerClose())*, has been **called** exactly once. Ultimately, the **WidgetTester** class exposes numerous methods to interact with widgets and the test environment programmatically, allowing developers to test all sorts of actions on a given widget.

Listing 21: Mock Water Leak Alarm Bloc - adds group

```
class MockWaterLeakAlarmBloc
    extends MockBloc<WaterLeakAlarmEvent, WaterLeakAlarmState>
    implements WaterLeakAlarmBloc {}

void main() {
  group('WaterSystemOkBanner', () {
    late WaterLeakAlarmBloc waterLeakAlarmBloc;

    setUp(() {
      waterLeakAlarmBloc = MockWaterLeakAlarmBloc();
    });

    group('adds', () {
      testWidgets(
        'tap close button hides banner',
        (WidgetTester tester) async {
          when(() => waterLeakAlarmBloc.state).thenReturn(
            const WaterLeakAlarmState(
              displayOkStatusBanner: true,
            ),
          );
          await tester.pumpBanner(
            waterLeakAlarmBloc: waterLeakAlarmBloc,
          );

          await tester.tap(
            find.byKey(const Key('waterSystemOkBanner_closeButton')),
          );
          verify(
            () => waterLeakAlarmBloc.add(
```

```
31                 const WaterLeakAlarmOkStatusBannerClose(),
32               ),
33             ).called(1);
34           },
35         );
36       });
37     });
38   }
39 
40   extension on WidgetTester {
41     Future<void> pumpBanner({
42       required WaterLeakAlarmBloc waterLeakAlarmBloc,
43     }) =>
44         pumpApp(
45           BlocProvider.value(
46             value: waterLeakAlarmBloc,
47             child: const WaterSystemOkBanner(),
48           ),
49         );
50   }
```

One final comment about the exemplified cases about widget testing shown in this subsection is that it would be remarkably challenging to test all the widgets in an application without following a coherent and unambiguous architectural design, such as the **Page-View pattern**, as we could not inject the necessary dependencies to gained the indispensable controllability over any given widget test case.

2.4.5. Remarks

After reviewing various insightful concepts and examples about testing the different layers and components of a large-scale Flutter application, we conclude this section with a few final remarks. Firstly, let us emphasize the importance of employing a standardized, predictable, and systematic approach when implementing and testing features. This methodological procedure should allow developers to build a reliable, easy-to-navigate, and sound codebase for any large-scale Flutter application, enhancing its maintenance, scalability, and testability.
Moreover, tests files and directories should always mirror the structure of the implementation project. Therefore, whether the tests belong to a domain or data layer package, or the business logic or presentation layer, there must be a test directory at the same level as

the folder containing the application implementation files, the lib folder, in Dart/Flutter projects. Additionally, these directories should have the same name both in the test and implementation directories, while test files should add the suffix _*test* to the name of its corresponding implementation file.
Lastly, we encourage using graphical tools that allow developers to track the testing coverage of any given file, package, or project. Thus, we propose using **lcov**[25], a graphical front-end for GCC's coverage testing tool that collects gcov[26] data for multiple source files and creates HTML pages containing the source code annotated with coverage information. More importantly, it supports statement, function, and branch coverage measurement and provides overview pages for easy navigation within the file structure.

[25]http://ltp.sourceforge.net/coverage/lcov.php
[26]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

3 | Results

This chapter navigates the reader through the results derived from the implementation endeavors presented in the previous chapter 2. Thereby, it provides quantitative and qualitative data supporting the objectives of this thesis.

3.1. Quantitative Data

The data presented in this section responds to a quantitative nature and serves as a means to appreciate and support the decisions taken throughout the implementation of the proposed Flutter application. Furthermore, this data intends to provide tangible insights into the outcome of leveraging the hybrid architecture, relying on BLoC and *flutter_bloc* for our state management solution, and enforcing comprehensive testing practices.

packages

The completion of this project required developers to implement 40 Dart packages. Out of all these packages, 12 correspond to data clients representing the application's data layer, 19 correspond to domain repositories representing the application's data layer, 2 correspond to plugins, and the remaining 7 correspond to utility or helper packages. It is worth mentioning that the *api_client*, one of the core data packages, exposed 8 resources which laid the foundation for another 8 domain repositories. Lastly, Table 3.1 below display further quantitative information related to client and repository packages. Notice the lines column does not refer to the number of lines implemented in a given package but its number of testable lines belonging to statement, branch, and path code.

Clients	Repositories	Tests	Covered Lines
12	19	985	4362

Table 3.1: Quantitative results from Dart packages.

lib

Let us now focus on the business logic and presentation layers, the upper layers of the proposed architecture encompassed by the lib directory. To deliver the desired value to end-users, developers implemented a total of 123 Application features. Most importantly, the application leveraged 69 distinct blocs that manage the state of the Flutter application and required 488 dedicated bloc tests to thoroughly cover all the business logic and functionality exposed by all the blocs. Lastly, testing the application's presentation layer involved 1306 widget tests which included, but were not limited to, all the rendering, navigation, and function-calling use cases. Lastly, Table 3.2 below displays a summarized quantitative data analysis related to the contents of the lib directory.

App Features	Blocs	Bloc Tests	Widget Tests	Total App Tests	Covered lines
123	69	488	1306	2014	13247

Table 3.2: Quantitative results from lib directory.

Totals

This short subsection aims to provide a concise yet precise view of the previously presented data. Thus, Table 3.3 below illustrates the data aggregation of the quantitative values previously collected and reviewed.

Packages	Clients	Repositories	Blocs	App Features	Tests	Covered lines
40	12	19	69	123	3655	20204

Table 3.3: Summary of total quantitative results.

3.2. Qualitative Data

This section gathers essential and compelling qualitative data, which serves as a means to back and support the process and outcome of this thesis project. The thesis' author carried out a series of surveys involving software engineers directly or indirectly implicated in the large-scale application's development referenced in this document. Furthermore, the objective was to collect insightful and valuable assessments about consequential decisions and core concepts discussed throughout this thesis work, such as software architecture, maintainability, scalability, testability, code quality, state management, Flutter, and code quality. See below a relevant description of the surveyed engineers:

- Óscar Martin - Senior Software Engineer I at Very Good Ventures, Project Lead, and Flutter Spain co-founder. He was directly involved in the project.
- Jaime Blasco - Software Engineer II at Very Good Ventures, Google Developer Expert for Flutter, and Flutter Spain co-founder. He was directly involved in the project.
- Jorge Coca - Head of Engineering at Very Good Ventures, Google Developer Expert for Flutter, and organizer of Chicago Flutter.
- Dominik Roszkowski - Principal Engineer at Very Good Ventures and Google Developer Expert for Flutter.

Let us now present the gathered data.

Code Quality

Both Óscar and Jaime evaluated the quality of the proposed App's codebase with four out of five maximum points. Óscar pointed out that the App has some large features containing nested sub-features, which increased the overall complexity of the application. However, the overall consensus was that the codebase's code quality was notably high. The App features clearly-differentiated domains, and classes and methods are predictable while having a single responsibility. Moreover, it is worth noting that we "kept the majority of the features independent and could be modularized in the future" based on requirements and specific app constraints.

"This is something very subjective, but a great application is read easily, and its features can be quickly located. From an objective point of view, a well-tested app (100% is our standard) that runs on CI/CD every code change tends to be a very important indicator of quality."

— **Jorge Coca**, on code quality in Flutter Apps.

Architecture

All surveyed engineers raised concerns about having either purely layered or feature-oriented architectures. Regarding purely layered architectures, they stated that scalability was an evident problem as adding or modifying features require developers to adjust multiple files and directories, creating version control conflicts in a distributed software environment. On the other hand, purely feature-oriented architectures add a level of complexity that makes feature definition, reusability, directory organization non-trivial. More specifically, achieving fully independent features that share resources without duplicating code and poorly affecting maintainability often requires heuristics and a non-standardized approach. Moreover, Jorge defined modularization as the is simply the art of organizing

code efficiently. He added that a well-modularized codebase simplifies development, reduces the cost in infrastructure, and improves communication and ownership across the organization, although it is purely a human trait intended to enhance development productivity since computers do not care about modularization. Ultimately, the consensus was that a hybrid architecture allows developers to leverage the strengths of each approach while minimizing the negative effects of their intrinsic drawbacks.

State Management

The survey asked the respondents to state which state management solutions they had used in the past. See the table below to check the responses.

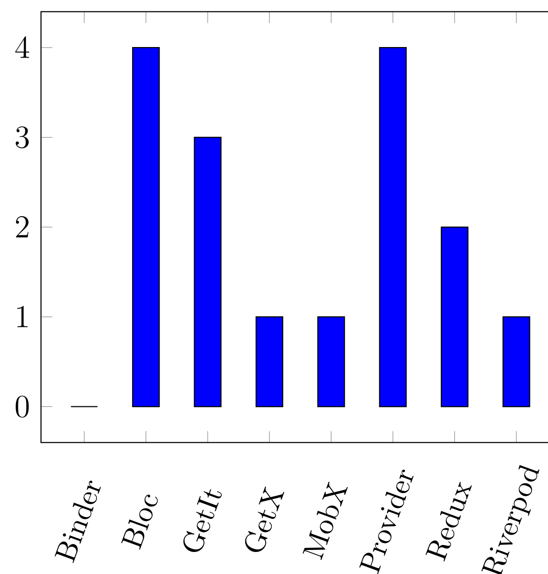

Figure 3.1: State Management Solutions

Then, they provided their personal input regarding the advantages and disadvantages of the BLoC pattern in Flutter Applications.

Advantages:

- Easier to modularize and to know the current state of your application.
- It handles the states through a stream of events that modify the current state, allowing developers to know which state the application is currently in and test it accordingly.
- Bloc is consistent, predictable, easy to test, hard to break, and has a strong community that shares tutorials, documentation, plugins, articles, videos...
- It is made for testing.
- It has built-in observability of the events and states.

- It provides constraints on the app modules' structure and architecture, making blocs look similar to each other.

Disadvantages:

- It requires a new development mindset since most developers learn declarative programming and not state machine drive architectures.
- Some systems are hard to define as state machines. For instance, bloc can be counterproductive for background processing e.g. especially when a given process needs to be repeated for some number of objects.
- Hard to learn for beginners.
- Some UI components or interactions do not fit well within the state/event paradigm (eg: showing Snack Bars)

Moreover, their level of satisfaction towards the BLoC pattern and the bloc library received an average of four-point-five out of five maximum points. Overall, Jorge pointed out that the Bloc pattern is the most suitable pattern to handle state management in Flutter applications. Additionally, he emphasized that BLoC acknowledges the reactive nature of applications' UIs and that it helps developers think of every possible combination of states a UI may handle, making it very easy to test.

Maintainability, Testability, and Scalability

Jorge mentioned that these terms are all interconnected. For him, a scalable project relates to the ability to serve 0 to N customers regardless of its infrastructure, meaning that the project's setup should enable 1 to M developers to work on the codebase efficiently, making it easy to maintain and release. To achieve this outcome, he concluded that a sound test suite and a codebase that makes testing easy is critical. Notice how this statement aligns perfectly with the core testing values and practices presented throughout this thesis.

"A well-maintained App allows for good scalability through tests."

— **Dominik Roszkowski**, on the relationship among maintainability, testability, and scalability.

Moreover, both Jorge and Dominik shared a similar perspective about the benefits of adding testing into a project's development cycle, outlining the following key points:

- ability to spot new bugs introduced in existing code, thus time saved in the long run.
- more focus on the implementation being approachable and maintainable.
- ease of verifying the implementation e.g. it's easier to run tests than to launch the complete program.
- the use cases envisioned by the original author are easily spotted.

However, Dominik also pointed out some worth-considering drawbacks:

- more time spent on the feature development
- bigger refactoring or API changes are more demanding as they require test updates
- if incorrectly written, testing can lead to confusion or a false sense of confidence
- as the App grows, running tests can take a significant amount of time on the dev machine or CI
- some tests (e.g. for data models) can be mundane and repetitive

Jaime and Óscar mentioned that it helps to reduce bug-fixing time during sprints. Additionally, it allows developers to keep improving and adding new features ensuring the previous code still behaves running correctly. Ultimately, they rated the application's testability with five out of the five maximum scores.

"Testing is simply the art of identifying, understanding, and controlling the behavior of your inputs (dependencies), so you can put them in different scenarios that you can compare against an expected output."

— **Jorge Coca**, on testing practices in Flutter Apps.

Regarding the 100% coverage enforcement, the common opinion was that it increases the confidence of developers and stakeholders in their ability to be successful, while any other number different than 100% highlights a gap in the testing strategy. Notice that gap could be in a non-critical area of the project or a critical flow of your product, but unless exercised by a human or in an automated fashion, there is no way to determine the gap location. Therefore, relying on automated testing saves money and time as human labor is always more expensive and error-prone. Moreover, Jorge confirms that enforcing 100% percent in all projects has allowed Very Good Ventures to ship faster, safer, and more reliable code, making the development process more cost-efficient. Moreover, Dominik provided further valuable insights about testing from the business and development perspectives.

From the **business** point of view:

- it gives the client confidence in the delivered code, especially during the hand-off.
- it reduces the time for QA and new features or changes in the long-run.
- it helps to build authority as experts in the field.

From the **developer**'s point of view:

- it forces them to think more consciously about their code.
- tests notify whether something is broken in the API or unexpected results arise.
- it improves self-confidence, especially if previously widespread testing was considered to be hard or *impossible*.

Both Jaime and Óscar rated the testability of the application's code base with five out a maximum of five points. They mentioned that testing is always a top priority through-

out any given development sprint and that the overall application architecture facilitates module and feature testing. However, Jaime pointed out that there is room for improvement by including golden tests that ensure the UI aspect and behavior, and integration tests that checked the entire application flow.

"At the end of the day, 100% coverage guarantees confidence and reduces the costs of development by enforcing rules and standards in an automated way."

— **Jorge Coca**, on why Very Good Ventures emphasizes 100% coverage on all projects.

As far as maintainability, Jaime stated that the implemented codebase has undergone a continuous evolution allowing developers to adapt to changing requirements. However, technical debt arose along the way, affecting specific parts or features of the application, such as inconsistencies in the application theming. Furthermore, they agreed that the modularization approach implemented throughout the application architecture facilitates the maintainability of features and modules. Overall, they rated the maintainability of the application's codebase with four-point-five out of a maximum of five scores.

Lastly, Óscar and Jaime rated the scalability of the application's codebase with five out of a maximum of five points. They stated that the implemented application features a clear division between layers and domains. Moreover, all dependencies shaping any given feature are constrained by the defined app requirements, while each feature is scoped, allowing developers to keep growing the codebase with new requirements without negatively impacting the previous code. However, Jaime emphasized the importance of reviewing features independently to further improve their modularity.

Flutter

Lastly, the survey prompted the respondents to give feedback and opinions about Flutter. Óscar and Jaime focused on the productivity and efficiency benefits using Flutter provides to developers. Based on their experience, they confirmed that development is faster, allowing developers to focus on the product instead of on the platform where it needs to be deployed. Moreover, they mentioned the numerous benefits of having a single team developing one application targetting various platforms, rather than multiple teams managing various applications depending on the target platform. Thus, Flutter development presents evident benefits in terms of team performance and costs. Jorge also pointed out that, although other cross-platform frameworks achieved being able to write once and deploy in more platforms, there was always a penalty price that someone had to pay, whether it was performance (final users), developer experience (employees), or lack of a strong community that elevated the practice (organization). However, the surveyed engineers also pointed out relevant drawbacks worth considering, such as:

- Flutter has to fight against the pre-existing stigma that cross-platform development is slow, bad, or does not scale.
- Dart is not a popular language and makes organizations reluctant about using it for industry projects.
- Dart is less powerful than Kotilin or Swift
- Flutter has limited access to system APIs (background processing, camera access and control)
- Flutter is not the official supported approach to developing Android and iOS apps.

"I think we will see Flutter running everywhere: it will start with Toyota and their vehicles, and I am sure other vehicle manufacturers will follow... but I think of stadium Jumbotrons, Times Square, smart devices at home... anywhere where there is a screen!"

— **Jorge Coca**, on the future of Flutter, its ecosystem, and community.

4 | Related Work

This chapter wraps up the work presented in this thesis by providing the reader with additional insights into other authors' academic endeavors related to the knowledge presented in this document. It reviews a series of research and thesis papers focused on software architecture, state management solutions, and testability in Flutter applications. Furthermore, the following academic contributions allowed this thesis' author to assemble a sizable portion of the content included in the Background and Motivations chapter 1. Ultimately, they served as a solid theoretical and empirical foundation for this thesis, inspiring its author throughout the research and implementation phases of the proposed work.

Sebastian Faust's thesis documented the crucial steps most development teams may face using Flutter in a large-scale application [31]. His work included the creation of a large-scale application used as a reference to introduce the steps taken during its development, providing a thorough review of the decisions and evaluated options shaping this process. Furthermore, he shared comprehensive insights into the wide range of explored, compared, and analyzed solutions about state management and software architecture. His work covered valuable topics such as an extensive review of the Flutter framework, immutability, dependency injection, and modularization, among other content. Moreover, his thesis shares an interview with Felix Angelov, the current Head of Architecture and Principal Engineer at Very Good Ventures, to support his decisions and implementation regarding state management and application architecture. He essentially built an arguably large-scale Flutter application leveraging a four-tier architecture, the BLoC pattern as the chosen solution for state management, and Dart packages to achieve enhanced modularization. Nonetheless, the size of his application and his understanding of "a large-scale application" fade compared to the work presented in this thesis. Moreover, he did not reference testing as a core pillar of software development based on its effects on software maintainability and scalability. Hence, his work lacks references to automated testing covering unit, bloc, and widget tests cases.

Moreover, Michał Szczepanik and Michał Kędziora [64], Ly Hong Hoang [39], Dmitrii Slepnev [59] contributed to the Flutter literature by furthering the analysis and review of state management solutions employed in applications leveraging this cross-platform

framework, being the latter author who provided the most comprehensive work about such solutions. Dimitrii's thesis focused on categorizing state management approaches and provided a means to select the most suitable solution for the most common use cases. His research efforts relied on quantitative analysis about mobile app development, including its market, underlying operating systems, and available options, a thorough review of the Flutter framework and Dart, and a comprehensive analysis of state management approaches. Thereby, he used the learned knowledge to implement a Flutter application leveraging each of the most representative state management solutions, including setState, InheritedWidget, Provider, GetX, BLoC, MobX, Redux. Ultimately, he based his analysis and validation criteria on the following six aspects: Complexity, Boilerplate code, Code generation, Time travel, Scalability, and Testability
His analysis led him to conclude that BLoC and its Flutter implementation with flutter_bloc were the most suitable choice to build a highly scalable and testable Flutter application, aligning with the state management solution proposed in this thesis.

Lastly, it is worth noting that there is a scarce number of academic articles focused on Flutter. Therefore, this thesis refers to trusted, genuine, and accurate content elaborated by influential and renowned figures in the Flutter community to complement the scientific papers composing this work's bibliography. Accordingly, we cited numerous official sources to include the utmost rigorous and precise knowledge on subjects like Flutter, Dart, state management, or bloc, which made up for the lack of academic papers.

5 | Conclusions

This thesis intended to provide a standardized and almost systematic approach to building large-scale Flutter applications. Firstly, we proposed a hybrid architecture combining the strengths and advantages of the layered and feature-oriented architectural design patterns. Furthermore, we enhanced this hybrid architecture by complementing it with a modularization approach based on Dart packages. Regarding the selected state management solution, we provided coherent criteria to support the choice of the BLoC pattern and its Flutter implementation with *flutter_bloc*. We also introduced the Page-View pattern, a straightforward approach to inject blocs into the presentation layer and simplify the access to the state from any given Widget. Moreover, this thesis emphasized the importance of testing as a core activity of the software development lifecycle while enforcing one hundred percent code coverage for the entire application's codebase, demonstrating its positive effects on the long-term maintainability and scalability of any given Flutter application. Accordingly, this thesis illustrated all the non-trivial decisions and implementation details through general, transferable, and comprehensive code examples. Most importantly, the results derived from this thesis work support the decisions taken throughout the completion of the Flutter application's implementation phase. From a quantitative perspective, the results validated our proposal as we were able to deliver a large-scale application meeting, or exceeding, the expectations about the quality of the product delivered to end-users. On the other hand, the qualitative data obtained through testimonials and observations of Flutter experts, directly and indirectly, involved in the project's development also backed the thesis author's proposed work. Ultimately,

this thesis contributed to the existing yet limited literature about the Flutter framework by introducing comprehensive research knowledge, which gathered and aggregated information from numerous reliable sources. It is also worth mentioning that the size of the project this thesis builds upon corresponds to an undoubtedly large-scale Flutter application. Lastly, unlike many other related studies, its development, testing, and validation were carried out in a professional, industry-oriented environment involving real-world stakeholders.

Before concluding this thesis, we propose the following relevant topics that could extend the work presented in this document, its line of study, and further contribute to the Flutter literature:

- Enhance the test suite of this Flutter application, or any other large-scale Flutter application, by including golden tests and integration testing. The analysis and implementation of these two test types should provide valuable insights into their implications on a given App's maintainability, scalability, and testability.
- Expand this work to evaluate the multi-platform performance of the proposed architecture and state management solution compared to other approaches.
- Build a fully modular Flutter application where any given feature could run independently, possibly leveraging a micro-service architecture, allowing total control over feature addition and deletion, and maximizing its reusability across different applications.

Bibliography

[1] *Microsoft Application Architecture Guide: Patterns & Practices.* Microsoft, 2nd edition, 2009.

[2] S. Akopkokhyants. *Beyond Dart's Basics.* 2014.

[3] T.-M. G. e. a. Andreas Biørn-Hansen, Christoph Rieger. An empirical investigation of performance overhead in cross-platform mobile development frameworks. 25:175–199, July 2020. ISSN 2997–3040. doi: 10.1007/s10664-020-09827-6.

[4] F. Angelov and Contributors. bloc library, . URL `https://bloclibrary.dev/`.

[5] F. Angelov and Contributors. bloc package, . URL `https://pub.dev/packages/bloc`.

[6] F. Angelov and Contributors. flutter_bloc package, . URL `https://pub.dev/packages/flutter_bloc`.

[7] F. Angelov and Contributors. hydrated_block package, . URL `https://pub.dev/packages/hydrated_bloc`.

[8] F. Angelov and Contributors. block_test package, . URL `https://pub.dev/packages/bloc_test`.

[9] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8:49–84, 07 2009. doi: 10.5381/jot.2009.8.5.c5.

[10] K. Beck. *Smalltalk Best Practice Patterns*, volume 1 of *Coding*. Prentice Hall, 1997.

[11] J. C. Bender and J. McWherter. *Professional test driven development with C#: Developing real world applications with TDD.* Wiley, 2011.

[12] R. V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37 (9):87–101, sep 1994. ISSN 0001-0782. doi: 10.1145/182987.184077. URL `https://doi.org/10.1145/182987.184077`.

[13] D. Boelens. Reactive programming - streams - bloc. aug 2018. URL `https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/`.

[14] G. Booch. *Object-oriented analysis and design with applications.* Addison-Wesley, 2nd edition, 2007.

[15] J. Bosch and P. Molin. Software architecture design: evaluation and transformation. pages 4 – 10, 04 1999. ISBN 0-7695-0028-5. doi: 10.1109/ECBS.1999.755855.

[16] T. A. M. Christoph Rieger. Towards the definitive evaluation framework for cross-platform app development approaches. 153:175–199, Apr. 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.04.001. URL https://www.sciencedirect.com/science/article/pii/S0164121219300743.

[17] J. Coca. Why we use flutter_bloc for state management, jun 2021. URL https://verygood.ventures/blog/why-we-use-flutter-bloc.

[18] G. Developers. Flutter live - flutter announcements and updates (livestream), 2018. URL https://www.youtube.com/watch?v=NQ5HVyqg1Qc&t=4842s.

[19] E. W. Dijkstra. The structure of the "the"-multiprogramming system. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, page 10.1–10.6, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450373708. doi: 10.1145/800001.811672. URL https://doi.org/10.1145/800001.811672.

[20] F. A. Documentation. State class, . URL https://api.flutter.dev/flutter/widgets/State-class.html.

[21] O. F. Documentation. Hot reload, . URL https://docs.flutter.dev/development/tools/hot-reload.

[22] O. F. Documentation. List of state management approaches, . URL https://docs.flutter.dev/development/data-and-backend/state-mgmt/options.

[23] O. F. Documentation. Using packages, . URL https://docs.flutter.dev/development/packages-and-plugins/using-packages.

[24] O. F. Documentation. Testing flutter apps, . URL https://docs.flutter.dev/testing.

[25] O. F. Documentation. What is flutter, . URL https://docs.flutter.dev/resources/faq#what-is-flutter.

[26] O. F. Documentation. Flutter architectural overview, . URL https://docs.flutter.dev/resources/architectural-overview.

[27] O. F. Documentation. Start thinking declaratively, . URL `https://docs.flutter.dev/development/data-and-backend/state-mgmt/declarative`.

[28] O. F. Documentation. Differentiate between ephemeral state and app state, . URL `https://docs.flutter.dev/development/data-and-backend/state-mgmt/ephemeral-vs-app`.

[29] K. C. Dodds. Prop drilling, may 2018. URL `https://kentcdodds.com/blog/prop-drilling`.

[30] M. K. et al. Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, 14, dec 2018. doi: 10.1007/s11334-018-0319-4.

[31] S. Faust. Using google´s flutter framework for the development of a large-scale reference application. 2020.

[32] fireup.pro team. 9 amazing mobile apps built with react native, 2021. URL `https://fireup.pro/blog/9-amazing-mobile-apps-built-with-react-native`.

[33] R. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991. doi: 10.1109/32.87281.

[34] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, Schenley Park Pittsburgh, PA, USA, 1994.

[35] B. Gezici, A. Tarhan, and O. Chouseinoglou. Internal and external quality in the evolution of mobile software: An exploratory study in open-source market. *Information and Software Technology*, 112:178–200, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.04.002. URL `https://www.sciencedirect.com/science/article/pii/S0950584918301290`.

[36] T.-M. Grønli, J. Hansen, G. Ghinea, and M. Younas. Mobile application platform heterogeneity: Android vs windows phone vs ios vs firefox os. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 635–641, 2014. doi: 10.1109/AINA.2014.78.

[37] M. Hasnany. Flutter web support hits the stable milestone, 2021. URL `https://medium.com/flutter/flutter-web-support-hits-the-stable-milestone-d6b84e83b425`.

[38] R. Hat. Stateful vs stateless, mar 2020. URL `https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless`.

[39] L. H. Hoang. State management analyses of the flutter application, nov 2019.

[40] P. Hunt. React: Rethinking best practices – jsconf eu, oct 2013. URL https://youtu.be/x7cQ3mrcKaY.

[41] IEEE. Ieee standard glossary of software engineering terminology. 1990. doi: 10.1109/ieeestd.1983.7435207.

[42] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[43] W. Leler. What's revolutionary about flutter, aug 2017. URL https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514.

[44] S. Liu. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021, 2021. URL https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.

[45] S. Liu. Most used libraries and frameworks among developers, worldwide, as of 2021, 2021. URL https://www.statista.com/statistics/793840/worldwide-developer-survey-most-used-frameworks/.

[46] L. Losoviz. From a single repo, to multi-repos, to monorepo, to multi-monorepo, aug 2021. URL https://css-tricks.com/from-a-single-repo-to-multi-repos-to-monorepo-to-multi-monorepo/.

[47] T. Mackinnon, S. Freeman, and P. Craig. *Endo-Testing: Unit Testing with Mock Objects*, page 287–301. Addison-Wesley Longman Publishing Co., Inc., USA, 2001. ISBN 0201710404. doi: 10.5555/377517.377534.

[48] T. A. Majchrzak, J. Ernsting, and H. Kuchen. Achieving business practicability of model-driven cross-platform apps. *Open Journal of Information Systems (OJIS)*, 2: 3–14, 01 2015.

[49] R. C. Martin. The dependency inversion principle. Technical report, 1996.

[50] S. McConnell. *Code complete: A practical handbook of software construction.* Microsoft Press, 2016.

[51] B. Nystrom. Understanding null safety. jul 2020. URL https://dart.dev/null-safety/understanding-null-safety.

[52] C. Pacheco. Flutter mvvm and clean architecture - part 3: Multi-packages structure, dec 2021. URL `https://cassiuspacheco.com/flutter-mvvm-and-clean-architecture-part-3-multi-packages-structure`.

[53] D. R. Prasanna. *Dependency injection.* Manning, 2009.

[54] G. Salvaneschi, A. Margara, and G. Tamburrelli. Reactive programming: A walk-through. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 953–954, 2015. doi: 10.1109/ICSE.2015.303.

[55] J. Savolainen and V. Myllarniemi. Layered architecture revisited — comparison of research and practice. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, pages 317–320, 2009. doi: 10.1109/WICSA.2009.5290685.

[56] M. Seemann and G. Block. *Dependency injection in .NET.* Manning, 2012.

[57] C. Sells. What's new in flutter 2, 2021. URL `httpshttps://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65`.

[58] S. Shlaer and S. Mellor. Recursive design of an application-independent architecture. *IEEE Software*, 14(1):61–72, 1997. doi: 10.1109/52.566429.

[59] D. Slepnev. State management approaches in flutter, 2020.

[60] T. Sneath. Announcing flutter for windows, 2022. URL `https://medium.com/flutter/announcing-flutter-for-windows-6979d0d01fed`.

[61] P. Soares. Flutter / angulardart – code sharing, better together (dartconf 2018). Google Developers, jan 2018. URL `https://youtu.be/PLHln7wHgPE`.

[62] S. Stoll. In plain english: So what the heck is flutter and why is it a big deal?, may 2018. URL `https://medium.com/flutter-community/in-plain-english-so-what-the-heck-is-flutter-and-why-is-it-a-big-deal-7a6dc926`

[63] M. Sullivan. Flutter: Don't fear the garbage collector, jan 2019. URL `https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30`.

[64] M. Szczepanik. and M. Kędziora. State management and software architecture approaches in cross-platform flutter applications. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 407–414. INSTICC, SciTePress, 2020. ISBN 978-989-758-421-3. doi: 10.5220/0009411604070414.

[65] D. Team. Dart overview, . URL https://dart.dev/overview.

[66] D. Team. Asynchronous programming: futures, async, await, . URL https://dart.dev/codelabs/async-await.

[67] D. Team. Asynchronous programming: Streams, . URL https://dart.dev/tutorials/language/streams.

[68] D. Team. How to use packages, . URL https://dart.dev/guides/packages.

[69] N. team. What is flutter and why use flutter for app development, 2020. URL https://nix-united.com/blog/the-pros-and-cons-of-flutter-in-mobile-application-development/.

[70] V. Team. Top companies using flutter in 2021, 2021. URL https://verygood.ventures/blog/top-companies-using-flutter-2021.

[71] theCodeReaper. Layered architecture pattern in software engineering, aug 2020. URL https://thecodereaper.com/2020/08/22/layered-architecture-pattern-in-software-engineering.

[72] G. Thomas. What is flutter and why you should learn it in 2020, dec 2019. URL https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020.

[73] M. Veng. *Dependency Injection and Mock on Software and Testing.* PhD thesis, 2014. URL http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-226214.

[74] S. Wardrop. Best practices for building scalable flutter applications, dec 2020. URL https://verygood.ventures/blog/scalable-best-practices.

[75] S. Wardrop. Flutter testing: A very good guide [10 insights], feb 2021. URL https://verygood.ventures/blog/guide-to-flutter-testing.

[76] L. Wei, Y. Liu, and S.-C. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 226–237, 2016.

[77] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, dec 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL https://doi.org/10.1145/267580.267590.

A | Appendix - Source Codes

Listing 22: Permission Client

```
1  export 'package:permission_handler/permission_handler.dart'
2      show PermissionStatusGetters, PermissionStatus;
3
4  /// {@template permission_client}
5  /// A client to handle requesting permissions on devices
6  /// {@endtemplate}
7  class PermissionClient {
8    /// {@macro permission_client}
9    const PermissionClient(this._platform);
10
11   final Platform _platform;
12
13   /// Request access to the device's contacts,
14   /// if access hasn't been previously granted
15   Future<PermissionStatus> requestContacts() =>
16     Permission.contacts.request();
17
18   /// Request access to the device's photos,
19   /// if access hasn't been previously granted
20   Future<PermissionStatus> requestPhotos() =>
21     Permission.photos.request();
22
23   /// Request access to the device's storage,
24   /// if access hasn't been previously granted
25   Future<PermissionStatus> requestStorage() =>
26     Permission.storage.request();
27
28   /// Request access to the device's camera,
29   /// if access hasn't been previously granted
30   Future<PermissionStatus> requestCamera() =>
```

```
31      Permission.camera.request();
32
33    /// Request access to the device's bluetooth in iOS 13 and above,
34    Future<PermissionStatus> requestBluetooth() =>
35      Permission.bluetooth.request();
36
37    /// Request access to look for Bluetooth devices (e.g. BLE peripherals)
38    /// in Android(iOS: Nothing)  if access hasn't been previously granted
39    Future<PermissionStatus> requestBluetoothScan() => _platform.isAndroid
40        ? Permission.bluetoothScan.request()
41        : Future.value(PermissionStatus.granted);
42
43    /// Checks the status of Bluetooth Permission.
44    Future<PermissionStatus> isBluetoothGranted() async {
45      if (_platform.isIOS) {
46        return Permission.bluetooth.status;
47      } else {
48        final scanStatus = await Permission.bluetoothScan.status;
49        final locationStatus = await Permission.location.status;
50        if (scanStatus.isGranted && locationStatus.isGranted) {
51          return PermissionStatus.granted;
52        }
53        return PermissionStatus.denied;
54      }
55    }
56
57    /// Request access to the device's location,
58    /// if access hasn't been previously granted
59    Future<PermissionStatus> requestLocation() =>
60      Permission.location.request();
61
62    /// Opens to the app settings page, allowing the user to change previously
63    /// denied permissions
64    ///
65    /// Returns true if the settings could be opened, otherwise false
66    Future<bool> openPermissionSettings() =>
67      openAppSettings();
68
69    /// Checks the device's bluetoothScan and location permissions for Android.
```

```
70    ///
71    /// It performs a no-op for iOS devices.
72    Future<PermissionStatus>
73        checkPlatformSpecificPermissionsForBluetooth() async {
74      if (_platform.isAndroid) {
75        final bluetoothScan = await requestBluetoothScan();
76        final location = await requestLocation();
77        return (bluetoothScan.isGranted && location.isGranted)
78            ? PermissionStatus.granted
79            : PermissionStatus.denied;
80      } else {
81        return Future.value(PermissionStatus.granted);
82      }
83    }
84
85    /// Checks the device's location services is enabled for Android
86    ///
87    /// It performs a no-op for iOS devices.
88    Future<bool> checkLocationServiceStatusForWifi() =>
89      _platform.isAndroid
90        ? Permission.location.serviceStatus.isEnabled
91        : Future.value(true);
92  }
```

Listing 23: Async Behavior Subject

```
1   /// Void callback
2   typedef VoidCallback = void Function();
3
4   /// A special BehaviorSubject that allows emitting
5   /// events from a Future response or another Stream
6   /// of events without blocking the data flow.
7   class AsyncBehaviorSubject<T> {
8     /// Constructs an [AsyncBehaviorSubject].
9     /// Optionally passes handlers to [onStart].
10    AsyncBehaviorSubject({
11     this.onStart,
12     this.onCancel,
13     this.onListen,
14    });
15
```

```
16    /// Called when the [Stream] values are
17    /// retrieved for the first time
18    @protected
19    VoidCallback? onStart;
20
21    /// Called every time a new subscriber starts listening
22    /// to the [Stream]. It continues to broadcast data as long
23    /// as there's at least one subscriber.
24    /// This method can be called multiple times.
25    @protected
26    VoidCallback? onListen;
27
28    /// Called every time the last subscriber stops listening.
29    /// This method can be called multiple times.
30    @protected
31    VoidCallback? onCancel;
32
33    BehaviorSubject<T>? __subject;
34
35    //Inits _subject lazily
36    BehaviorSubject<T> get _subject {
37      if (__subject == null) {
38        __subject = BehaviorSubject<T>(
39          onListen: onListen,
40          onCancel: () {
41            _onCancelControllers();
42            onCancel?.call();
43          },
44        );
45        onStart?.call();
46      }
47      return __subject!;
48    }
49
50    /// Current value from the [Stream]
51    T? get value => _subject.valueOrNull;
52
53    /// Stream controlled by the [AsyncBehaviorSubject] controller
54    Stream<T> get stream => _subject.stream;
```

```
55
56    /// Sends a data event to the [Stream]
57    void add(T value) {
58      _subject.add(value);
59    }
60
61    /// Sends an error to the [Stream]
62    void addError(Object error, [StackTrace? stackTrace]) {
63      _subject.addError(error, stackTrace);
64    }
65
66    /// Adds a value or error from the future response
67    Future<void> addFuture(Future<T> future) async {
68      try {
69        final value = await future;
70        if (_subject.isClosed) return;
71        add(value);
72      } catch (e, s) {
73        if (_subject.isClosed) return;
74        addError(e, s);
75      }
76    }
77
78    /// Adds values emitted by the provided [Stream].
79    /// If blockEvents is set to true, it prevents
80    /// using add/addError until the stream is closed.
81    Future<void> addStream(
82      Stream<T> source, {
83      bool? cancelOnError,
84      bool blockEvents = false,
85    }) {
86      if (blockEvents) {
87        return _subject.addStream(
88          source,
89          cancelOnError: cancelOnError,
90        );
91      } else {
92        return _addStreamLazily(
93          source,
```

```
94          cancelOnError: cancelOnError,
95        );
96      }
97    }
98
99    /// Adds values emitted by the provided [Stream].
100   /// It will stop listening to the stream if there
101   /// is no more active subscribers.
102   Future<void> addStreamUntilOnCancel(
103     Stream<T> source, {
104     bool? cancelOnError,
105   }) {
106     return _addStreamLazily(
107       source,
108       cancelOnError: cancelOnError,
109       cancelOnSubjectCancel: true,
110     );
111   }
112
113   Future<void> _addStreamLazily(
114     Stream<T> source, {
115     bool? cancelOnError,
116     bool? cancelOnSubjectCancel,
117   }) async {
118     final completer = Completer<void>();
119     var isOnDoneCalled = false;
120     void complete() {
121       if (!isOnDoneCalled) {
122         isOnDoneCalled = true;
123
124         completer.complete();
125       }
126     }
127
128     final controller = source.listen(
129       add,
130       onError: (Object e, StackTrace s) {
131         addError(e, s);
132
```

```
133          if (identical(cancelOnError, true)) {
134            complete();
135          }
136        },
137        onDone: complete,
138        cancelOnError: cancelOnError,
139      );
140      if (identical(cancelOnSubjectCancel, true)) {
141        _cancelableStreamControllers.add(controller);
142      } else {
143        _streamControllers.add(controller);
144      }
145
146      return completer.future;
147    }
148
149    void _onCancelControllers() {
150      for (final controller in _cancelableStreamControllers) {
151        controller.cancel();
152      }
153      _cancelableStreamControllers.clear();
154    }
155
156    /// Subscriptions of available streams
157    final Set<StreamSubscription<T>> _streamControllers = {};
158
159    /// Subscriptions of available streams that are
160    /// canceled on [onCancel]
161    final Set<StreamSubscription<T>> _cancelableStreamControllers = {};
162
163    /// Closes the stream
164    @mustCallSuper
165    void close() {
166      __subject?.close();
167      for (final controller in _cancelableStreamControllers) {
168        controller.cancel();
169      }
170      for (final controller in _streamControllers) {
171        controller.cancel();
```

```
172       }
173     }
174   }
175
176   /// Extension methods for [AsyncBehaviorSubject]
177   /// with [Map] values by id
178   extension MapSubjectExtension<Key, Value>
179       on AsyncBehaviorSubject<Map<Key, Value>> {
180     /// Updates a value from the map.
181     ///
182     /// The [Stream] adds a new event with a copy of the [Map]
183     /// with this new [Value] updated.
184     void updateItem(Key key, Value? item) {
185       if (item == null) {
186         return removeItem(key);
187       }
188       final newValue = Map<Key, Value>.from(value ?? <Key, Value>{});
189       newValue[key] = item;
190       add(newValue);
191     }
192
193     /// Removes a [Value] from the [Map].
194     ///
195     /// The stream adds a new event with a copy of
196     /// the [Map] without the entry of this [Key]
197     void removeItem(Key key) {
198       final newValue = Map<Key, Value>.from(
199         value ?? <Key, Value>{})..remove(key,
200       );
201       add(newValue);
202     }
203   }
```

Listing 24: Home Details Bloc

```
1   part 'home_details_event.dart';
2   part 'home_details_state.dart';
3
4   class HomeDetailsBloc
5   extends Bloc<HomeDetailsEvent, HomeDetailsState> {
6     HomeDetailsBloc({
```

```
7       required HomeRepository homeRepository,
8       required String homeId,
9     })  : _homeRepository = homeRepository,
10          _homeId = homeId,
11          super(const HomeDetailsState()) {
12      on<HomeDetailsHomeLoaded>(_homeLoaded);
13      on<HomeDetailsFetchRequested>(_fetchRequested);
14    }
15
16    final HomeRepository _homeRepository;
17    final String _homeId;
18    StreamSubscription<Map<String, Home>>? _homeSubscription;
19    StreamSubscription<HomeAvatarUpdatedPayload>? _avatarSubscription;
20
21    FutureOr<void> _fetchRequested(
22      HomeDetailsFetchRequested event,
23      Emitter<HomeDetailsState> emit,
24    ) async {
25      _homeSubscription = _homeRepository.homes.listen((homes) {
26        final home = homes.values.firstWhereOrNull(
27          (home) => home.id == _homeId,
28        );
29        if (home != null) {
30          add(HomeDetailsHomeLoaded(home: home));
31        }
32      });
33
34      await emit.forEach<HomeAvatarUpdatedPayload>(
35        _homeRepository.homeAvatarUpdated(homeId: _homeId),
36        onData: (payload) => state.copyWith(
37          home: state.home?.copyWith(avatar: payload.avatar),
38        ),
39      );
40    }
41
42    FutureOr<void> _homeLoaded(
43      HomeDetailsHomeLoaded event,
44      Emitter<HomeDetailsState> emit,
45    ) async {
```

```
46      try {
47        final homeInfo = await _homeRepository.getHomeInfo(_homeId);
48        final avatar = await _homeRepository.getAvatarUrls(_homeId);
49        final home = event.home.copyWith(homeInfo: homeInfo, avatar: avatar);
50        emit(state.copyWith(home: home, status: HomeDetailsStatus.loaded));
51      } catch (e) {
52        emit(state.copyWith(status: HomeDetailsStatus.loadingFailed));
53      }
54    }
55
56    @override
57    Future<void> close() {
58      _homeSubscription?.cancel();
59      _avatarSubscription?.cancel();
60      return super.close();
61    }
62  }
```

Listing 25: Home Details Page

```
1   class HomeDetailsPage extends StatelessWidget {
2     const HomeDetailsPage({Key? key}) : super(key: key);
3
4     static const routeName = '/home-details';
5
6     static Route route() {
7       return MaterialPageRoute<void>(
8         settings: const RouteSettings(name: routeName),
9         builder: (context) => MultiBlocProvider(
10          providers: [
11            BlocProvider(
12              create: (_) => DeleteHomeBloc(
13                homeId: context.read<HomeBloc>().homeId,
14                homeRepository: context.read<HomeRepository>(),
15              ),
16            ),
17            BlocProvider(
18              create: (context) => HomeDetailsBloc(
19                homeRepository: context.read<HomeRepository>(),
20                homeId: context.read<HomeBloc>().homeId,
21              )..add(HomeDetailsFetchRequested()),
```

```
22              child: const HomeDetailsPage(),
23            ),
24          ],
25          child: const HomeDetailsPage(),
26        ),
27      );
28    }
29
30    @override
31    Widget build(BuildContext context) {
32      final l10n = context.l10n;
33      return CustomLoadingOverlay(
34        loadingWhen: (context) {
35          return context.select(
36            (DeleteHomeBloc b) =>
37              b.state.status == DeleteHomeStatus.inProgress,
38          );
39        },
40        child: Scaffold(
41          appBar: CustomAppBar(
42            title: l10n.homeDetailsPageTitle,
43            color: CustomColors.lightWater,
44            actions: [
45              Padding(
46                padding: const EdgeInsets.only(right: 20),
47                child: IconButton(
48                  key: const Key(
49                    'homeDetails_customAppBar_options_iconButton',
50                  ),
51                  icon: CustomIcons.icDotsThreeOutline(size: 25),
52                  onPressed: () =>
53                    openHomeOptionsBottomSheet(context),
54                ),
55              ),
56            ],
57          ),
58          body: const HomeDetailsView(),
59        ),
60      );
```

```
61   }
62 }
```

Listing 26: Edit Device Event

```
1  part of 'edit_device_bloc.dart';
2
3  abstract class EditDeviceEvent extends Equatable {
4    const EditDeviceEvent();
5
6    @override
7    List<Object?> get props => [];
8  }
9
10 class EditDeviceAreaRequested extends EditDeviceEvent {}
11
12 class EditDeviceNameUpdated extends EditDeviceEvent {
13   const EditDeviceNameUpdated(this.name);
14
15   final String name;
16
17   @override
18   List<Object> get props => [name];
19 }
20
21 class EditDeviceAreaUpdated extends EditDeviceEvent {
22   const EditDeviceAreaUpdated(this.area);
23
24   final Area? area;
25
26   @override
27   List<Object?> get props => [area];
28 }
29
30 class EditDeviceSaveRequested extends EditDeviceEvent {}
```

Listing 27: Edit Device State

```
1  part of 'edit_device_bloc.dart';
2
3  class EditDeviceState extends Equatable {
4    const EditDeviceState({
```

```
5      required this.originalDevice,
6      required this.editedDevice,
7      this.saveStatus = EditDeviceSaveStatus.unknown,
8      this.areas,
9    });
10
11   final EditedDevice originalDevice;
12   final EditedDevice editedDevice;
13   final EditDeviceSaveStatus saveStatus;
14   final List<Area>? areas;
15
16   bool get hasChanged =>
17     originalDevice != editedDevice;
18   bool get canSave =>
19     hasChanged && editedDevice.deviceName.valid;
20
21   @override
22   List<Object?> get props => [
23         originalDevice,
24         editedDevice,
25         saveStatus,
26         areas,
27       ];
28
29   EditDeviceState copyWith({
30     EditedDevice? originalDevice,
31     EditedDevice? editedDevice,
32     EditDeviceSaveStatus? saveStatus,
33     List<Area>? areas,
34   }) {
35     return EditDeviceState(
36       originalDevice: originalDevice ?? this.originalDevice,
37       editedDevice: editedDevice ?? this.editedDevice,
38       saveStatus: saveStatus ?? this.saveStatus,
39       areas: areas ?? this.areas,
40     );
41   }
42 }
43
```

```
44 enum EditDeviceSaveStatus {
45   unknown,
46   loading,
47   success,
48   failed,
49 }
```

Listing 28: Edit Device Bloc

```
1  part 'edit_device_event.dart';
2  part 'edit_device_state.dart';
3
4  class EditDeviceBloc extends
5  Bloc<EditDeviceEvent, EditDeviceState> {
6    EditDeviceBloc({
7      required String homeId,
8      required DeviceRepository deviceRepository,
9      required AreaRepository areaRepository,
10     required this.device,
11   })  : _homeId = homeId,
12         _deviceRepository = deviceRepository,
13         _areaRepository = areaRepository,
14         super(
15           EditDeviceState(
16             editedDevice: EditedDevice.fromDevice(device),
17             originalDevice: EditedDevice.fromDevice(device),
18           ),
19         ) {
20     on<EditDeviceNameUpdated>(_deviceNameUpdated);
21     on<EditDeviceAreaUpdated>(_deviceAreaUpdated);
22     on<EditDeviceSaveRequested>(_saveRequested);
23     on<EditDeviceAreaRequested>(_areasRequested);
24   }
25
26   final Device device;
27   final String _homeId;
28   final DeviceRepository _deviceRepository;
29   final AreaRepository _areaRepository;
30
31   FutureOr<void> _areasRequested(
32     EditDeviceAreaRequested event,
```

```
33      Emitter<EditDeviceState> emit,
34    ) async { ... }
35
36    FutureOr<void> _deviceNameUpdated(
37      EditDeviceNameUpdated event,
38      Emitter<EditDeviceState> emit,
39    ) { ... }
40
41    FutureOr<void> _deviceAreaUpdated(
42      EditDeviceAreaUpdated event,
43      Emitter<EditDeviceState> emit,
44    ) { ... }
45
46    Future<void> _saveRequested(
47      EditDeviceSaveRequested event,
48      Emitter<EditDeviceState> emit,
49    ) async { ... }
50  }
```

Listing 29: Edit Device View

```
1   class EditDeviceView extends StatelessWidget {
2     const EditDeviceView({Key? key}) : super(key: key);
3
4     @override
5     Widget build(BuildContext context) {
6       final l10n = context.l10n;
7
8       return BlocListener<EditDeviceBloc, EditDeviceState>(
9         listenWhen: (previous, current) =>
10            current.saveStatus == EditDeviceSaveStatus.failed ||
11            current.saveStatus == EditDeviceSaveStatus.success,
12        listener: (context, state) {
13          if (state.saveStatus == EditDeviceSaveStatus.failed) {
14            CustomSnackBar.error(
15              l10n.savingDeviceError,
16              key: const Key('editDeviceView_save_error_snackBar'),
17            ).show(context: context);
18          }
19          if (state.saveStatus == EditDeviceSaveStatus.success) {
20            Navigator.of(context).pop();
```

```
21          }
22        },
23          child: Scaffold(
24            appBar: CustomAppBar( ... ),
25            body: const Padding(
26              padding: EdgeInsets.symmetric(
27                horizontal: CustomSpacing.lg,
28              ),
29              child: ScrollableColumn(
30                children: [
31                  _DeviceNameTextField(),
32                  SizedBox(height: CustomSpacing.lg),
33                  _AreaDropdownButton(),
34                ],
35              ),
36            ),
37          ),
38      );
39    }
40  }
```

Listing 30: Device Name Text Field

```
1  class _DeviceNameTextField extends StatelessWidget {
2    const _DeviceNameTextField({Key? key}) : super(key: key);
3
4    @override
5    Widget build(BuildContext context) {
6      final deviceName = context.select(
7        (EditDeviceBloc bloc) =>
8          bloc.state.editedDevice.deviceName,
9      );
10
11     final l10n = context.l10n;
12     return CustomTextFormField(
13       key: const Key('editDeviceView_nameInput_textField'),
14       labelText: l10n.editDeviceNameLabelText,
15       initialValue: deviceName.value,
16       maxLength: 50,
17       onChanged: (name) =>
18         context.read<EditDeviceBloc>().add(
```

```
19              EditDeviceNameUpdated(name),
20            ),
21        keyboardType: TextInputType.text,
22        errorText: deviceName.invalid ?
23          l10n.validationAddDeviceNameError
24          : null,
25      );
26    }
27  }
```

Listing 31: Plug Pairing Body Switcher

```
1  class PlugPairingBodySwitcher extends StatelessWidget {
2    const PlugPairingBodySwitcher({Key? key})
3      : super(key: key);
4
5    @override
6    Widget build(BuildContext context) {
7      final l10n = context.l10n;
8
9      return BlocConsumer<PlugPairingBloc, PlugPairingState>(
10       listener: (context, state) {
11         if (state.cableCheckFailed) {
12           Navigator.of(context).pop();
13           CustomSnackBar.error(
14             l10n.plugPairingCableError,
15             key: const Key('plugPairingBody_error_abraSnackBar'),
16           ).show(context: context);
17         }
18         if (state.wasPreviouslyConnected) {
19           Navigator.of(context).pop();
20           showDeviceAlreadyConnectedDialog(context);
21         }
22         if (state.connectedSuccessfully) {
23           context.read<AddDeviceBloc>().add(
24                 AddDevicePaired(deviceId: state.pairedDeviceId!),
25               );
26         }
27       },
28       buildWhen: (previous, current)
29         => previous.step != current.step,
```

```
30        builder: (context, state) {
31          switch (state.step) {
32            case PlugPairingStep.checkingCable:
33              return const Center(
34                child: CircularProgressIndicator(),
35              );
36            case PlugPairingStep.instructions:
37              return const PlugInstructionsBody();
38            case PlugPairingStep.connecting:
39              return const PairingDeviceBody();
40            case PlugPairingStep.failed:
41              return const PlugPairingFailedBody();
42          }
43        },
44      );
45    }
46  }
```

Listing 32: Alarm Resource - alarms group

```
1   group('alarms', () {
2         group('throws', () {
3           test('AlarmResourceException on AmplifyException', () {
4             when(
5               () => graphQLCategory.query(
6                 request: any(
7                   named: 'request',
8                   that: isA<GraphQLRequest>().having(
9                     (request) => request.document,
10                    'document',
11                    AlarmQuery.alarms,
12                  ),
13                ),
14              ),
15            ).thenThrow(AmplifyException(''));
16
17            expect(
18              () => alarmResource.alarms(''),
19              throwsA(isA<AlarmResourceException>()),
20            );
21          });
```

```
22
23        test('JsonParsingException on invalid json parsing', () {
24          when(
25            () => graphQLCategory.query(
26              request: any(
27                named: 'request',
28                that: isA<GraphQLRequest>().having(
29                  (request) => request.document,
30                  'document',
31                  AlarmQuery.alarms,
32                ),
33              ),
34            ),
35          ).thenAnswer(
36            (_) async => alarmsMalformedResponse,
37          );
38
39          expect(
40            () => alarmResource.alarms(''),
41            throwsA(isA<JsonParsingException>()),
42          );
43        });
44      });
45
46      test('returns correct AlarmCollection', () async {
47        when(
48          () => graphQLCategory.query(
49            request: any(
50              named: 'request',
51              that: isA<GraphQLRequest>().having(
52                (request) => request.document,
53                'document',
54                AlarmQuery.alarms,
55              ),
56            ),
57          ),
58        ).thenAnswer((_) async => alarmsValidResponse);
59        final result = await alarmResource.alarms('');
60        final responseJson =
```

```
61            json.decode(alarmsValidResponse)
62            as Map<String, dynamic>;
63        final alarmCollectionPayload = AlarmCollection
64          .fromJson(
65            responseJson['alarms'] as Map<String, dynamic>,
66          );
67
68        expect(
69          result,
70          equals(alarmCollectionPayload),
71        );
72      });
73    });
```

Listing 33: Alarm Repository

```
1  void main() {
2    group('AlarmRepository', () {
3      late ApiClient apiClient;
4      late AlarmResource alarmResource;
5      late AlarmRepository alarmRepository;
6
7      setUp(() {
8        apiClient = MockApiClient();
9        alarmResource = MockAlarmResource();
10       alarmRepository = AlarmRepository(apiClient);
11       when(() => apiClient.alarmResource).thenReturn(
12         alarmResource,
13       );
14       when(
15         () => alarmResource.alarms(any()),
16       ).thenAnswer(
17         (_) async => FakeAlarmCollection(),
18       );
19       when(
20         () => alarmResource.alarmUpdated(
21           homeId: any(named: 'homeId'),
22         ),
23       ).thenAnswer(
24         (_) => Stream.empty(),
25       );
```

```
26     });
27   });
28 }
```

Listing 34: Alarm Repository - alarms group

```
1  group('alarms', () {
2        group('throws', () {
3          test(
4            ' AlarmResourceFailure on '
5            'alarms AlarmResourceException',
6            () {
7              when(() => alarmResource.alarms(any()))
8                  .thenThrow(AlarmResourceException(''));
9
10             expect(
11               alarmRepository.alarms('homeId'),
12               emitsError(isA<AlarmResourceFailure>()),
13             );
14           },
15         );
16
17         test(
18           'throws AlarmResourceFailure on '
19           'alarmUpdated AlarmResourceFailure',
20           () {
21             when(
22               () => alarmResource.alarmUpdated(
23                 homeId: any(named: 'homeId'),
24               ),
25             ).thenAnswer(
26               (_) => Stream.error(AlarmResourceFailure(null)),
27             );
28             expect(
29               alarmRepository.alarms('homeId'),
30               emitsInOrder(<dynamic>[
31                 {alarm.id: alarm},
32                 emitsError(isA<AlarmResourceFailure>()),
33               ]),
34             );
35           },
```

```
36          );
37        });
38
39        test('fetches data', () {
40          expect(
41            alarmRepository.alarms('homeId'),
42            emits({alarm.id: alarm}),
43          );
44        });
45
46        test('updates data', () {
47          when(
48            () => alarmResource.alarmUpdated(
49              homeId: any(named: 'homeId'),
50            ),
51          ).thenAnswer(
52            (_) => Stream.value(
53              FakeAlarmUpdatedPayload(),
54            ),
55          );
56          expect(
57            alarmRepository.alarms('homeId'),
58            emitsInOrder(<dynamic>[
59              {alarm.id: alarm},
60              {alarm.id: updatedAlarm},
61            ]),
62          );
63        });
64      });
```

Listing 35: Water Leak Alarm Bloc - Mocks and Fakes

```
1  class MockDeviceRepository extends Mock
2    implements DeviceRepository {}
3
4  class MockAlarmRepository extends Mock
5    implements AlarmRepository {}
6
7  class FakeAlarmResolvePayload extends Fake
8    implements AlarmResolvePayload {}
9
```

```
10 void main() {
11   group('WaterLeakAlarmAlarmBloc', () {
12     late DeviceRepository deviceRepository;
13     late AlarmRepository alarmRepository;
14
15     setUp(() {
16       deviceRepository = MockDeviceRepository();
17       alarmRepository = MockAlarmRepository();
18     });
19   });
20 }
```

Listing 36: Water Leak Alarm Bloc - initial state

```
1  test(
2      'initial state is the default WaterLeakAlarmState',
3      () {
4        expect(
5        WaterLeakAlarmBloc(
6          homeId: 'homeId',
7          deviceRepository: deviceRepository,
8          alarmRepository: alarmRepository,
9        ).state,
10       const WaterLeakAlarmState(),
11     );
12   },
13 );
```

Listing 37: Water Leak Alarm Ok Status Banner Closed group

```
1  group('WaterLeakAlarmOkStatusBannerClosed', () {
2    blocTest<WaterLeakAlarmBloc, WaterLeakAlarmState>(
3      'dismiss dialog',
4      build: () => WaterLeakAlarmBloc(
5        homeId: 'homeId',
6        deviceRepository: deviceRepository,
7        alarmRepository: alarmRepository,
8      ),
9      seed: () => WaterLeakAlarmState(
10       displayOkStatusBanner: true,
11     ),
12     act: (bloc) => bloc.add(
```

```
13          const WaterLeakAlarmOkStatusBannerClosed(),
14        ),
15        expect: () => <WaterLeakAlarmState>[
16          WaterLeakAlarmState(),
17        ],
18      );
19  });
```

Listing 38: Top Page-View Pattern

```
1   class App extends StatelessWidget {
2     const App({
3       Key? key,
4       required AppConfigRepository appConfigRepository,
5       required AppMonitoringRepository appMonitoringRepository,
6       required ConnectivityRepository connectivityRepository,
7       required AuthenticationRepository authenticationRepository,
8       required bool isUserAuthenticated,
9       required AccountRepository accountRepository,
10      required LocalPhotosRepository localPhotosRepository,
11      required BluetoothRepository bluetoothRepository,
12      required HomeRepository homeRepository,
13      required WifiInfoRepository wifiInfoRepository,
14      required SmartHomeRepository smartHomeRepository,
15      required AreaRepositoryResolver areaRepositoryResolver,
16      required AlarmRepository alarmRepository,
17      required HubRepositoryResolver hubRepositoryResolver,
18      required DeviceRepositoryResolver deviceRepositoryResolver,
19      required HomeMemberRepositoryResolver homeMemberRepositoryResolver,
20      required ContactsRepository contactsRepository,
21    })  : _appConfigRepository = appConfigRepository,
22          _appMonitoringRepository = appMonitoringRepository,
23          _connectivityRepository = connectivityRepository,
24          _authenticationRepository = authenticationRepository,
25          _isUserAuthenticated = isUserAuthenticated,
26          _accountRepository = accountRepository,
27          _localPhotosRepository = localPhotosRepository,
28          _bluetoothRepository = bluetoothRepository,
29          _homeRepository = homeRepository,
30          _wifiInfoRepository = wifiInfoRepository,
31          _smartHomeRepository = smartHomeRepository,
```

```
32          _alarmRepository = alarmRepository,
33          _deviceRepositoryResolver = deviceRepositoryResolver,
34          _hubRepositoryResolver = hubRepositoryResolver,
35          _areaRepositoryResolver = areaRepositoryResolver,
36          _homeMemberRepositoryResolver = homeMemberRepositoryResolver,
37          _contactsRepository = contactsRepository,
38          super(key: key);
39
40    final AppConfigRepository _appConfigRepository;
41    final AppMonitoringRepository _appMonitoringRepository;
42    final ConnectivityRepository _connectivityRepository;
43    final AuthenticationRepository _authenticationRepository;
44    final bool _isUserAuthenticated;
45    final AccountRepository _accountRepository;
46    final LocalPhotosRepository _localPhotosRepository;
47    final BluetoothRepository _bluetoothRepository;
48    final HomeRepository _homeRepository;
49    final WifiInfoRepository _wifiInfoRepository;
50    final SmartHomeRepository _smartHomeRepository;
51    final AlarmRepository _alarmRepository;
52    final ContactsRepository _contactsRepository;
53
54    final DeviceRepositoryResolver _deviceRepositoryResolver;
55    final HubRepositoryResolver _hubRepositoryResolver;
56    final AreaRepositoryResolver _areaRepositoryResolver;
57    final HomeMemberRepositoryResolver _homeMemberRepositoryResolver;
58
59    @override
60    Widget build(BuildContext context) {
61      return MultiRepositoryProvider(
62        providers: [
63          RepositoryProvider.value(
64            value: _appConfigRepository,
65          ),
66          RepositoryProvider.value(
67            value: _connectivityRepository,
68          ),
69          RepositoryProvider.value(
70            value: _authenticationRepository,
```

```
71          ),
72          RepositoryProvider.value(
73            value: _accountRepository,
74          ),
75          RepositoryProvider.value(
76            value: _localPhotosRepository,
77          ),
78          RepositoryProvider.value(
79            value: _bluetoothRepository,
80          ),
81          RepositoryProvider.value(
82            value: _wifiInfoRepository,
83          ),
84          RepositoryProvider.value(
85            value: _homeRepository,
86          ),
87          RepositoryProvider.value(
88            value: _smartHomeRepository,
89          ),
90          RepositoryProvider.value(
91            value: _alarmRepository,
92          ),
93          RepositoryProvider.value(
94            value: _contactsRepository,
95          ),
96          RepositoryProvider.value(
97            value: _deviceRepositoryResolver,
98          ),
99          RepositoryProvider.value(
100           value: _hubRepositoryResolver,
101         ),
102         RepositoryProvider.value(
103           value: _areaRepositoryResolver,
104         ),
105         RepositoryProvider.value(
106           value: _homeMemberRepositoryResolver,
107         ),
108       ],
109       child: MultiBlocProvider(
```

```
110          providers: [
111            BlocProvider(
112              create: (_) => AppBloc(
113                appConfigRepository: _appConfigRepository,
114                authenticationRepository: _authenticationRepository,
115                isAuthenticated: _isUserAuthenticated,
116              ),
117            ),
118            BlocProvider(
119              create: (_) => AppMonitoringBloc(
120                authenticationRepository: _authenticationRepository,
121                accountRepository: _accountRepository,
122                appMonitoringRepository: _appMonitoringRepository,
123                isAuthenticated: _isUserAuthenticated,
124                shouldTrackData: kReleaseMode,
125              ),
126              lazy: false,
127            ),
128            BlocProvider(
129              create: (_) =>
130                AppStoreReviewBloc(isAppStoreReviewAvailable),
131            ),
132            BlocProvider(create: (_) => ThemeModeBloc()),
133            BlocProvider(
134              create: (_) => HomesBloc(_homeRepository)
135                ..add(
136                  HomesFetchRequested(),
137                ),
138            ),
139            BlocProvider(create: (_) => HomeSelectionBloc())
140          ],
141          child: const AppView(),
142        ),
143      );
144    }
145  }
```

Listing 39: App Tester

```
1  extension AppTester on WidgetTester {
2    Future<void> pumpApp(
```

```
3     Widget widgetUnderTest, {
4     AppConfigRepository? appConfigRepository,
5     AppBloc? appBloc,
6     AppStoreReviewBloc? appStoreReviewBloc,
7     ConnectivityRepository? connectivityRepository,
8     TargetPlatform? platform,
9     ThemeModeBloc? themeModeBloc,
10    AuthenticationRepository? authenticationRepository,
11    AccountRepository? accountRepository,
12    HubRepository? hubRepository,
13    BluetoothRepository? bluetoothRepository,
14    HomesBloc? homesBloc,
15    HomeSelectionBloc? homeSelectionBloc,
16    HomeBloc? homeBloc,
17    DeleteHomeMemberBloc? deleteHomeMemberBloc,
18    WifiInfoRepository? wifiInfoRepository,
19    HomeRepository? homeRepository,
20    DeviceRepository? deviceRepository,
21    SmartHomeRepository? smartHomeRepository,
22    LocalPhotosRepository? localPhotosRepository,
23    AreaRepository? areaRepository,
24    AreaRepositoryResolver? areaRepositoryResolver,
25    AlarmRepository? alarmRepository,
26    DeviceRepositoryResolver? deviceRepositoryResolver,
27    HubRepositoryResolver? hubRepositoryResolver,
28    HomeMemberRepository? homeMemberRepository,
29    HomeMemberRepositoryResolver? homeMemberRepositoryResolver,
30    ContactsRepository? contactsRepository,
31  }) async {
32    registerFallbackValues();
33    await pumpWidget(
34      MultiRepositoryProvider(
35        providers: [
36          RepositoryProvider.value(
37            value: appConfigRepository
38              ?? MockAppConfigRepository(),
39          ),
40          RepositoryProvider.value(
41            value: connectivityRepository
```

```
42                  ?? MockConnectivityRepository(),
43              ),
44              RepositoryProvider.value(
45                value: authenticationRepository
46                  ?? MockAuthenticationRepository(),
47              ),
48              RepositoryProvider.value(
49                value: accountRepository
50                  ?? MockAccountRepository(),
51              ),
52              RepositoryProvider.value(
53                value: bluetoothRepository
54                  ?? MockBluetoothRepository(),
55              ),
56              RepositoryProvider.value(
57                value: wifiInfoRepository
58                  ?? MockWifiInfoRepository(),
59              ),
60              RepositoryProvider.value(
61                value: homeRepository
62                  ?? MockHomeRepository(),
63              ),
64              RepositoryProvider.value(
65                value: smartHomeRepository
66                  ?? MockSmartHomeRepository(),
67              ),
68              RepositoryProvider.value(
69                value: localPhotosRepository
70                  ?? MockLocalPhotosRepository(),
71              ),
72              RepositoryProvider.value(
73                value: areaRepository
74                  ?? MockAreaRepository(),
75              ),
76              RepositoryProvider.value(
77                value: alarmRepository
78                  ?? MockAlarmRepository(),
79              ),
80              RepositoryProvider.value(
```

```
81              value: deviceRepositoryResolver
82                ?? MockDeviceRepositoryResolver(),
83            ),
84            RepositoryProvider.value(
85              value: deviceRepository
86                ?? MockDeviceRepository(),
87            ),
88            RepositoryProvider.value(
89              value: hubRepositoryResolver
90                ?? MockHubRepositoryResolver(),
91            ),
92            RepositoryProvider.value(
93              value: areaRepositoryResolver
94                ?? MockAreaRepositoryResolver(),
95            ),
96            RepositoryProvider.value(
97              value: homeMemberRepositoryResolver ??
98                  MockHomeMemberRepositoryResolver(),
99            ),
100           RepositoryProvider.value(
101             value: hubRepository ?? MockHubRepository(),
102           ),
103           RepositoryProvider.value(
104             value: homeMemberRepositoryResolver ??
105                 MockHomeMemberRepositoryResolver(),
106           ),
107           RepositoryProvider.value(
108             value: homeMemberRepository
109               ?? MockHomeMemberRepository(),
110           ),
111           RepositoryProvider.value(
112             value: contactsRepository
113               ?? MockContactsRepository(),
114           ),
115         ],
116         child: MultiBlocProvider(
117           providers: [
118             BlocProvider.value(
119               value: appBloc ?? MockAppBloc(),
```

```
120                ),
121                BlocProvider.value(
122                  value: themeModeBloc
123                  ?? MockThemeModeBloc(),
124                ),
125                BlocProvider.value(
126                  value: appStoreReviewBloc
127                  ?? MockAppStoreReviewBloc(),
128                ),
129                BlocProvider.value(
130                  value: homesBloc
131                  ?? MockHomesBloc(),
132                ),
133                BlocProvider.value(
134                  value: homeSelectionBloc
135                  ?? MockHomeSelectionBloc(),
136                ),
137                BlocProvider.value(
138                  value: homeBloc ?? _MockHomeBloc(),
139                ),
140              ],
141              child: MaterialApp(
142                title: 'App',
143                localizationsDelegates: const [
144                  AppLocalizations.delegate,
145                  GlobalMaterialLocalizations.delegate,
146                  GlobalWidgetsLocalizations.delegate,
147                  GlobalCupertinoLocalizations.delegate,
148                ],
149                home: Theme(
150                  data: ThemeData(platform: platform),
151                  child: Scaffold(body: widgetUnderTest),
152                ),
153              ),
154            ),
155          ),
156        );
157        await pump();
158      }
```

```
159 }
```

B | Appendix - App Screenshots

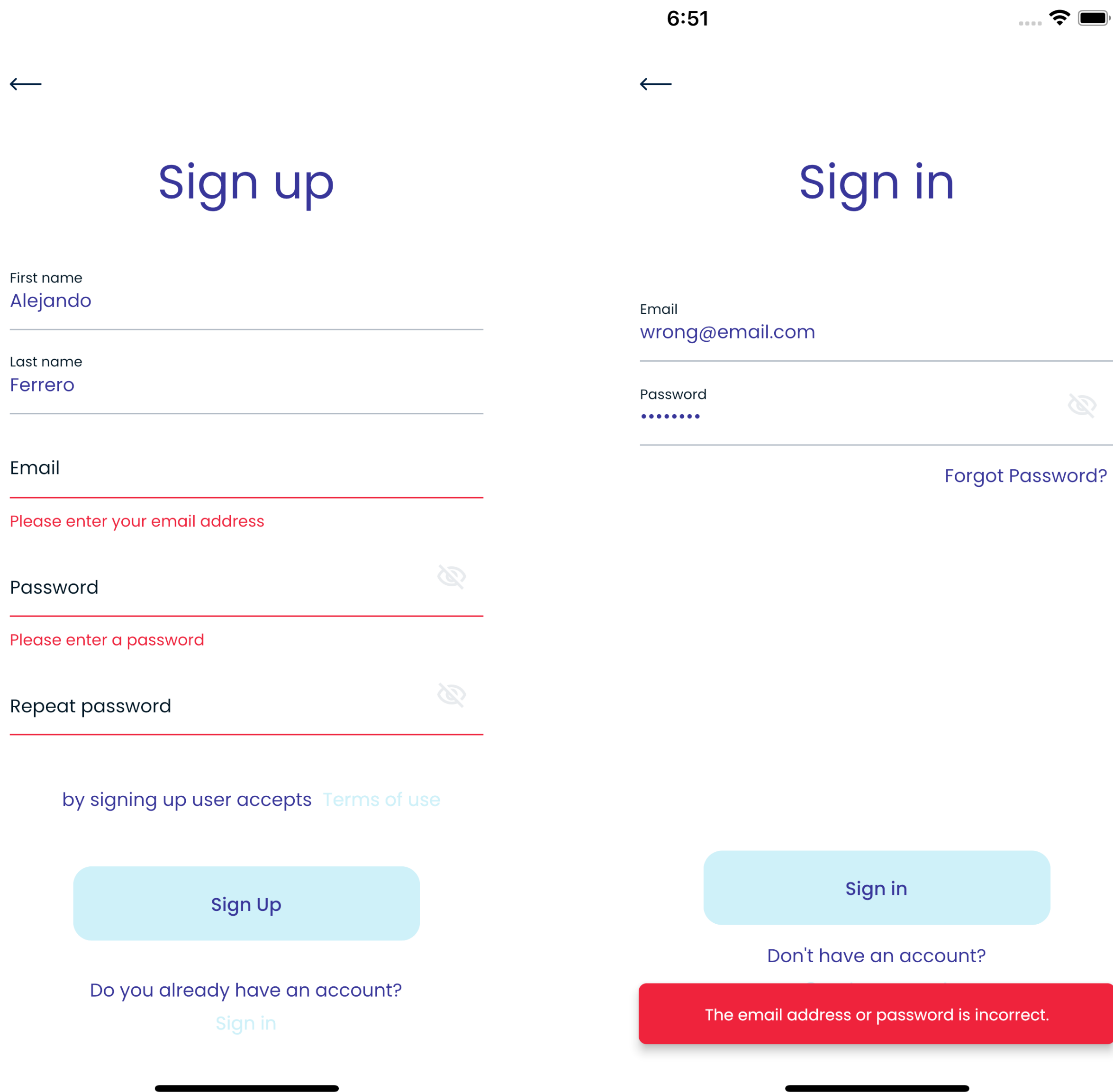

Figure B.1: Authentication Screens

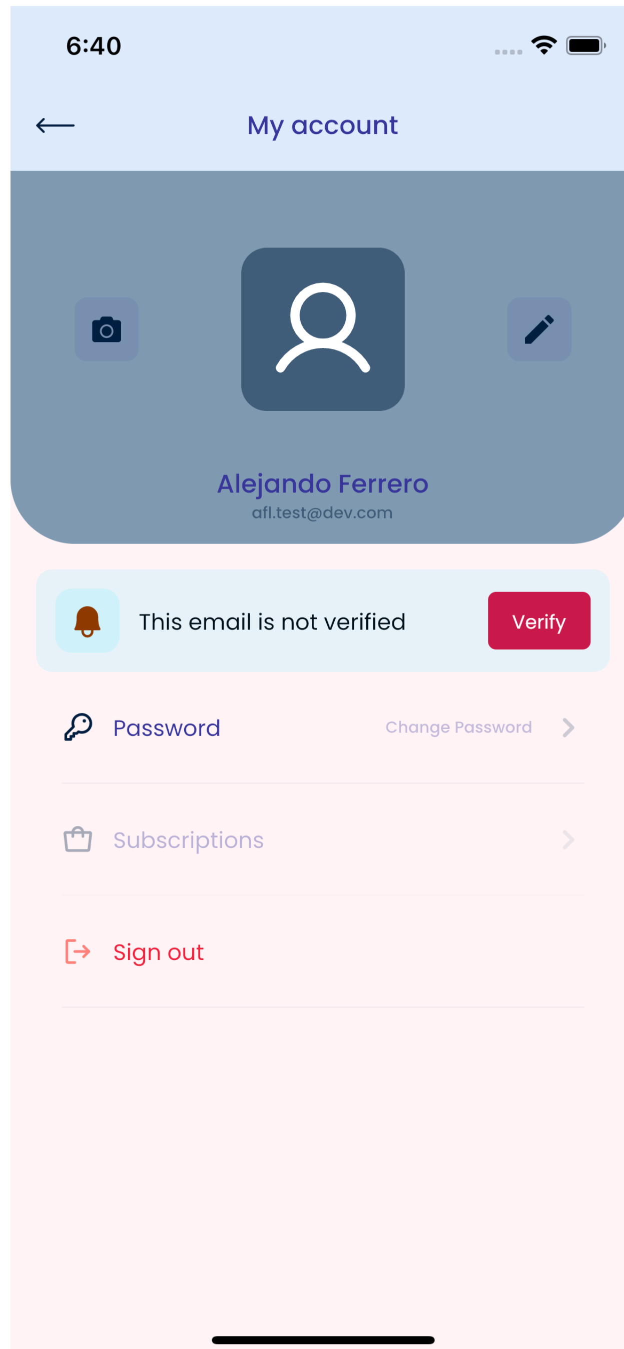

Figure B.2: Account Details Screen

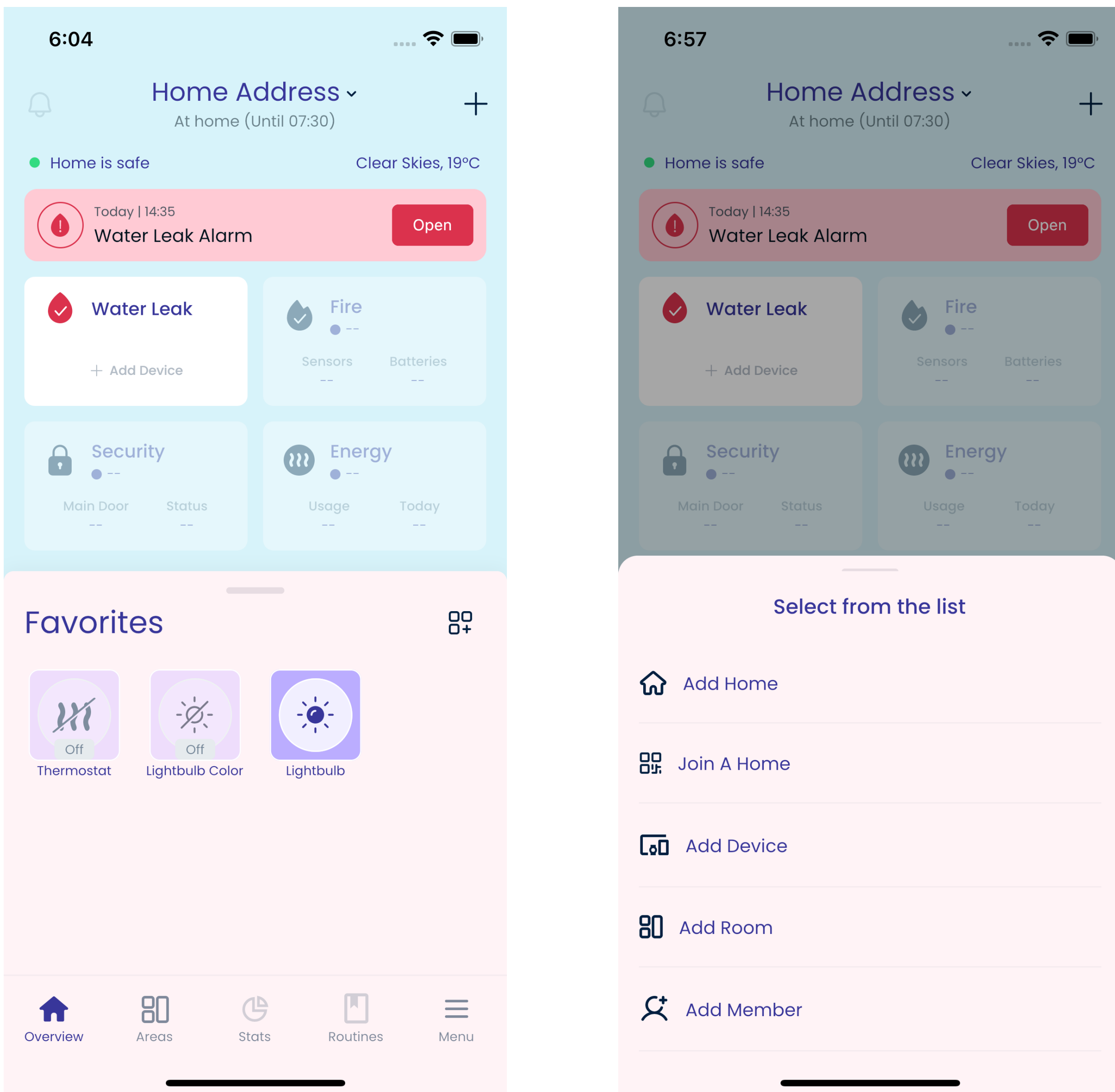

Figure B.3: Overview Screens

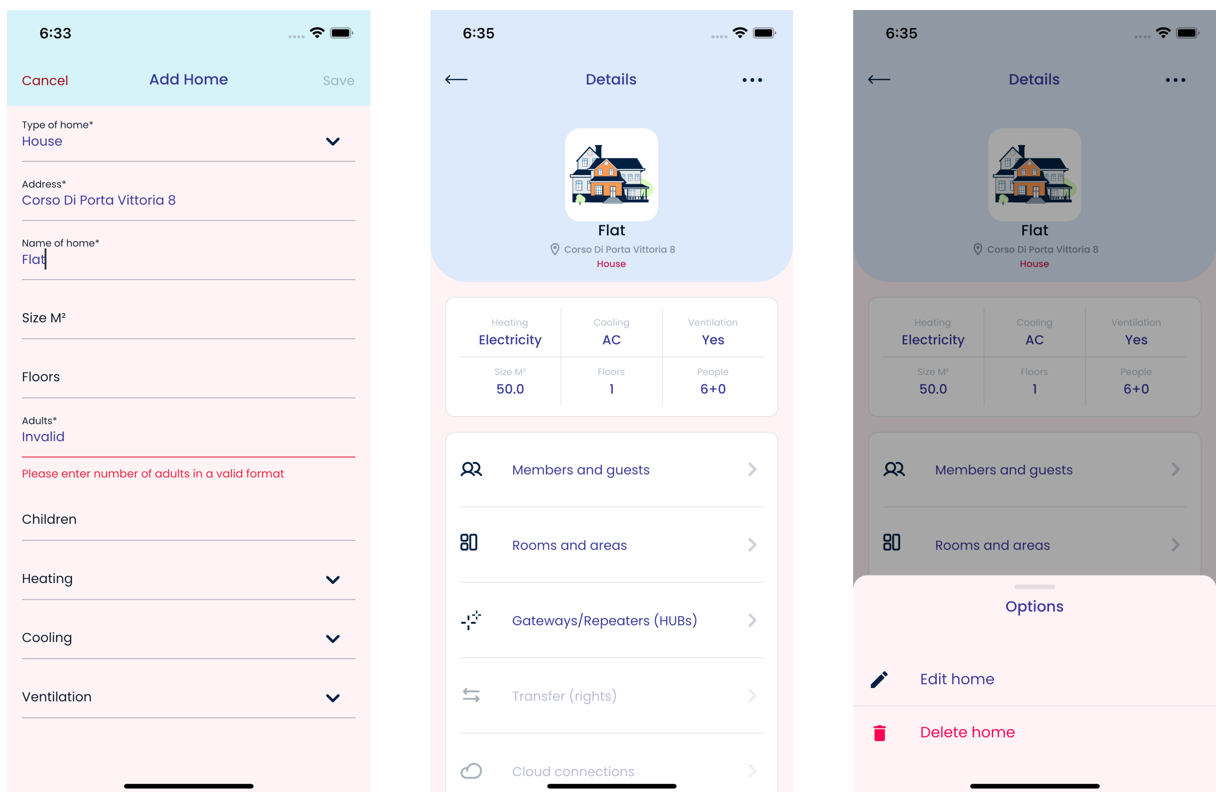

Figure B.4: Home Screens

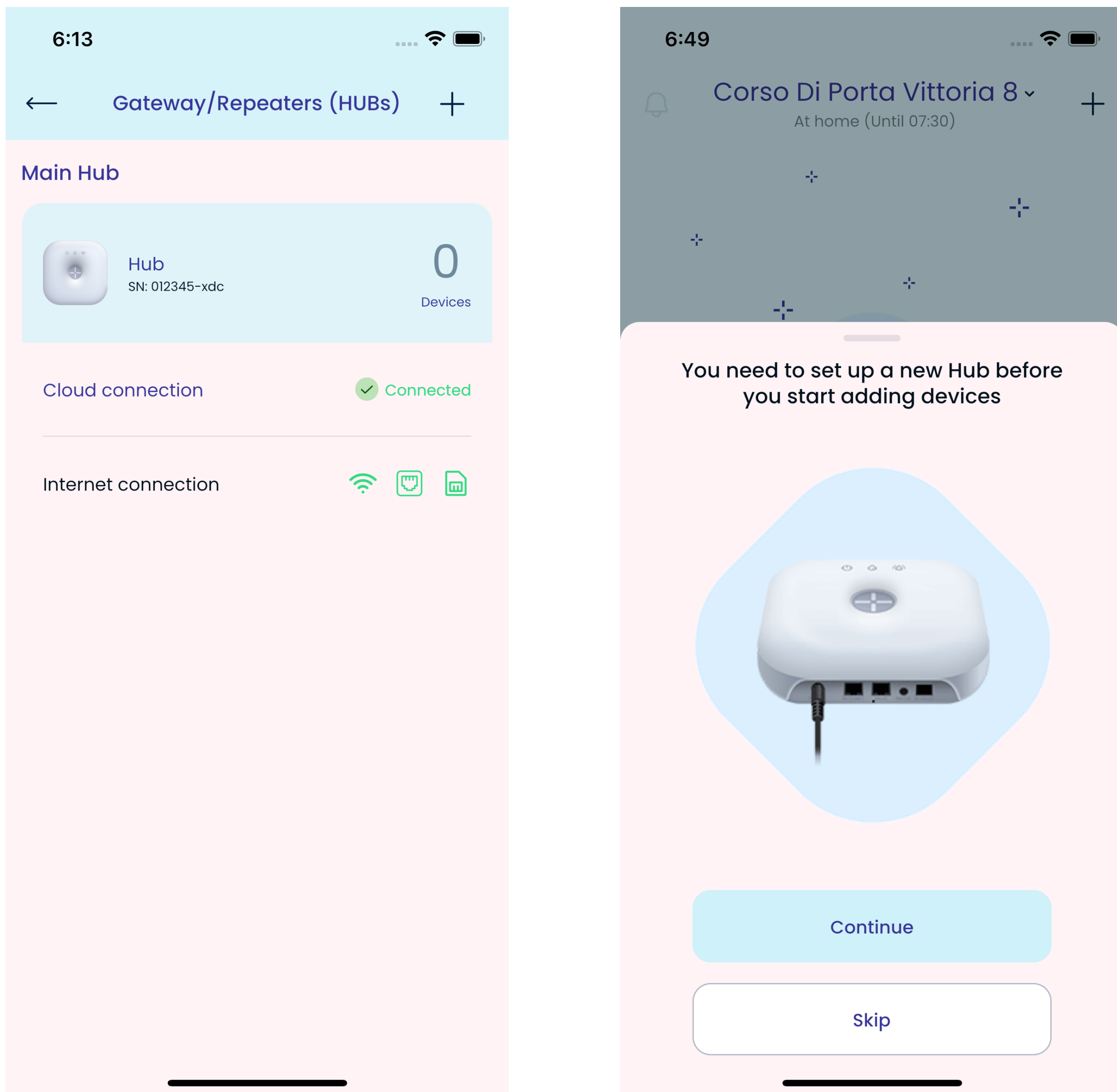

Figure B.5: Hub Screens

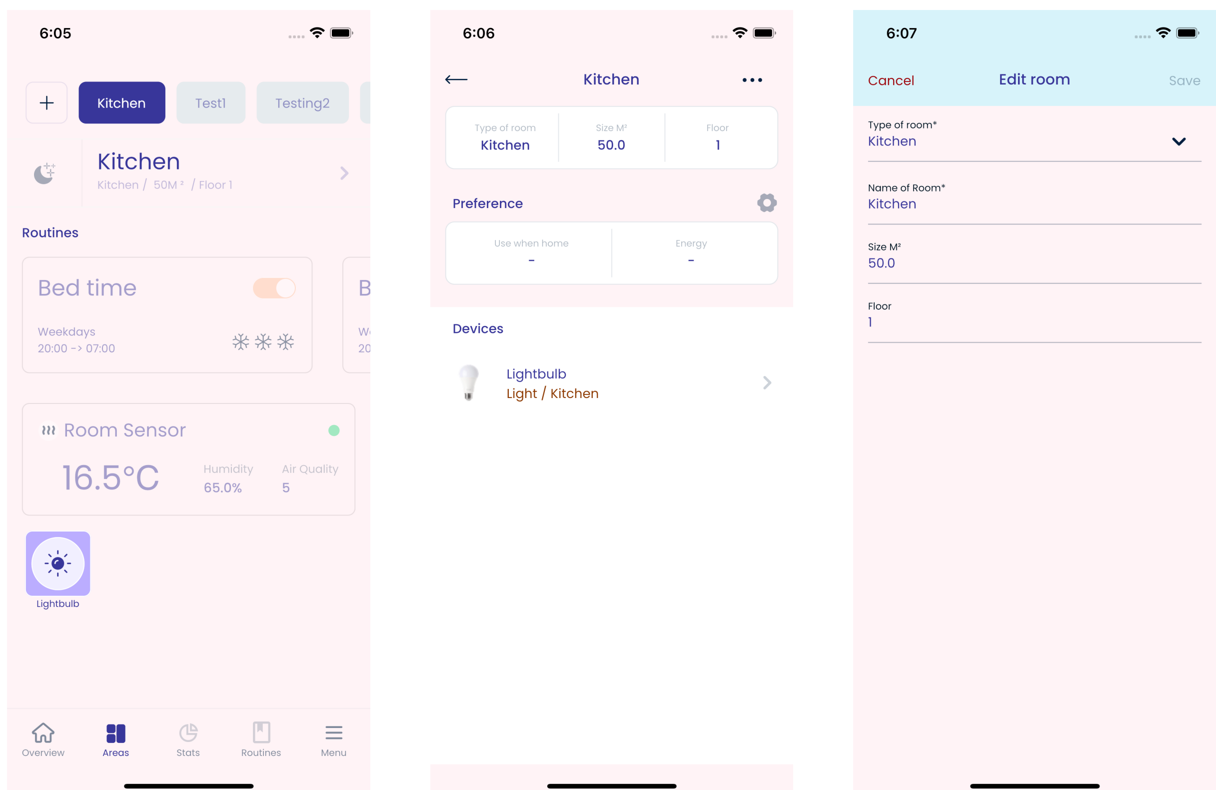

Figure B.6: Areas & Rooms Screens

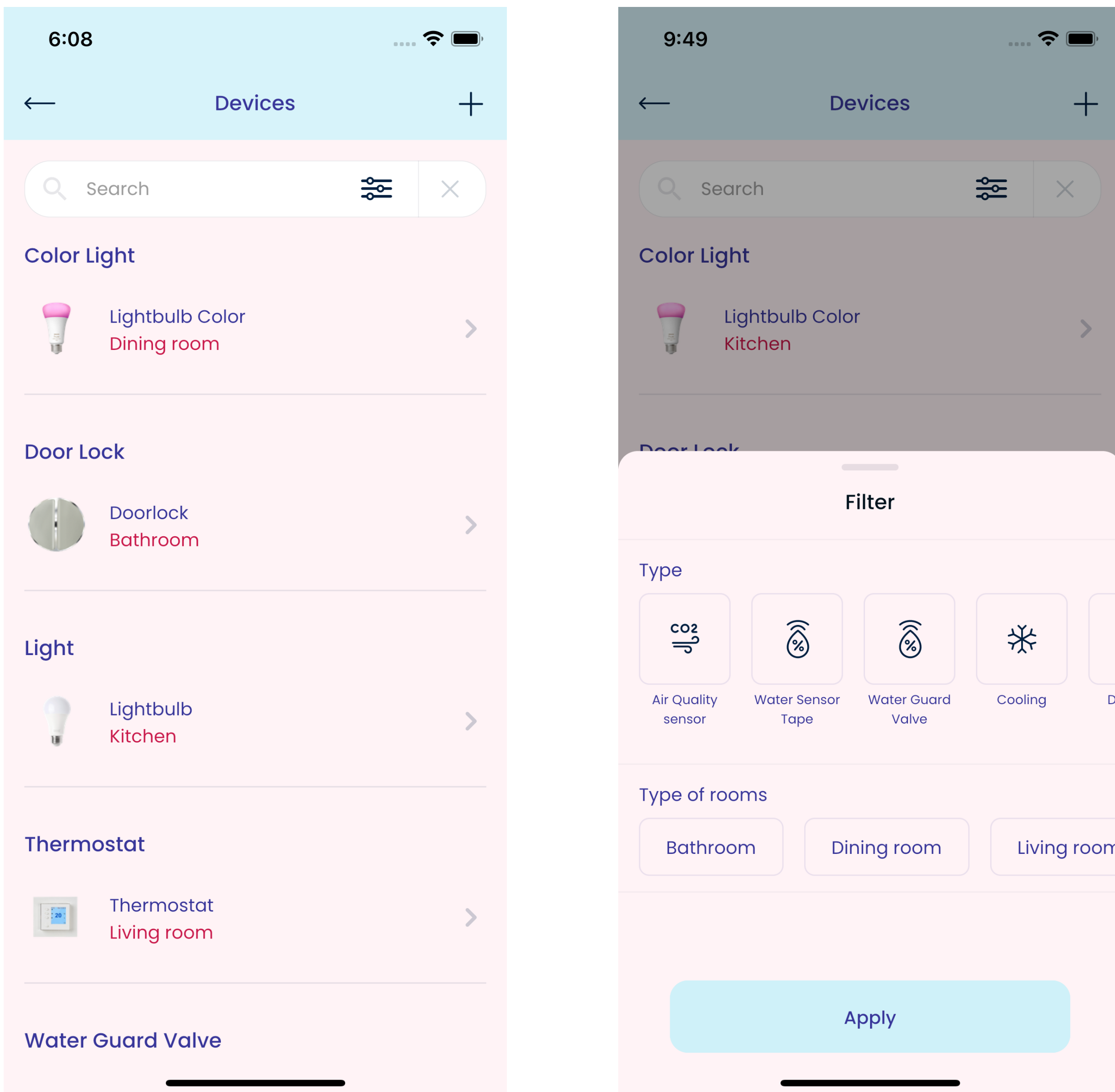

Figure B.7: Devices Screens

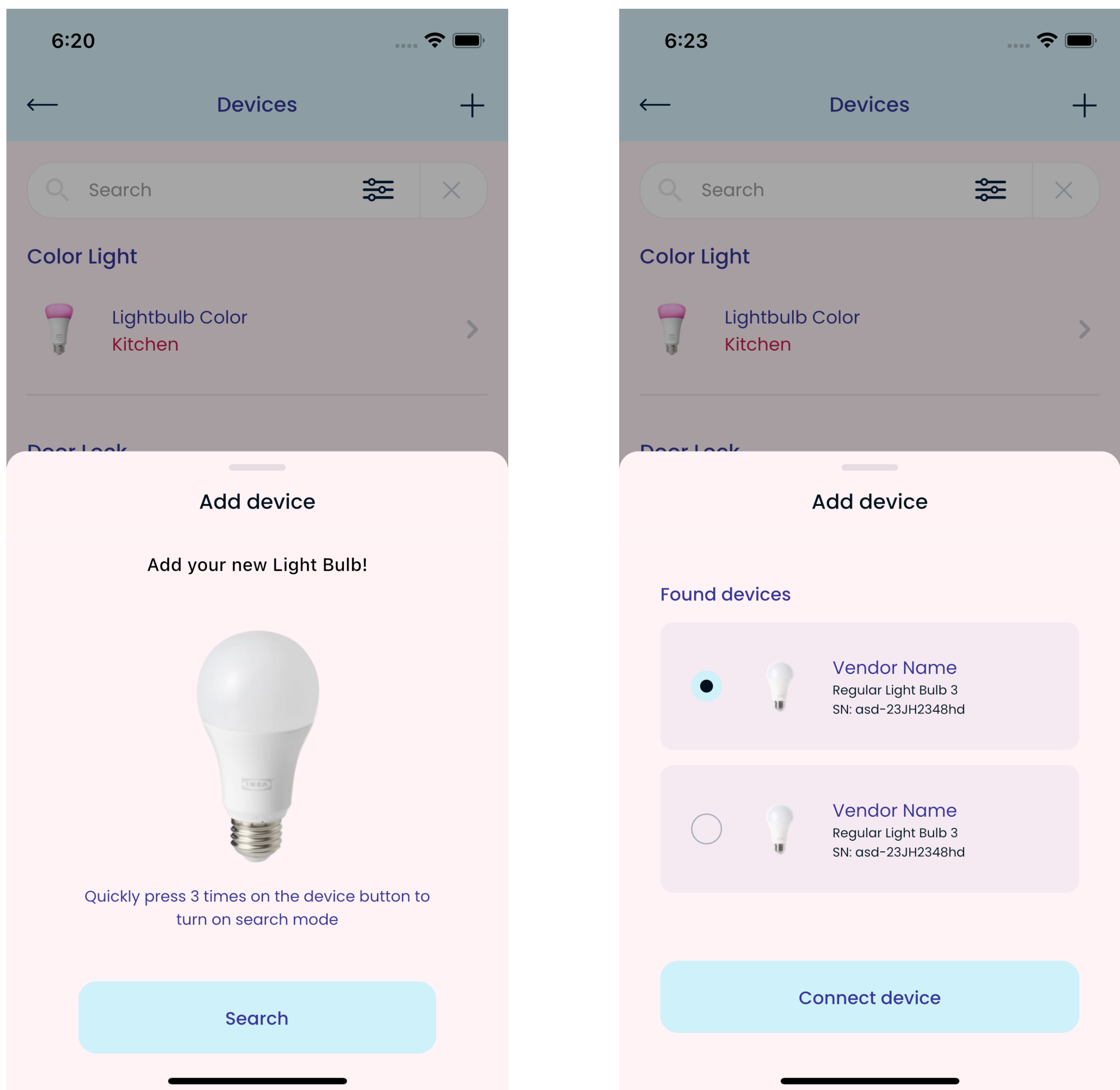

Figure B.8: Add Device Flow Screens

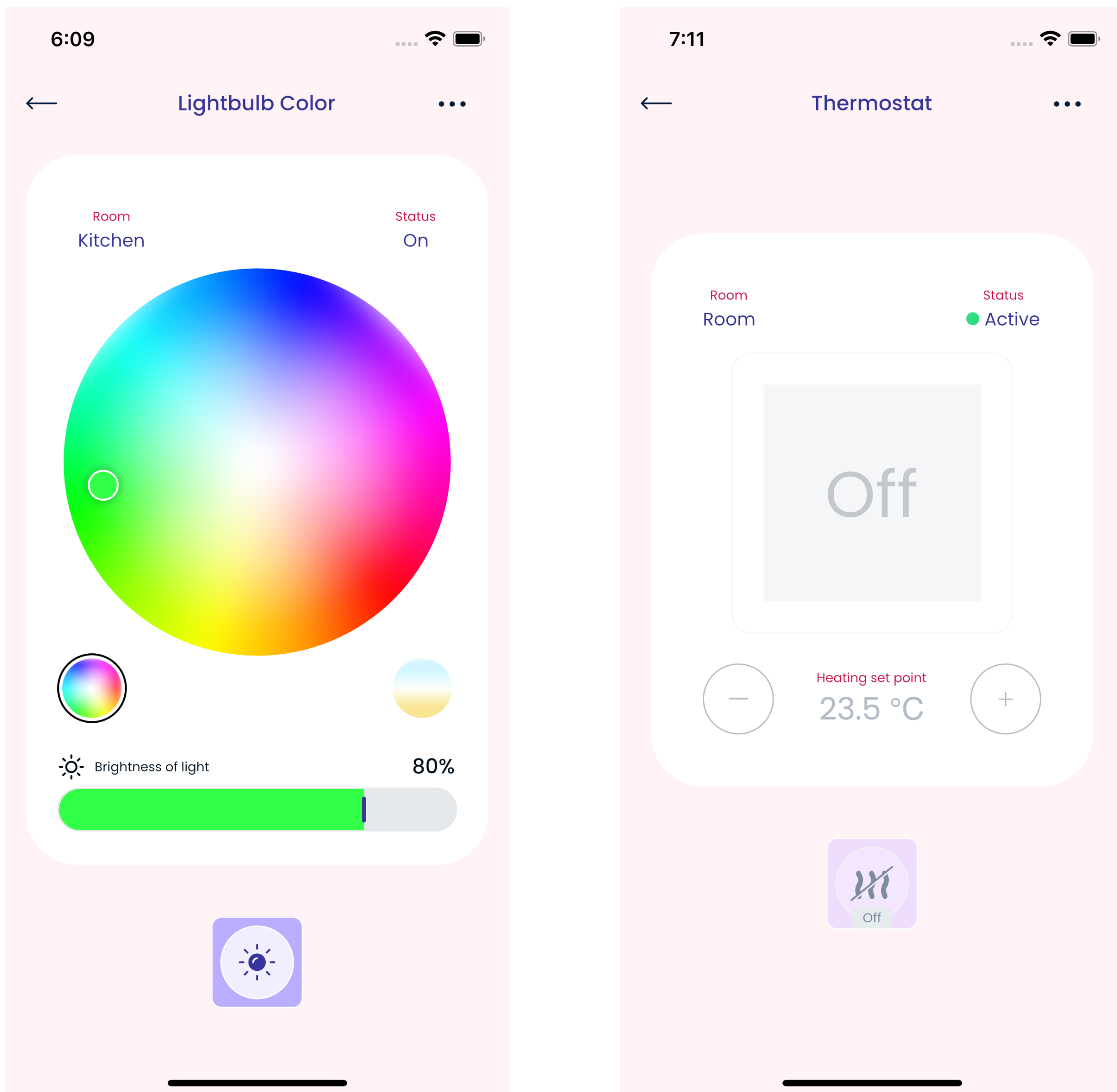

Figure B.9: Control Device Screens

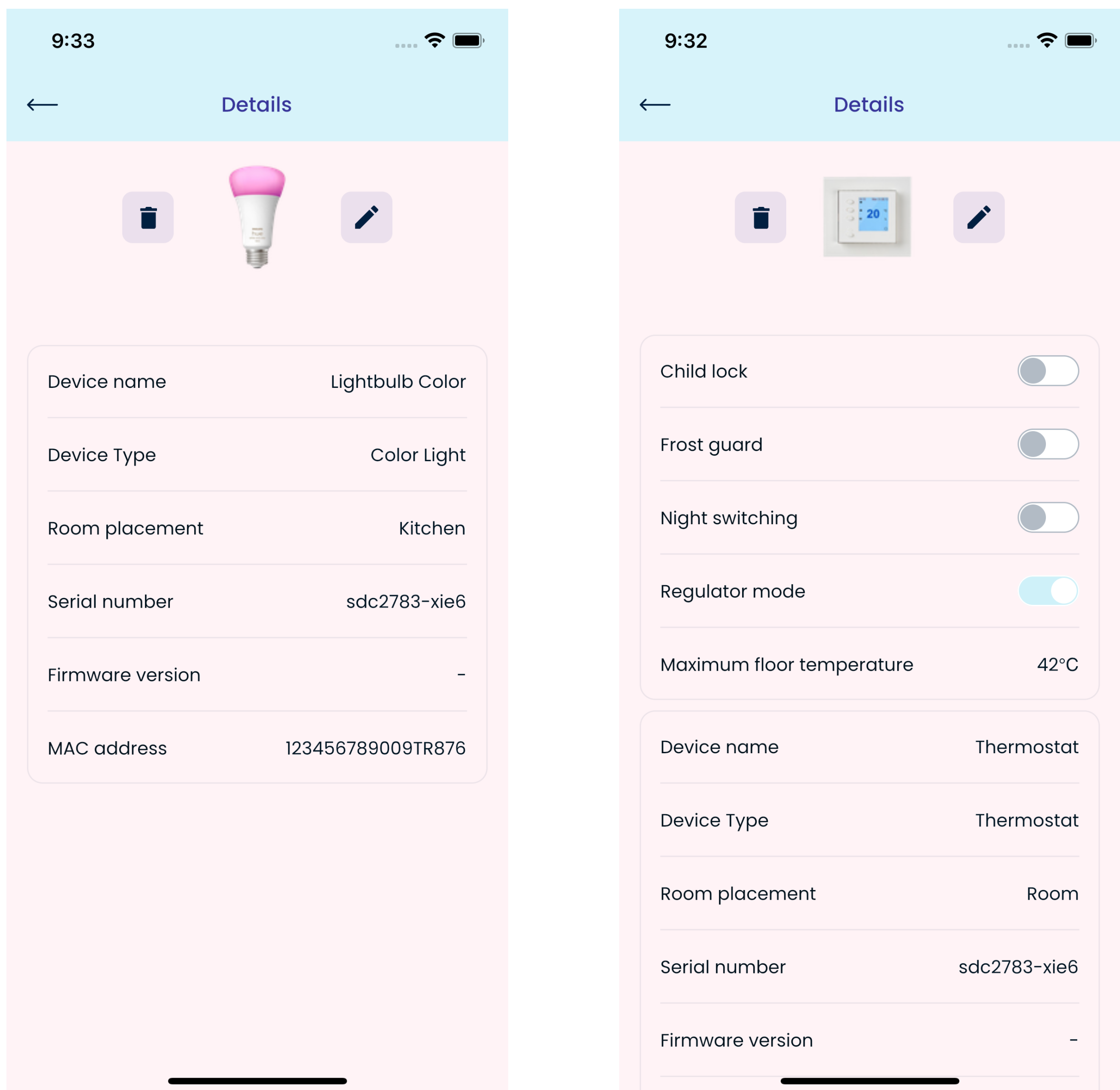

Figure B.10: Device Details Screens

List of Figures

1 Cross-Platform Categorization 2
2 Most used cross-platform mobile frameworks 3

1.1 Flutter architectural layers 9
1.2 InhertitedWidget state example 12
1.3 Flutter render pipeline 13
1.4 Native rendering approach 14
1.5 WebView rendering approach 14
1.6 Reactive Views rendering approach 15
1.7 Flutter rendering approach 15
1.8 Flutter declarative formula 16
1.9 Cubit architecture 18
1.10 Cubit flow 19
1.11 Bloc architecture 19
1.12 Bloc flow 20
1.13 Dart platforms 22
1.14 Layered architecture 27
1.15 Testing conceptual framework 30

2.1 Four-tier layered architecture 41
2.2 API Client Structure 43
2.3 Permission Client Structure 46
2.4 AsyncBehaviorSubject Structure 46
2.5 Repository Structure 47
2.6 Visual Application Feature Structure 49
2.7 Visual Application Feature Structure 50
2.8 Combined Application Feature Structure 52
2.9 Hybrid Architecture with Multimodule Monorepo 53
2.10 Simplified Project Directory Structure 54

3.1 State Management Solutions 86

B.1 Authentication Screens . . . 134
B.2 Account Detais Screen . . . 135
B.3 Overview Screens . . . 136
B.4 Home Screens . . . 137
B.5 Hub Screens . . . 138
B.6 Areas & Rooms Screens . . . 139
B.7 Devices Screens . . . 140
B.8 Add Device Flow Screens . . . 141
B.9 Control Device Screens . . . 142
B.10 Device Details Screens . . . 143

List of Tables

2.1 Flutter Test Classification 74

3.1 Quantitative Results from Dart Packages 83
3.2 Quantitative Results from lib directory 84
3.3 Summarized Total Quantitative Results 84

List of Source Codes

1 Flutter Hello World . . . 10
2 Api Client . . . 44
3 Area Repository . . . 47
4 Area Resource Failure . . . 48
5 Welcome Page . . . 49
6 Home Bloc . . . 50
7 Delete Device Event . . . 57
8 Delete Device State . . . 58
9 Delete Device Bloc . . . 59
10 Edit Device Page . . . 61
11 Plug Pairing Body . . . 63
12 Bottom Navigation Cubit . . . 64
13 Alarm Resource - Mocks and Fakes . . . 67
14 Alarm Resource can be instantiated . . . 68
15 Alarm Repository - Mocks and Fakes . . . 69
16 Water Leak Alarm Data Fetched group . . . 71
17 Mock Water Leak Alarm Bloc - setup . . . 75
18 Custom extension on WidgetTester . . . 76
19 Mock Water Leak Alarm Bloc - renders group . . . 76
20 Mock Water Leak Alarm Bloc - shows group . . . 78
21 Mock Water Leak Alarm Bloc - adds group . . . 79
22 Permission Client . . . 101
23 Async Behavior Subject . . . 103
24 Home Details Bloc . . . 108
25 Home Details Page . . . 110
26 Edit Device Event . . . 112
27 Edit Device State . . . 112
28 Edit Device Bloc . . . 114
29 Edit Device View . . . 115
30 Device Name Text Field . . . 116

31 Plug Pairing Body Switcher 117
32 Alarm Resource - alarms group 118
33 Alarm Repository 120
34 Alarm Repository - alarms group 121
35 Water Leak Alarm Bloc - Mocks and Fakes 122
36 Water Leak Alarm Bloc - initial state 123
37 Water Leak Alarm Ok Status Banner Closed group 123
38 Top Page-View Pattern 124
39 App Tester 127

C | Acknowledgements

First and foremost, I would like to thank my family. My parents and brother are the common factors behind all my successes and the main reason I avoided many failures. They have always supported me in every decision I made along the way while keeping me motivated to achieve my goals. They encouraged me to move to the U.S. to pursue my dream of studying and playing sports at a university level for four years. They allowed me to stay one extra year to work in Pittsburgh after graduating. And then again, they supported me when I decided to move back to Europe to get a Master's degree at Politecnico di Milano. These selfless actions, among other countless ones, speak volumes of the family I have and how far they are willing to go to give me a better life. It is an understatement that I would not be where I am or be who I am without my family, and graduating from Politecnico di Milano with a Master's degree in Computer Science and Engineering is yet another achievement that goes to my record, THANKS TO THEM. There are no words in any language to describe how grateful and lucky I am to have these people in my life.

I would also like to thank Very Good Ventures (VGV) and all the people that work at this amazing company. I have nothing but words of gratitude towards VGV for everything they have done for me ever since I started my internship with them. They never fell short of arranging valuable guidance and assistance when required or providing me with all the necessary resources to work on my thesis. Aside from all the technical knowledge I learned through my internship, they instilled in me so many priceless values and principles that I will forever carry with me and be grateful for. A special shout-out goes to Jorge and Óscar for all the support they showed me from the very first interview to this day.

I cannot write anything in English without thinking of my Canadian brother, Mircea Manuel Mangu. Who would have thought that all those lab reports you corrected for me, those sleepless nights where you put up with my unusual engineering schedules, and the countless occasions when you taught me a new life lesson would lead me here? Manny, you know better than anybody that every success I achieve in my life also belongs to

you, and this Master's degree is no exception.

How could I forget about il mio fratello italiano, Simone Staffa? I still think that missing class that day was probably the luckiest thing that ever happened to me during my days in Milano. I am positive that my experience at PoliMi would have not been the same without you. It was a pleasure and an honor to learn from you, be in the same class with you, and work together on so many successful projects. All these adventures will go down in history, and we now both have a degree to prove it... Grazie mille!

Last but not least, had it not been for my classmates and friends Gianluca and Benjamino, I would probably still be stuck in that Algebra Logic class. I owe you some of the happiest moments I can remember while studying or attending class and during the toughest moments we endured during the pandemic. Thank you for the memories and that delicious pasta!