Using Google´s Flutter Framework for the Development of a Large-Scale Reference Application

Bachelor Thesis to obtain the bachelor’s degree in the
Bachelor of Science Program *“Media Informatics”*
taught at the Faculty of Computer Science and Engineering Science of
Technical University Cologne.

Presented by	Sebastian Faust
Email	sebastian.faust1997@gmail.com

In Collaboration with	Capgemini Cologne
First Examiner	Prof. Dr. Christian Kohls
Second Examiner	Valentin Klein

Gummersbach, 31.01.2020

Declaration of Originality

I confirm that this assignment is my own work and that I have not sought or used inadmissible help of third parties to produce this work and that I have clearly referenced all sources used in the work. All work that I did not created myself is marked as such.

This work has not yet been submitted to another examination institution – neither in Germany nor outside Germany – neither in the same nor in a similar way and has not yet been published [1].

Gummersbach, 31.01.2020

Place, Date

[illegible]

Legal Signature

Abstract

With Google's Flutter framework continuing to grow in popularity for companies and developers alike, the need for an understanding of how to utilize the framework in a large-scale context has become more relevant than ever. The purpose of this thesis is to document the crucial steps most development teams using Flutter in a large-scale application will face. Additionally, a fully documented, large-scale reference application was generated so that other developers may use it as an aid when creating their own Flutter projects on a similar scale. Multiple steps were taken to ensure that optimal solutions were chosen for each aspect of the development process. For each of those aspects, a wide range of possible solutions were explored, compared and analysed. Finally, one of the possible solutions was chosen based on a wide range of scientific papers and community-generated sources. Additionally, an interview with an expert in the field was conducted to further validate those decisions. After the application was fully implemented, ten crucial aspects of the development process were identified. Those ten aspects are now explained in detail in this thesis. Ultimately, the knowledge provided by this thesis can act as a map for peers using Flutter in a large-scale context and help them overcome the crossroads they will most likely come to face.

Keywords: Flutter, state-management, layered architecture, BLoC pattern

Table of Contents

Glossary

Bridge	Enables the communication of two components, that would traditionally not be able to communicate [2].
Canvas	A component of a given mobile framework that is responsible for rendering pixels on the screen of the mobile device (Quartz 2D in IOS) [3], [4].
Component	*"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture"* [5]. Commonly also called module.
Interface	As defined by Branden Hookway, an *"interface"* is a shared boundary across which two or more separate components of a computer system exchange information [6]. In the context of object-oriented programming, *"interface"* also describes an abstract class that only defines a set of unimplemented functions, which ultimately serves the same purpose as the interface defined by Hookway.
Large-Scale	In the context of this thesis, *"large-scale"* software is referring to software that is characterized by being dependant on a clean structure and on a scalable architecture to remain maintainable with an ever-growing codebase.
Localization	An application is *"localized"* for a given country if all texts in the application are available in the native language of that country.
Mutate	In this thesis *"mutate"* refers to the altering of aspects of a given object. For example, *mutating* state is the changing of aspects of that state to create a new state.
Native	An application is considered *"native"* when it was developed in a platform-specific language, and can only run on mobile devices that operate under that platform [7], [8].
Package	In this thesis, the term *"package"* refers to a collection of software classes that are packaged together. They can be added to a program and provide some sort of additional functionality to that program.
Platform	*"A platform is a group of technologies that are used as a base upon which other applications, processes or technologies are developed."* [9]
Side Effect	In the context of software development, the term *"side effect"* refers to any modification a given function makes to the program that is outside of the function's local scope [10]. For example, if a function modifies an object it took in as a parameter, this modification is a side effect, as the object passed in is part of a larger scope.
Snippet	A short excerpt of source code.
State	Any data in an application that can change over time [11].
User Interface (UI)	*"Any component of an interactive system (software or hardware) that provides information and controls necessary for the user to perform a particular work task with the interactive system"* [12]
Widget	Visual component (or a component that interacts with the visual aspects) of an application [13].

Table of Figures

Table of Tables

Table of Code Snippets

1 Introduction

Cross-platform development is a fundamentally intriguing value proposition [7], [8], [14], [15]. In most cases, developers will want to reach the broadest possible audience with their application. To ensure this is possible, the application should be released on multiple platforms [16]. If the development team chooses to build their application *"natively"* [7], [8] for multiple platforms, they will need to maintain multiple codebases. Multiple codebases leads to more maintenance, more development effort and ultimately more financial cost [14]. Cross-platform frameworks promise to mitigate that cost by using one codebase to support multiple platforms [7], [14]. Because this is such an intriguing proposition, there have been a multitude of cross-platform approaches over the last decade [7]. Each with their own drawbacks and advantages. But generally speaking, they all offer the same trade-off: Less maintenance and less development cost for less performant and less stable applications [8], [17], [18].
In 2018 Google [19] released its own cross-platform framework to the public. The framework named *"Flutter"* [20] promises to keep the advantages of a cross-platform solution while maintaining native performance. This claim led to a lot of attention for the framework [21]. Big companies like BMW [22] are moving to Flutter for their consumer-facing application [23] and Flutter is now the fastest-growing skill among software engineers on *"LinkedIn"* [24]. The Flutter community is steadily growing, but as Flutter is still such a new framework, there is not yet a clear consensus on how to develop large-scale applications with it. Many possible approaches and recommendations are spread out over blog posts, articles and social media discussions. This thesis aims to help developers that plan on using the Flutter framework in a large-scale context.

1.1 Goals

This thesis makes two contributions to the Flutter community: firstly, it outlines a select set of crucial design decisions that were made during the development of a large-scale Flutter application. The focus here is on highlighting particularly difficult obstacles and novel solutions. Peers may use these descriptions as an aid when facing the same obstacles in their own large-scale Flutter applications.
Secondly, this thesis generates a fully documented, open-source, large-scale application that peers may use as a guide when creating their own projects with Flutter.

1.2 Creation Context

This thesis was written by a student in the Bachelor of Science Program *"Computer Science and Media Technology"* at Technical University Cologne [25]. The work was executed in collaboration with Capgemini Cologne [26]. Capgemini's DevonFw open-source initiative [27] maintains the *"My Thai Star"* [28] application as a reference project for using a broad range of technologies in a large-scale context. Parts of that application were recreated using Flutter for this thesis. A more detailed outline of *My Thai Star* and the exact scope of this project is given in the next chapter.

1.3 Approach

This section summarizes the steps that were taken to achieve the goals described in *Section 1.1*. In general, the creation process can be categorized into the following four phases.

1.3.1 Preparation

In preparation for this thesis a guide [29] on developing large-scale applications using Flutter, was written and published through Capgemini's DevonFw open-source initiative. The guide is based on a wide range of community-generated sources, scientific papers on cross-platform development and the official Flutter documentation. The guide was well received by the Flutter community. As of the writing of this thesis, it is the most *"starred"* GitHub [30] repository ever published by DevonFw. As part of the preparation, a small-scale application was developed to test some of the recommendations made by the guide.

1.3.2 Domain Analysis

Secondly, the domain of the *My Thai Star* reference application was analysed, and the exact scope of the project was defined.

1.3.3 Interview

After the domain was analysed, an interview with an expert in the field was conducted to further validate and improve the findings made in the preparation phase. Felix Angelov is one of the developers responsible for rebuilding BMWs consumer-facing application with Flutter. He is also the main contributor and author of the BLoC package [31]. Thusly he has experience in using Flutter with a BLoC pattern-based architecture in a large-scale context. The BLoC pattern [32] and its related architecture are described in *Section 4.1* and *4.2* respectively. On the ninth of December 2019, a one hour long interview was held with Angelov under the topic of *"The BLoC Pattern and Flutter in large-scale applications"* [23]. The knowledge generated by that interview is cited in multiple sections of the thesis.

1.3.4 Implementation

All the knowledge generated during the previous phases was then condensed into the actual implementation of the reference application. Selected portions of the development process are outlined in *Chapter 4*.

1.4 Structure of this Paper

The structure of this paper is as follows: the second chapter outlines the domain of the *My Thai Star* reference application. This is also where the exact scope of the project is defined. The third chapter provides an introduction to the Flutter framework and its central concepts. On the basis of these two chapters, the third chapter presents selected aspects of the development process of the *My Thai Star* Flutter implementation. Ten aspects where chosen, each of which is detailed in its own section of the chapter. Lastly, the fifth chapter will summarize the insights produced by the thesis and reflect on the work that was done.

2 The Reference Project

The *My Thai Star* application is an open-source reference project maintained by Capgemini’s DevonFw initiative to showcase a set of technologies in a large-scale context. The application is a reservation system for a fictional Thai restaurant. This chapter elaborates how that reservation system is implemented and what the state of the system was when this thesis started. Secondly, this chapter showcases the *My Thai Star* domain, what use-cases it covers and what terminology was defined to describe domain-specific entities. Thirdly, this chapter explains what the responsibilities of the different *My Thai Star* components are. And lastly, this chapter defines the exact scope of the project; which parts of *My Thai Star* were recreated in Flutter and which parts were left out.

2.1 Implementations

The *My Thai Star* GitHub repository [28] contains multiple implementations of a reservation system using different technologies. *Figure 1* shows the three components that form the application and with which technologies they are implemented. Each component has several different versions. The back-end, for example, is not implemented using Java [33], NodeJS [34] and .Net [35] in conjunction but instead, there are three functionally identical versions of the back-end each implemented with a different technology. The several implementations of the components are meant to showcase current best-practices and conventions of that specific technology in a large-scale context.

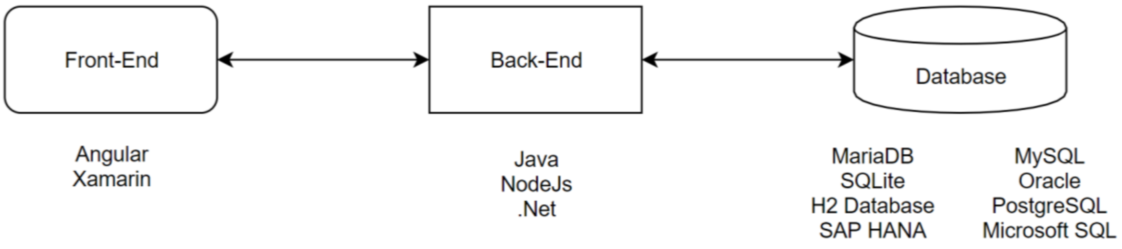
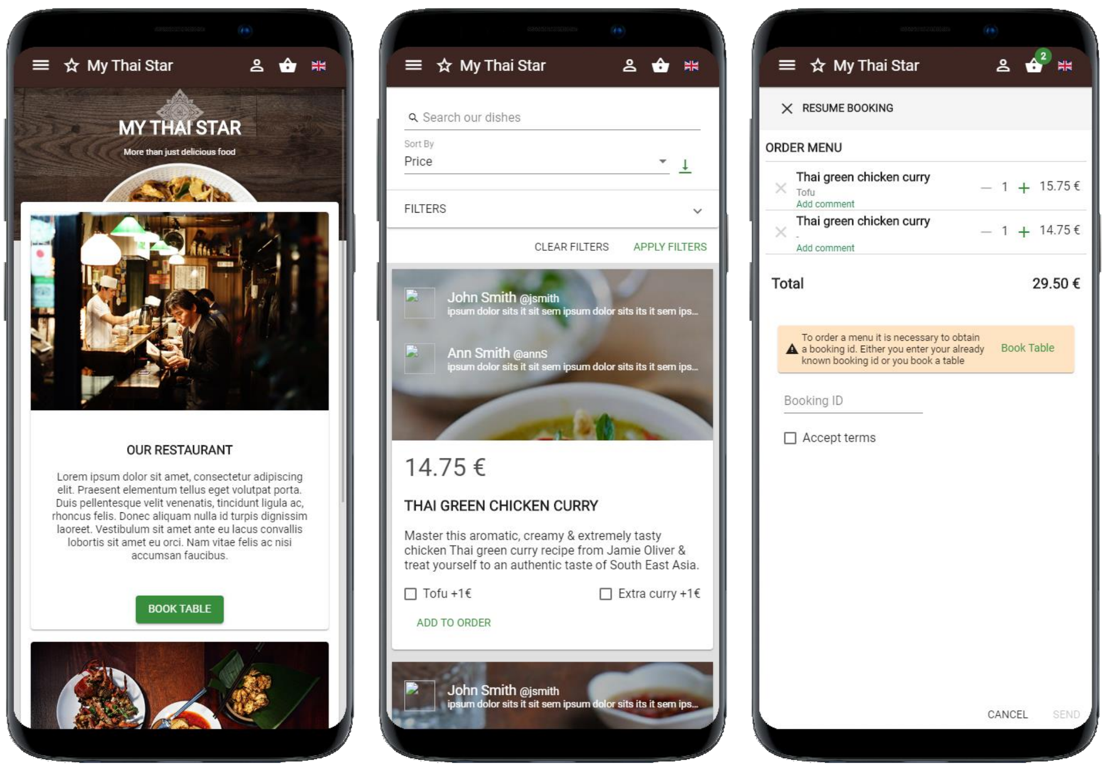


Figure 1 *My Thai Star* Components

All back-end implementations share a common interface. That way each front-end implementation can communicate with each back-end implementation interchangeably. The communication between front- and back-end can occur through either JSON [36] or XML [37] files.

From here on out, when this thesis is referring to the original *My Thai Star* implementation, it is referring to the implementation consisting of an Angular [38] front-end, Java [33] back-end and an H2 Database [39]. As of the writing of this thesis, all versions of *My Thai Star* are at vastly different stages of development and support different sets of features. The Angular, Java, H2 version is to be considered the main implementation, as it is the default version defined by the Docker-Compose file [40] in the root of the *My Thai Star* GitHub repository. The following figure shows screenshots of that Angular front-end to convey a basic idea of what the application looks like. A detailed description of what each of the components is responsible for is provided in *Section 2.3*.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent elementum tellus eget volutpat porta. Duis pellentesque velit venenatis, tincidunt ligula ac, rhoncus felis. Donec aliquam nulla id turpis dignissim laoreet. Vestibulum sit amet ante eu lacus convallis lobortis sit amet eu orci. Nam vitae felis ac nisi accumsan faucibus.

ipsum dolor sits it sit sem ipsum dolor sits its it sem ips...

Master this aromatic, creamy & extremely tasty chicken Thai green curry recipe from Jamie Oliver & treat yourself to an authentic taste of South East Asia.

To order a menu it is necessary to obtain a booking id. Either you enter your already known booking id or you book a table

Figure 2 Example Screens of the *My Thai Star* Angular front-end [28]

2.2 The Domain

As mentioned above, the *My Thai Star* application is an implementation of a reservation system for a fictional Thai restaurant. The following table lists all fully implemented features of the original *My Thai Star* implementation as of the writing of this thesis:

Feature	Description
Digital Menu	Displays a list of dishes that the restaurant offers. The list is searchable and sortable.
Book a Table	Gives the option to place a booking for a table at a specified date.
Invite a Friend	Gives the option to invite someone to a placed booking.
Order Food	Gives the option to order a list of dishes for a booking that was placed in advance.
Waiter Cockpit	Gives the staff the option to see all placed bookings and made orders.
Localization	All texts in the app are available in multiple languages.

Table 1 Features of the original *My Thai Star* implementation

The *My Thai Star* documentation defines a set of domain-specific terms. The following table summarizes the terms relevant for this thesis:

Term	Description
Booking	One reservation made by a guest for a specified date and time.
Dish	One meal offered by the restaurant.
Extra	Modifications made to a *“dish”*. Any *“dish”* may have a set of possible extras it can be served with, such as extra curry or tofu instead of meat.
Order	A list of *“dishes”* that will be served at a specified *“booking”*.
Booking Token	An alphanumeric code that uniquely identifies one *“booking”*. It can be used to place an *“order”* for a given *“booking”*.

Table 2 Definition of domain-specific terms

An interaction of a user with the Original *My Thai Star* application could look like this: the user fills in a digital form and books a table for a specified date and time. Then the user receives a confirmation for that booking with a booking token. Next, the user selects a list of dishes from the digital menu that they would like to be served when they attend their booking. Once all dishes are selected, the user is asked to enter the booking token they received previously to confirm their order.

2.3 Components

As explained in *Section 2.1*, *My Thai Star* is made up of 3 types of components. Each of these components has a different set of responsibilities which are summarized in the following table:

Component	Description
Front-end	Responsible for validating user input, the localization of texts, handling the state of the current order and communicating with the back-end through HTTP [41] calls.
Back-end	Provides the following three services to the front-end: **Dish Management** Provides a searchable list of all dishes. Dishes are loaded from the database. **Booking Management** Will validate and save a received booking and generate a booking token. Authenticated staff can search through saved bookings. Bookings are saved in the database. **Order Management** Will add an order to a booking through a provided booking token. Orders are saved in the database. The back-end will eventually also handle user authentication. That feature is however not yet fully functional as of the writing of this thesis.
Database	Stores data persistently.

Table 3 *My Thai Star* components

2.4 Scope of this Project

Because this thesis was conducted by one person in nine weeks, it would have been impossible to recreate the entirety of *My Thai Star* using Flutter. And because this thesis is meant to produce a reference project, the decision was made to focus on a small subset of *My Thai Star's* features and aim to develop that subset with a high level of care. As stated in the Glossary, for this thesis *"large-scale"* is not referring to a large number of developers, but instead referring to the need of *large-scale* development projects to follow a clean structure and a scalable architecture to remain maintainable with an ever-growing code base. In other words, the thesis produces a medium-sized application, with a special focus on structure, scalability, and maintainability.

More specifically, a new front-end component for the *My Thai Star* project is created for this thesis. This new component communicates with one of the existing back-ends as outlined in *Section 2.3*. Furthermore, the *"Invite a Friend"* and *"Waiter Cockpit"* feature outlined in *Section 2.2* are not implemented due to time constraints. All features that were not fully implemented are marked with the annotation [42] *"@notFullyImplemented"* in the source code of the project [43] to make them immediately identifiable. The following figure illustrates which parts of the *My Thai Star* application are rebuilt during this thesis and how they fit into the existing project:

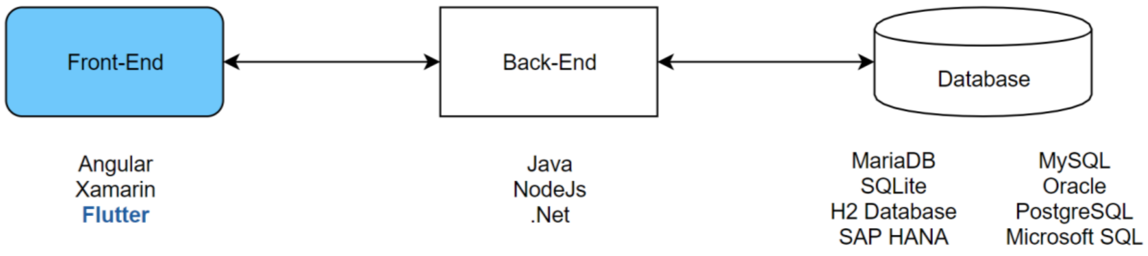


Figure 3 Addition to the *My Thai Star* Components

3 The Flutter Framework

This chapter aims to give a brief introduction to the Flutter framework. This topic is already covered in detail in the aforementioned guide [29] that was written in preparation for this thesis. Thusly this chapter will share a lot of similarities with the chapter *“The Flutter Framework”* of that guide. This chapter was however still included here to make the thesis more self-contained and to not require the reading of another document to fully understand its contents.

Firstly, this chapter explains how the rendering of Flutter applications functions and how Flutter’s approach differs from other cross-platform frameworks. Secondly, a comparison between Flutter’s *“declarative”* [44] programming approach and a more traditional *“imperative”* [44] approach of other frameworks is made. Next, the general structure of Flutter applications is showcased and lastly, the different types of widgets provided by the framework are described and compared.

3.1 Technical Level

Flutter claims to produce applications with a *“native”* [7], [8] performance while keeping the many benefits [7], [17] of a cross-platform framework. This section showcases how Flutter aims to fulfil that claim. To illustrate this, a comparison between Flutter’s cross-platform approach, other popular cross-platform approaches and native app rendering is made. It is important to note here that these are high-level abstractions that summarize a lot of different approaches. The aim is to only give a general understanding of the concepts.

3.1.1 Native

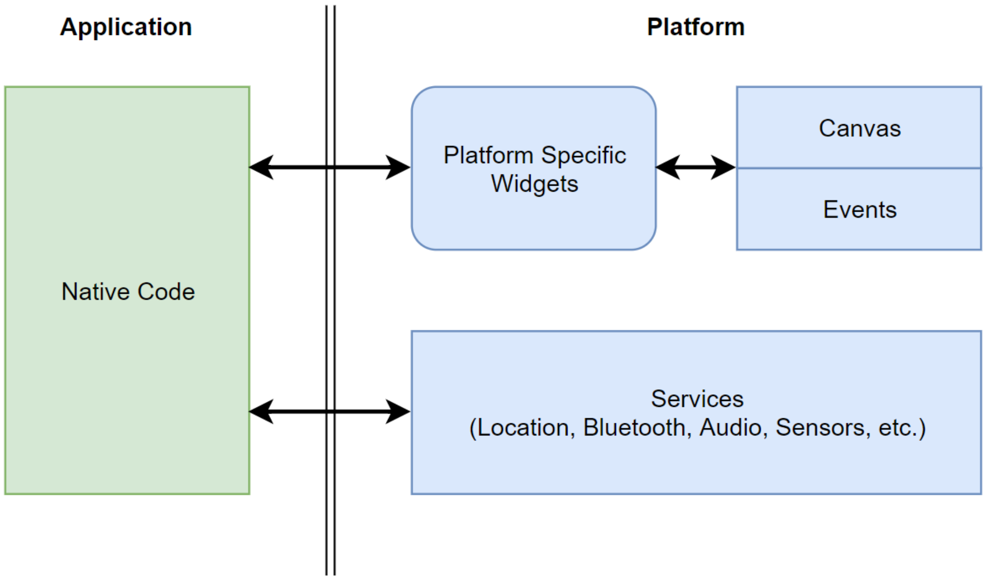

Application
Platform
Native Code
Platform Specific Widgets
Canvas
Events
Services
(Location, Bluetooth, Audio, Sensors, etc.)

Figure 4 Native app rendering [45]

The traditional way to build a mobile application is to write native code for each platform that should be supported. As of the writing of this thesis, that would most likely [14], [16], [46] be one for IOS [47] and one for Android [48].

In this approach, the application is written in a platform-specific language that defines platform-specific widgets. These platform-specific widgets hold information on how they look and where they are positioned on the screen. This information is then passed from the widgets to the canvas, which is responsible for rendering the widgets on the screen of the mobile device. Events such as screen taps are read in by the platform and passed on to the widgets. The native code can then define how these widgets should react to new events. Native applications have direct access to platform-specific services and sensors [45], [49], [50]. Important to note here is that the rendering of the application is a responsibility of the platform.

3.1.2 Embedded Web Apps

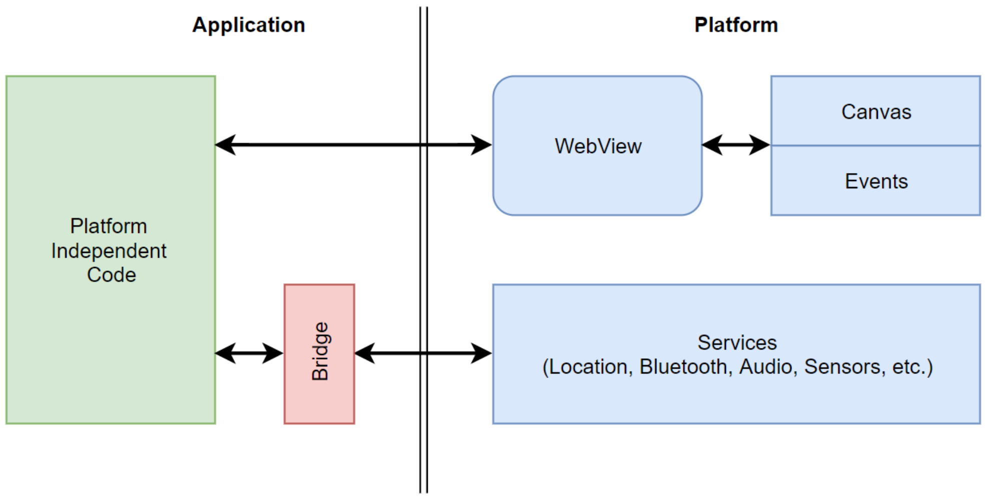

Application
Platform
Platform Independent Code
Bridge
WebView
Canvas
Events
Services
(Location, Bluetooth, Audio, Sensors, etc.)

Figure 5 Embedded web app rendering [45]

Embedded web apps were one of the earliest approaches to cross-platform development. For this approach, a given team builds the application with HTML [51], CSS [52], and JavaScript [36] once. That application is then rendered through a native *Web View* which is in its essence a mobile browser [45], [49]. The rendering of embedded web apps takes place in the platform; the *Web View* is responsible for rendering the HTML, CSS and JavaScript code on the canvas. This approach comes with several problems. Firstly, the development team is limited to a web technology stack. Secondly, the communication between the app and native services always has to run through a bridge. And lastly, the performance of the application can depend on the internet connection of the device running the application.

3.1.3 Interpreted

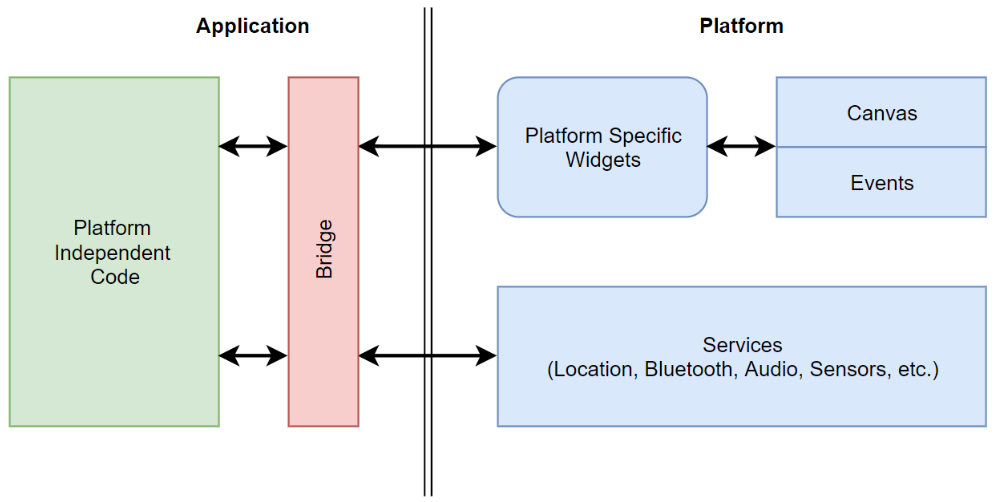

Application
Platform
Platform Independent Code
Bridge
Platform Specific Widgets
Canvas
Events
Services
(Location, Bluetooth, Audio, Sensors, etc.)

Figure 6 *Interpreted* app rendering [45]

Applications built with *"Interpreted approaches"* [7] like React Native [53] are written in a platform-independent language. The platform-independent code defines a set of generalized widgets that make up the application. Each generalized widget has one associated platform-specific widget for each platform the framework supports [54]. Depending on which platform the application is currently running on, a bridge between the application and the platform decides which platform-specific widgets these generalized widgets are displayed as [54]. For example, the platform-independent code might define that a generic button should be displayed at a given point. The bridge then interprets this command as *"display an Android button"* or *"display an IOS button"* depending on the current platform. All communication with platform services also runs through that bridge. The rendering of the application, as with all previously mentioned approaches, happens on the platform. The main problem with this approach is that the communication through the bridge is a potential bottleneck that can lead to performance issues [55].

3.1.4 Flutter

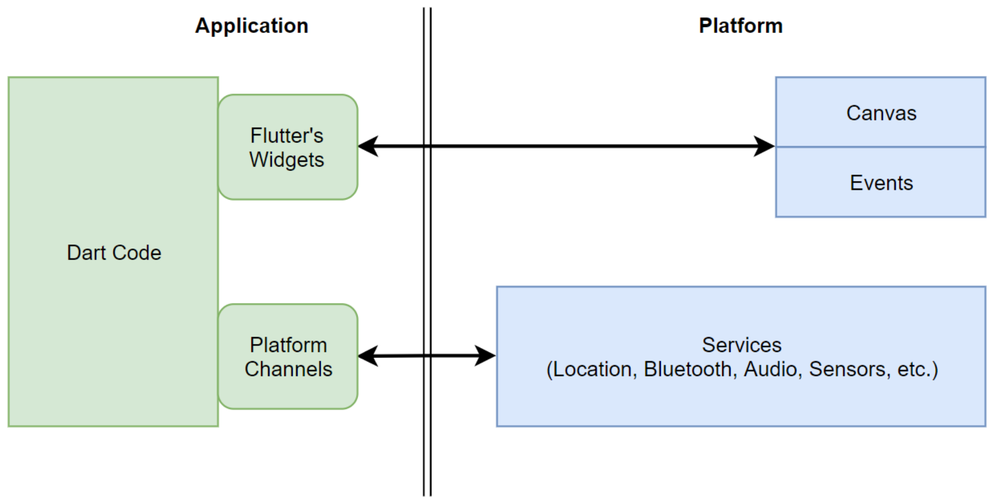


Figure 7 Flutter app rendering [45]

Flutter's approach is to move the entire rendering process into the application. The rendering runs through Flutter's own engine and uses Flutter's own widgets [45], [49], [50]. The Flutter framework communicates directly with the canvas of the platform. This canvas then displays the finished frames that were rendered in the application. This limits the bridging between the app and native environment to rendered frames and events which minimizes the bottleneck that could potentially be caused by bridging [45], [49], [50]. Communication between the Flutter application and the platform services runs through *"Platform Channels"* [56] which are in their essence also bridges, thus this potential bottleneck still persists.
One might think that keeping an entire rendering engine inside an application would lead to rather large application files, but as of the writing of this thesis, the compressed framework is only 4.3 megabytes in size [57].

3.2 Declarative Framework

Flutter's own introduction website states that Flutter is a *"declarative"* [44] framework [11]. This means that in Flutter, the user interface is never *"imperatively"* [44] or explicitly called in code. The code rather declares that the user interface should look a certain way, given a certain state. Another way to think about this concept is to imagine that the entire user interface of a Flutter application is the result of one function that takes in the state of the current app as a parameter [9].
A good way to illustrate the difference between *imperative* and *declarative* programming is through an example. The following two code snippets implement the same behaviour in two different frameworks. In these examples, a button is implemented that changes its colour to red once it is pressed. In an *imperative* framework like Android, one would *imperatively* call the button through its ID and directly define its behaviour through an *On-Click Listener*.

```
Button button = findViewById(R.id.button_id);
button.setBackground(blue);
boolean pressed = false;

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view){
        button.setBackground(pressed ? red : blue);
    }
});
```

Code Snippet 1 Red button in Android (Imperative) [29]

In Flutter, on the other hand, it is *declared* that the colour should be displayed in a certain way given a certain state. It is crucial to notice that the button in the Flutter example is the return value of a *"build"* method. Each widget in Flutter has such a *build* method to define how it is displayed on the screen. How multiple widgets are orchestrated in an application is explored in the next section.

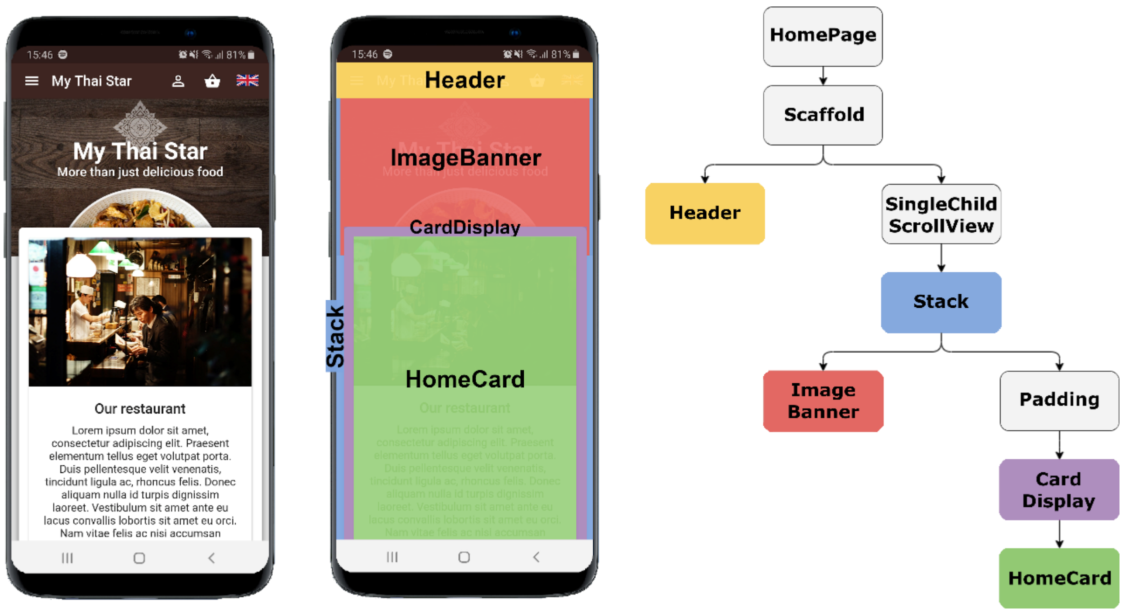
```
bool pressed = false; //State

@override
Widget build(BuildContext context) {
    return FlatButton(
        color: pressed ? Colors.red : Colors.blue,
        onPressed: () {
            setState(){ //Trigger rebuild of the button
                pressed = !pressed;
            }
        }
    );
}
```

Code Snippet 2 Red button in Flutter (Declarative) [29]

Whenever the state of a Flutter application changes, the parts of the application that depend on that state need to be rebuilt to display these changes. This can either be forced through the *"set state"* function or the widget can be notified that it needs to update using a publish-subscribe setup; the latter of these approaches is further explored in *Section 4.1*.

3.3 Widget Tree

A Flutter application is in its essence a tree of nested widgets [58]. This is illustrated by the following figure. it shows the *Home Page* of the Flutter *My Thai Star* app with some of its widgets highlighted. The widget tree associated with that screen is on the right side of the figure:



Our restaurant

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent elementum tellus eget volutpat porta. Duis pellentesque velit venenatis, tincidunt ligula ac, rhoncus felis. Donec aliquam nulla id turpis dignissim laoreet. Vestibulum sit amet ante eu lacus convallis lobortis sit amet eu orci. Nam vitae felis ac nisi accumsan

Figure 8 *My Thai Star* Flutter with highlighted widgets and widget tree [43]

This page and widget tree can be defined with the following Dart [59] code. Any Flutter application follows this structure. Every widget has a *build* method in which it can call any number of other widgets. Even the root of the application itself is such a widget.

```dart
class HomePage extends StatelessWidget {
  static const double _cardDisplayTopPadding = 170;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: Header(),
      drawer: AppDrawer(),
      body: SingleChildScrollView( //Makes Stack scrollable
        child: Stack(
          children: <Widget>[
            ImageBanner(),
            Padding(
                child: CardDisplay(), //Holds 2 HomeCards
                padding: EdgeInsets.only(top: _cardDisplayTopPadding)),
          ],
        ),
      ),
    );
  }
}
```

Code Snippet 3 *My Thai Star* Flutter *Home Page* implementation [43]

3.4 Types of Widgets

As explained in the previous section, widgets are an essential part of any Flutter application. The Flutter framework provides three distinct types of widgets. These types and their respective use cases are briefly outlined in this section.

3.4.1 Stateless Widgets

Stateless Widgets are the most basic type of widget [60]. These widgets must be completely immutable [61]–[63], which means that none of their values can change after initialization [64]. A full description of what immutability is and what benefits it provides is given in *Section 4.5*. A *Stateless Widget* primarily consists of a *build* method as shown in the following code snippet. This *build* method can be called multiple times a second by the Flutter framework and thusly should be kept as lightweight as possible [65], [66]. As per the recommendations of the Flutter team, *Stateless Widget*s should make up the vast majority of a Flutter application to improve its performance [65], [66].

```
class MyWidget extends StatelessWidget {

  ///Called multiple times a second.
  ///
  ///Must be kept lightweight.
  ///This is where the UI is built.
  @override
  Widget build(BuildContext context) {...}
}
```

Code Snippet 4 *Stateless Widget* lifecycle [13]

3.4.2 Stateful Widgets

Stateful Widgets are responsible for holding the mutable state of a Flutter application [67]. They consist of two parts, an immutable widget object and a mutable state object [68]. The widget is solely responsible for initializing the state object and holding on to it during the runtime of the application. The state object itself is more long-lived than other parts of the Flutter application; it will remain in memory even if the related widget is rebuilt [68]. The state object also has a more complex life cycle than a widget; it provides an initialization function, a *build* method and a *dispose* function as shown in the following code snippet. Crucially, a state object can hold mutable data in its member variables that can change after the state object is initialized.

As per the recommendations of the Flutter team, *Stateful Widget*s should be used as little as possible for performance reasons [65], [66]. There are primarily three reasons for why one would use a *Stateful Widget* over a *Stateless Widget*:

(1) The widget needs to hold any kind of data that has to change during its lifetime.
(2) The widget needs to do some sort of initialization at the beginning of its lifetime.
(3) The widget needs to dispose of anything or clean up after itself at the end of its lifetime.

```
class MyWidget extends StatefulWidget {

  ///Called immediately when first building the StatefulWidget
  @override
  State<StatefulWidget> createState() => MySate();
}

class MyState extends State<MyWidget>{

  ///Called once one creation
  @override
  initState(){...}

  ///Called multiple times a second.
  ///
  ///Must be kept lightweight.
  ///This is where the UI is built.
  @override
  Widget build(BuildContext context){...}

  ///Called once before the State is disposed (app shut down)
  @override
  dispose(){...}
}
```

Code Snippet 5 *Stateful Widget* lifecycle [13]

3.4.3 Inherited Widget

These widgets allow the propagation of data within a widget tree [13], [69]. A given *Inherited Widget* can expose any kind of data to all its descendants in the widget tree. All descendants can then access that data through their *Build Context* [70] as shown in the following figure. A *Build Context* is an object passed into every *build* method of a widget that contains a reference to all ancestors of that widget in the widget tree [13].
*Inherited Widget*s are the most common way of distributing state throughout the widget tree of an application and the underlying technology for many of the state management solutions described in *Subsection 4.1.2*.

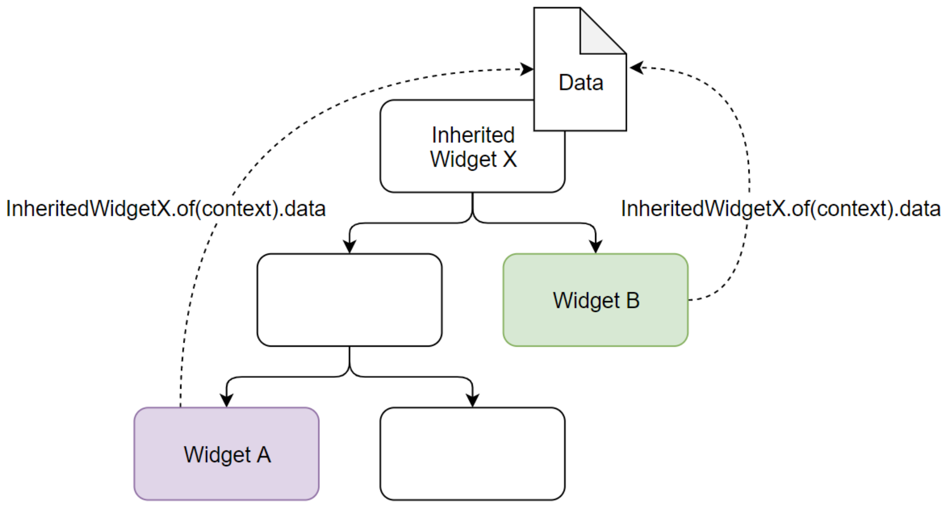

Data
Inherited Widget X
InheritedWidgetX.of(context).data
InheritedWidgetX.of(context).data
Widget B
Widget A

Figure 9 Example of an *Inherited Widget*

4 Developing a Large-Scale Application

This chapter outlines crucial parts of the development process of the *My Thai Star* Flutter front-end component. The design decisions presented are crossroads that most developers using Flutter in a large-scale context will face. As a basis for all these decisions, multiple possible solutions where analysed and compared.

Ultimately ten particularly interesting aspects of the development process where picked and are now highlighted here. These aspects being: the choice of a state management solution, the choice of an architectural style, what model classes are and how to implement them, object equality in Dart, the advantages of using immutable data structures, dependency injection, the structure of the files that make up the project, modularizing aspects of the project, the validation of digital forms and the localization of the application. These aspects will be covered in that order and each of them is assigned a dedicated section of this chapter.

The majority of these sections follow the same structure; the concept at hand will be introduced in a more general context, then that concept will be placed in the scope of large-scale Flutter applications and lastly the *My Thai Star* Flutter application will be presented as an example of the implementation of that concept. The aim of this structure is to make it more obvious how these concepts could be implemented in any large-scale Flutter application.

4.1 State-Management

State-management, in the context of Flutter, refers to the way an application handles the distribution of state throughout the widget tree [11], [71]. It answers the question of how different parts of the application interact with one another and how mutable data is stored and accessed. Thusly choosing a state-management solution is one of the most central decisions when building a Flutter application. It is also one of the earliest decisions made during a given development process, as almost any other part of the application is dependent on what type of state-management solution was chosen. It is crucial to analyse the advantages and disadvantages of possible approaches and picking the right one for the use-case at hand because it is nontrivial to retroactively change the state-management solution later in the development process [23]. This section presents a summary of such an analysis and outlines why one particular solution was chosen for this project. It also gives a detailed description of that solution and outlines by which means it was implemented in the *My Thai Star* Flutter application.

4.1.1 The difference of State-Management and Architecture

Before exploring possible solutions, however, it is important to understand the difference between *"state-management"* and *"architecture"*. These two aspects of an application are tightly coupled but not synonymous. State-management, as outlined above, is the management of the state of an application. This can be done either by directly using Flutter's inbuild tools or through a framework that abstracts those tools [11], [72]. Architecture, on the other hand, is the overarching structure of an application. A set of rules that an app conforms to [73]. Any architecture for a Flutter application will have some sort of state management, but state-management is not an architecture by itself.

Within the Flutter community, *“state-management”* and *“architecture”* are often used synonymously, but I personally believe that it is important to make a distinction.

4.1.2 Choosing a State-Management Solution

Flutter, unlike many other mobile development frameworks [48], [53], does not impose any kind of architecture or state-management solution on its developers. This open-ended approach has led to multiple state-management solutions and a handful of architectural approaches spawning from the community [74]. Some of these approaches have even been endorsed by the Flutter team itself [71]. A detailed comparison of the three most popular approaches has already been conducted in the aforementioned guide [29]. Because of that comparison, the BLoC pattern [32] was chosen as a state-management solution for this thesis. The full analysis is not repeated here, but the findings are summarized in the following table:

Solution	**Origin**	**Short Description**	**Pros**	**Cons**
Provider	Developed by Remi Rousselet in 2018. Now a collaboration of the Flutter Team and Rousselet [75].	A package that provides an improved interface for Flutter’s inbuilt *Inherited Widget*s [69]. Gives the ability to provide state from a widget to all its descendants in the widget tree.	Good for small applications [76], [77]; easy to learn; endorsed by the Flutter team multiple times [71], [78], [79].	Has no related architecture; is not a scalable approach.
Redux	Originally built for React in 2015 by Dan Abramov [53]. Later ported to Flutter by Brian Egan in 2017 [80].	State-management solution with an associated architectural pattern. Uses a *“store”* [81], [82] as the central location for all business logic. All business logic is extracted from the UI and placed in the *store*. The UI should only send actions to the *store* (such as user inputs) and display the UI dependant on the current state of the *store* [81], [82].	Clearly defined rules [82]; state changes are perfectly predictable [76]; can be used to implement a three-layered architecture [81]: *UI – store - data*	The *store* will get very large with large applications [76]; has a high learning curve [76].
BLoC Pattern	Designed by Paolo Soares, one of Google’s own developers, in 2018 [32].	An architectural pattern that functions as a state-management solution. All business logic is extracted from the UI into BLoCs (Business Logic Components) [32]. The UI should only publish events to the BLoCs and display the UI based on the state of the BLoCs.	Has clear architectural rules [32]; was endorsed multiple times by the Flutter team [71], [85]; state changes are perfectly predictable [76]; enables the implementation of a four-layered architecture [83]–[86]: *UI – BLoC – repository - data*	Has a high learning curve [76].

Table 4 Summary of state-management solution comparison

After the aforementioned guide was completed and this comparison was published, an interview with Felix Angelov was conducted to discuss the BLoC pattern in the context of a large-scale application as part of this thesis [23]. During the interview, Angelov stated that he agrees with the verdict of the provider package [75] made by the Flutter guide. He also said that his team originally used a combination of Redux and BLoC. The Redux *store* for application state and BLoCs for low-level state. He explained that they ended up switching the application to exclusively use the BLoC Pattern to reduce the learning curve and on-boarding time for new developers. He went on to say that the BLoC Pattern gave the added advantage of having one BLoC responsible for each use-case instead of having one large *store* responsible for all use-cases. This gave them the ability to further modularize their application. In summary, the interview with Felix Angelov showed that Angelov's observations during his large-scale project at BMW were very much in line with the observations made during the writing of the Flutter guide.

4.1.3 The BLoC Pattern in Depth

Because the BLoC pattern [32] is such a central part of this thesis, this subsection extends the short description in *Table 4*. The aim of this subsection is to provide a more in-depth explanation of how the BLoC pattern functions, what advantages it gives, which rules it imposes and what technology was used to implement it in this thesis. As mentioned in the description above, the BLoC pattern is an architectural pattern originally designed by one of Google's own developers Paolo Soares. In 2018 Soares was implementing the same application with two frameworks based on the Dart [59] programming language: Flutter [20] and Angular Dart [87]. He wanted a way to use the same business logic code on both versions. And thus, the BLoC pattern was created. His goal was to extract all business logic from the framework-specific UI components into framework independent business logic components (BLoCs). Every state change of the application would only be allowed to take place inside of a BLoC. This way both the Flutter and Angular Dart application could use the same BLoCs and code redundancy could be kept to a minimum. With this pattern, the UI only emits events to the BLoCs and displays the widgets based on the state of the BLoCs. Soares also limited the output and input of BLoCs to streams [88]. This way UI updates would be handled consistently across the application. The UI would subscribe to the stream of state provided by a BLoC and change whenever new state was emitted, which also removed the need of manually triggering rebuilds of the UI.

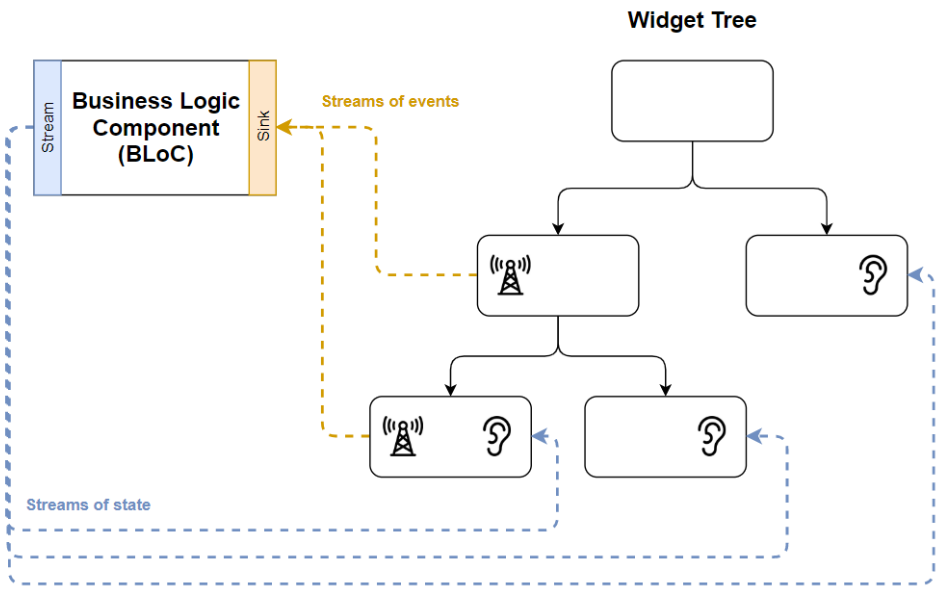


Figure 10 BLoC turning input events into a stream of state

4.1.3.1 *Architectural Advantages of the BLoC Pattern*

Implementing the BLoC pattern in a given application yields the following set of architectural advantages for that application:

(1) Business logic is easy to find / in one place [13], [32], [77], [84].
(2) Business logic is independent of the UI [32].
(3) Business logic is easily testable [84], [89].
(4) Widgets only rebuild when business logic related to them changes [76].
(5) State changes are perfectly predictable and are only allowed to happen in one place [23], [32].

4.1.3.2 *Rules of the BLoC Pattern*

Soares defined a set of eight rules to follow when implementing the BLoC pattern [32]. If these rules are followed during the development process, the given application will gain the advantages outlined in *4.1.3.1*.

Rules for the BLoCs

(1) Input/outputs are simple sinks/streams only.
(2) All dependencies must be injectable and platform agnostic.
(3) No platform branching is allowed.
(4) The actual implementation can be anything if rules 1-3 are followed.

Rules for Widgets

(1) Each *"complex enough"* widget has a related BLoC.
(2) Widgets do not format the inputs they send to the BLoC.
(3) Widgets should display the BLoCs state with as little formatting as possible.
(4) If you do have platform branching, it should be dependent on a single bool state emitted by a BLoC.

4.1.3.3 The BLoC Package

The BLoC package was released by Felix Angelov in October 2018 [31]. It serves as a high-level abstraction of the concepts described above. It limits the boilerplate one might face when implementing the BLoC pattern from scratch and adds some additional features. These features include but are not limited to:

- Exposing an interface to enable the tracking of state changes in BLoCs.
- Enabling the distribution of BLoCs within the widget tree.
    - Realized by maintaining a dependency on the aforementioned provider package [75].
- Providing a simple way for widgets to subscribe to the stream of state emitted by BLoCs.

Furthermore, The BLoC package enforces some additional architectural rules not present in the original BLoC pattern. Firstly, it enforces immutability for all state and event classes. The exact advantages of this are outlined in *Section 4.5*. And secondly, it limits the number of input and output streams of a given BLoC to one. When asked about this decision in the aforementioned interview, Angelov explained that this forces developers to create small BLoCs with single responsibilities. Angelov went on to say that in his experience it is always better to favour small and concise BLoCs over large BLoCs that are responsible for multiple aspects of an application.
Because of the features listed above and because the additional constraints enforced by the package add value to the original definition of the BLoC pattern, the BLoC package is used to implement the BLoC pattern in the *My Thai Star* Flutter implementation.

The following two code snippets show examples of how some of these additional features offered by the package can be implemented in-code. The first snippet shows how the implementation of a BLoC class might look like:

```
///Extents [Bloc] & defines the class of the events it consumes &
///the class of the state it emits
class NewBloc extends Bloc<Event, State> {

  ///Defines the initial state emitted by the [Bloc]
  @override
  State get initialState => State();

  ///Called every time a new event is dispatched to the [Bloc].
  ///
  ///Defines what happens when a new event is consumed.
  @override
  Stream<State> mapEventToState(Event event) async* {

    ///Emits new state like this:
    ///"yield" adds a new value to the stream but does not
    ///terminate the function.
    yield State();
  }
}
```

Code Snippet 6 Implementation of a BLoC using the BLoC package

The second snippet shows how BLoCs can be provided to and accessed from the widget tree:

```
class Page extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return BlocProvider<NewBloc>(
      //Provides a NewBloc to all descendants in the tree
      builder: (BuildContext context) => NewBloc(),
      child: Container(
        child: BlocBuilder<NewBloc, State>(
          //Subscribes to the state emitted by the closest provided NewBloc
          //above this widget in the tree. Rebuilds descendants every time
          //new state is emitted by the BLoC.
          builder: (context, state) {
            //Displays UI based on the state emitted by NewBloc
            if (state is StateA) return WidgetA();
            if (state is StateB) return WidgetB();
            else {
              //Accesses closest provided NewBloc above
              //this widget in the tree and dispatches an event to it
              BlocProvider.of<NewBloc>(context).dispatch(Event());
              return Loading();
            }
          },
        ),
      ),
    );
  }
}
```

Code Snippet 7 Demonstration of widgets using the BLoC package

4.2 Architecture

Choosing and following a clearly defined architecture is an essential part of large-scale software projects; a good architecture will ensure that a given application stays manageable with an ever-growing codebase. As outlined in *Subsection 4.1.2*, an application implementing the BLoC pattern can easily be modified to implement a layered architecture with only a few additional constraints. This section explores what layered architecture is and how it can be implemented with the BLoC pattern. Furthermore, this section gives some insights into how BLoCs were designed during the development of the *My Thai Star* Flutter implementation and gives some generalized tips on how to design BLoCs in any application. And lastly, the layered architecture of the *My Thai Star* Flutter application is presented as an example.

4.2.1 The BLoC Pattern and Layered Architecture

The idea of layering a computer program into hierarchical levels of responsibilities was first described by Edsger W. Dijkstra in 1983 [90]. This type of layered architecture has well established itself over the last 30 years. It has been the central topic of many publications and was used successfully in large-scale applications countless times [91]. The general idea behind this architectural style is to split an application *"horizontally"* into layers that are each responsible for one aspect of the application [92]. An example of this approach is illustrated in the following figure:

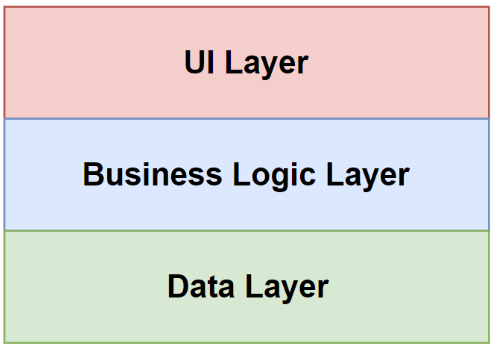


Figure 11 Basic three-layered architecture

The goal is to keep these layers as independent as possible from one another. Changing code in one layer should not affect the functionality of other layers. A common approach to achieve this nowadays is to implement the principle of one-way dependencies [93]. The idea being that code in one layer may depend on code in another layer, but two layers should never depend on each other. Taking *Figure 11* as an example, the UI layer may depend on the business logic layer, but in return, the business logic layer should then not depend on the UI layer. By following this architecture, we gain the following two advantages:

(1) Code with the same responsibility is in one location.
(2) Changes in one layer do not (or only slightly) affect the functionality of other layers.

The BLoC pattern, as outlined in *Section 4.1.3*, provides a solid foundation for implementing a layered architecture in Flutter. An application using the BLoC pattern can be split into the same layers as shown in *Figure 11*. The UI layer consists of all widget files, the business logic layer contains all BLoCs and the data layer is responsible for the communication with external services or devices.
To ensure one-way dependencies in between business logic layer and data layer and to fulfil the second rule for BLoCs, the Flutter community has largely decided on implementing the *"Repository Pattern"* [94] in between those two layers [31], [74], [83], [86]. The *Repository Pattern* solves cases of two classes depending on each other by creating a platform-agnostic interface that these two classes then depend on instead [93]. In the case of Flutter this works as follows:

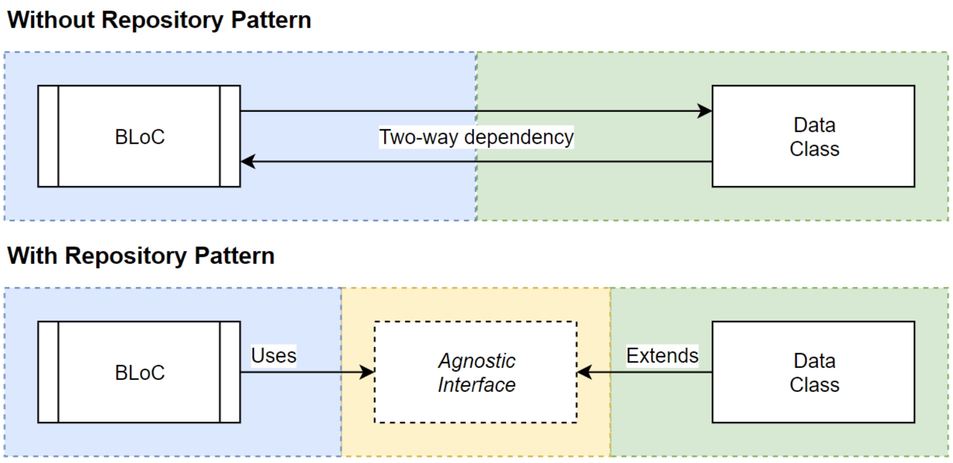


Figure 12 *Repository Pattern* in Flutter

Data classes extend a platform-agnostic interface and BLoCs depend on that interface. BLoCs are then injected with the actual implementation of that interface. The topic of dependency injection will be covered in detail in *Section 4.6*. The *Repository Pattern* ensures that the principle of one-way dependencies is kept intact. The resulting layered architecture is abstracted in the following figure:

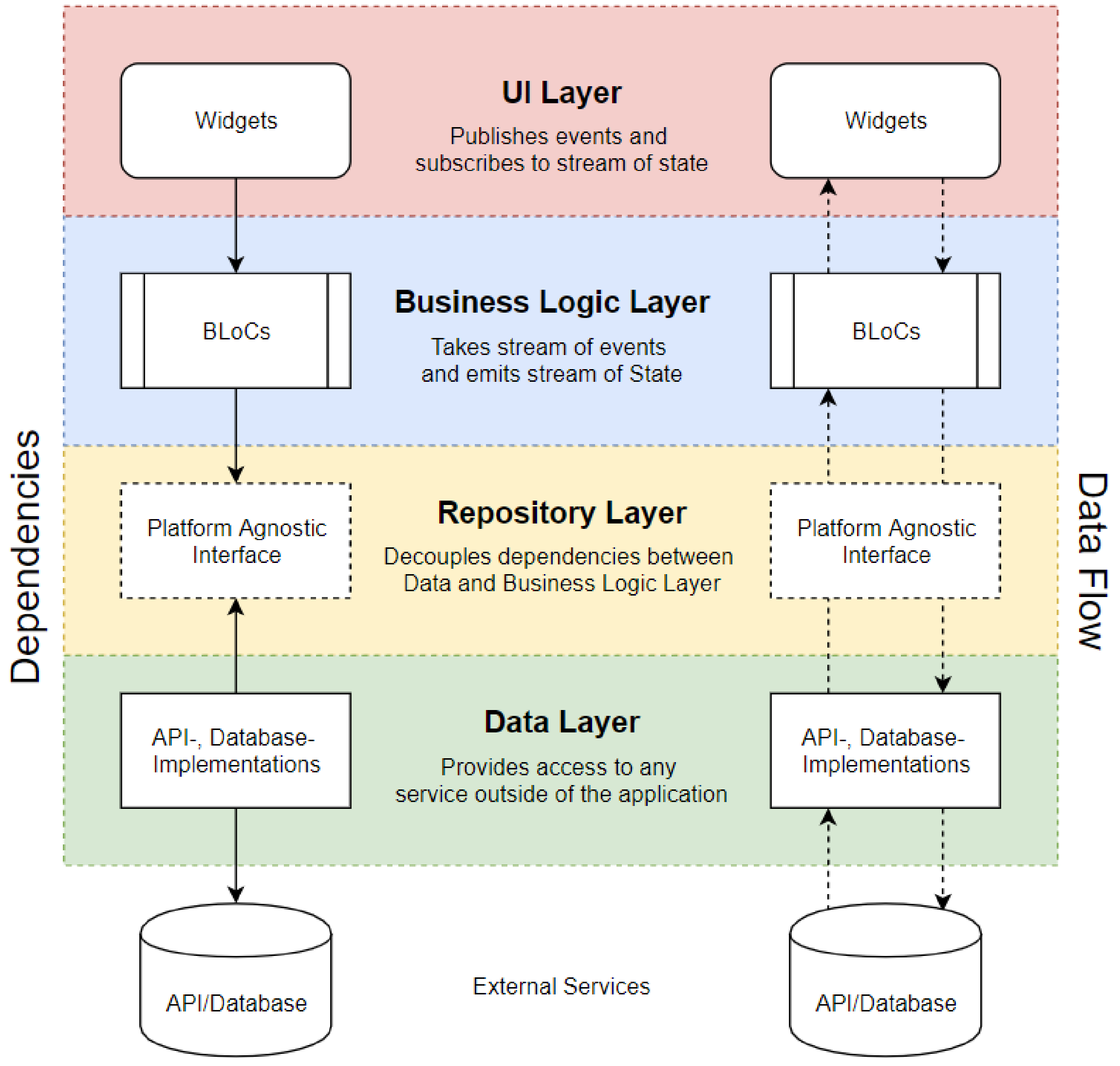

UI Layer
Publishes events and subscribes to stream of state
Widgets
Widgets
Business Logic Layer
Takes stream of events and emits stream of State
BLoCs
BLoCs
Repository Layer
Decouples dependencies between Data and Business Logic Layer
Platform Agnostic Interface
Platform Agnostic Interface
Data Layer
Provides access to any service outside of the application
API-, Database-Implementations
API-, Database-Implementations
API/Database
External Services
API/Database
Dependencies
Data Flow

Figure 13 Four layered architecture using the BLoC pattern [29]

The following table describes each layer in more detail and serves as an extension to the diagram in *Figure 13*. Each layer only consists of one type of file and has one responsibility within the application.

Layer	Responsibility	Consists of
User Interface	Forms the user interface of the application. Contains as little business logic as possible, or as the designer of the pattern puts it: the UI is *"as stupid as possible"* [32]. Takes in user inputs and processes them by emitting events to the BLoCs in the business logic layer. Displays the user interface based on the state of the BLoCs.	Flutter widgets
Business Logic	Contains the business logic of the application. All state changes are defined here. Consumes events that are emitted by the UI layer and mutates state based on those events. New state is then emitted to all classes that subscribe to it. Can only communicate with the data layer through a platform-agnostic interface. The implementation of that interface must be injected into the BLoC (more on this in *Section 4.6*).	BLoCs
Repository	Decouples business logic layer and data layer.	Platform-agnostic interfaces
Data	Communicates with external devices or services. Every class in this layer must extend an interface from the repository layer.	External service/device implementations

Table 5 Layer responsibilities in four-layered architecture using the BLoC pattern

To fulfil the first rule for BLoCs, any communication between UI and business logic layer is limited to streams as shown in the following figure:

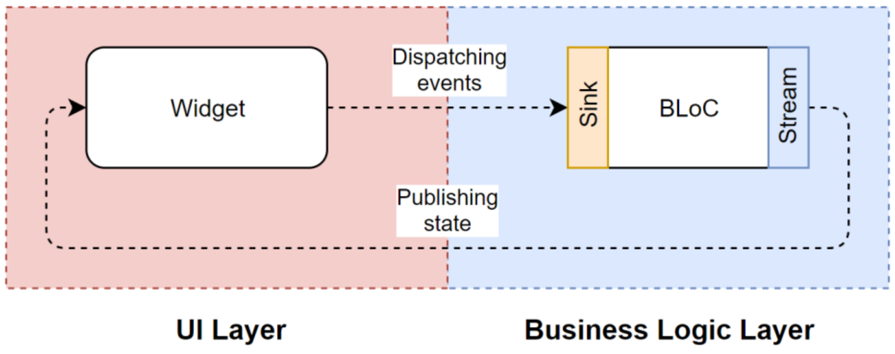


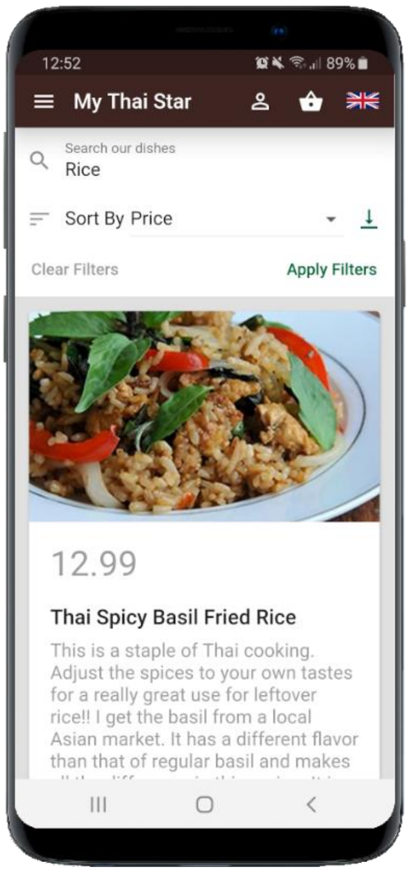
Figure 14 Communication between business logic layer and UI layer [29]

4.2.2 Designing BLoCs

The most challenging part of implementing this architecture in the *My Thai Star* Flutter application was separating the business logic into BLoCs. Defining the size, scope and responsibilities of BLoCs is a non-trivial task. This subsection condenses the findings made during the development process and the recommendations made by Felix Angelov during the aforementioned interview [23]. The aim is to give some tips to developers trying to implement this architecture.
The general approach during the development process was as follows:

(1) Start implementing one *"feature"* as a BLoC. Angelov and his team at BMW define *"feature"* loosely as *"one piece of value given to the customer"* [23]
(2) If this BLoC becomes difficult to handle, split it into two BLoCs.
(3) Go back to step 2.

This is the same process recommended by Felix Angelov [23]. Smaller, more focused BLoCs have the added advantages of being more reusable and easier to test [23]. An example of the process outlined above is the creation of the *“digital menu”* feature in the *My Thai Star* Flutter app. The *Menu Page* of the application displays a list of dishes fetched from the *My Thai Star* back-end component as shown in the following figure. This list is based on a query defined by the user.



This is a staple of Thai cooking. Adjust the spices to your own tastes for a really great use for leftover rice!! I get the basil from a local Asian market. It has a different flavor than that of regular basil and makes

Figure 15 *My Thai Star* Flutter *Menu Page* [43]

This feature was originally implemented as one *“Dish BLoC”*. The UI emitted events containing the complete query. This caused the BLoC to fetch new dishes from the back-end. The *Dish BLoC* would then emit a list of dishes matching the query. It became obvious later in the development, that it was necessary to save the query as state so that it could be preserved when the user left the page. One option would have been to add a member *“last query”* to the state emitted by the *Dish BLoC*, but this would have been an unclean solution. It would have been necessary to check if this member was initialized every time it was accessed by the UI. It became clear that the *digital menu* feature actually consisted of two features: preserving the query and fetching the dishes. The *Dish BLoC* would have been responsible for both. Thus, it was decided to split the old *Dish BLoC* into a *Current Search BLoC* and a new *Dish BLoC*; one responsible for handling the state of the search and one responsible for fetching new dishes. The following figure compares these two implementations on an abstract level:

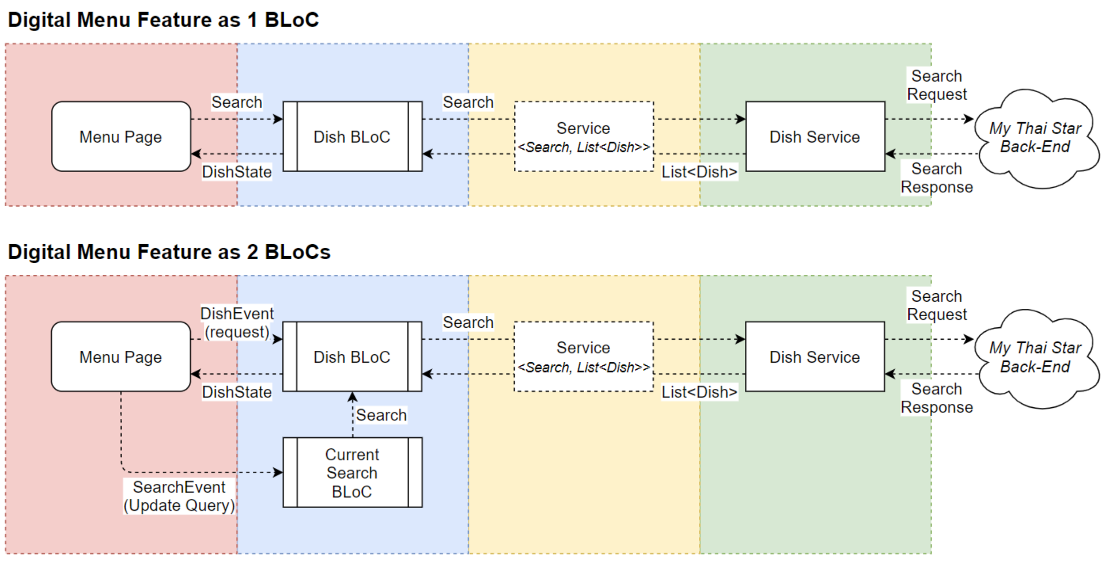


Figure 16 Data flow of the *digital menu* feature in two versions

The second approach gives the added advantage of widgets responsible for requesting more dishes, not needing access to the current search query. The *Dish BLoC* can independently access the current query through its dependency on the *Current Search BLoC*. An example of this are the search bar and the *"Apply Filters"*- button seen in *Figure 15*. The first one can modify the query and thusly only needs access to the *Current Search BLoC* and the second one can request more dishes and only needs access to the *Dish BLoC*.
The event sent to the *Dish BLoC* to trigger the fetching of more dishes can now simply be an *"enum"* [95] because it no longer needs to carry additional information. The exact implementation of the *Dish BLoC* is outlined in *Subsection 4.2.3*.
A given BLoC and the features it is responsible for can theoretically be split into smaller BLoCs almost indefinitely. It is the responsibility of the developer to decide when a given BLoC is at a reasonable size.

4.2.3 Architecture of this Project

This subsection outlines how the architecture described in *Subsection 4.2.1* was implemented in the *My Thai Star* Flutter application. This section ignores BLoCs responsible for the validation of forms and the localization of texts, as these topics are covered in detail in *Section 4.9* and *4.10* respectively. Instead, this section covers the *"book a table"*, *"order food"* and *"digital menu"* feature as outlined in *Section 2.2*.
The following figure shows the architecture from an abstract level. It highlights the dependencies of different classes of the application to one another. Following that figure, this subsection is divided into one heading per layer of the application. Starting with the business logic layer, each of the layers that make up the application will be outlined, and parts of the implementations will be highlighted as examples.

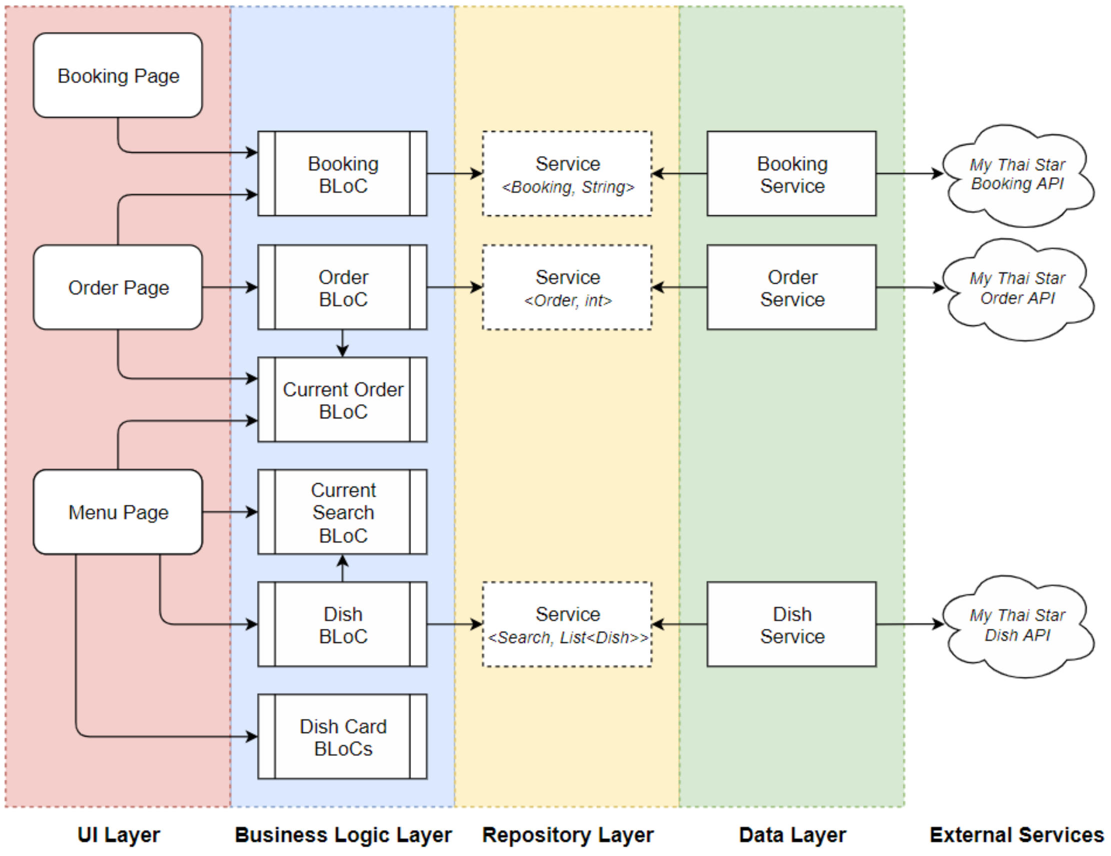


Figure 17 Layered architecture of the *My Thai Star* Flutter app

4.2.3.1 *The Business Logic Layer*

All the BLoCs shown in *Figure 17* were designed with the method outlined in *Subsection 4.2.2*. They all consume events sent to them by the UI layer or other BLoCs and mutate their state based on those events. They then emit that new mutated state as a stream so that any class depending on that state can subscribe to it. The functionality of each BLoC displayed in *Figure 17* is summarized in the following table:

BLoC	**Description**
Booking	Consumes information needed to book a table. Attempts to make that booking and emits state that describes if the booking was successful. If it was, the state contains a booking token.
Current Order	Holds a list of dishes that make up the current order. Consumes events that add/remove dishes from the current order. Emits state describing the current order.
Order	Consumes a booking token and attempts to order the list of dishes provided by the *Current Order BLoC*. The dishes are ordered for the booking related to the booking token. Emits state that describes if the ordering was successful.
Current Search	Holds the current search query. Consumes events that change aspects of that search. Emits state describing the current search.
Dish	Consumes events that trigger the request for more dishes. When such a request is consumed, this BLoC tries to fetch new dishes based on the search defined by the *Current Search BLoC*. It then emits state describing if the fetching was successful. If it was, the state contains a list of dishes matching the current search.

BLoC	Description
Dish Card	Handles the state of one *Dish Card* widget. The state of a *Dish Card* can be modified by selecting or deselecting extras for the given dish. The *Dish Card BLoC* consumes events that add/remove extras. It then emits the new dish as state.

Table 6 Descriptions of the BLoCs in the Flutter *My Thai Star* application

As an example of how such a BLoC might be created, the implementation of the *Dish BLoC* is presented in the following code snippet. As explained in *4.1.3.3*, the BLoC package [31] was used for the implementation of the BLoC pattern in this project.

```
///Gives the ability to request new [Dish]es from
///the [DishBloc].
///
///Is an enum because it does not need to carry any additional data.
///The [DishBloc] already has everything it need because it is injected with
///the [CurrentSearchBloc] on creation.
enum DishEvent { request }

class DishBloc extends Bloc<DishEvent, DishState> {
  final CurrentSearchBloc _searchBloc;

  ///Turns [Search] object into [List<Dish>]
  final Service<Search, List<Dish>> _dishService;

  ///Creates one [DishBloc].
  ///
  ///The dependency on the [_dishService] &
  ///[CurrentSearchBloc] are injected.
  DishBloc({@required searchBloc, @required dishService})
      : _searchBloc = searchBloc,
        _dishService = dishService;

  @override
  DishState get initialState => InitialDishState();

  @override
  Stream<DishState> mapEventToState(DishEvent event) async* {
    if(currentState is LoadingDishState) return;

    Search currentSearch = _searchBloc.currentState.search;

    yield LoadingDishState();

    try {
      yield await _loadDishes(currentSearch);
    } catch (e) {
      yield ErrorDishState(e.toString());
    }
  }

  Future<DishState> _loadDishes(Search currentSearch) async {
    List<Dish> newState = await _dishService.post(currentSearch);
    return ReceivedDishState(newState);
  }
}
```

Code Snippet 8 Implementation of the *Dish BLoC* [43]

The dependencies on the *Current Search BLoC* and on the *Service* that the *Dish BLoC* uses to fetch more dishes are injected into the *Dish BLoC* on creation. How the *Service* class functions exactly, is explained in *4.2.3.3.* As mentioned in *Subsection 4.2.2*, the *Current Search BLoC* is injected into the *Dish BLoC* because it holds the current search query needed for the *Dish BLoC* to fetch more dishes.
Once the *Dish BLoC* receives a *Dish Event (request)*, it emits a *Loading Dish State* to inform all listening widgets that the BLoC is in the process of fetching more dishes. The *Dish Bloc* then attempts to fetch a list of dishes matching the current state of the *Current Search BLoC*. If the service does not throw an exception [96], the fetching was successful and the list is emitted as a *Received Dish State.*
As mentioned in *Subsection 4.2.1*, implementing business logic in this way makes state changes obvious and perfectly predictable. The following finite state diagram [97] shows all possible state transitions of the *Dish BLoC.*

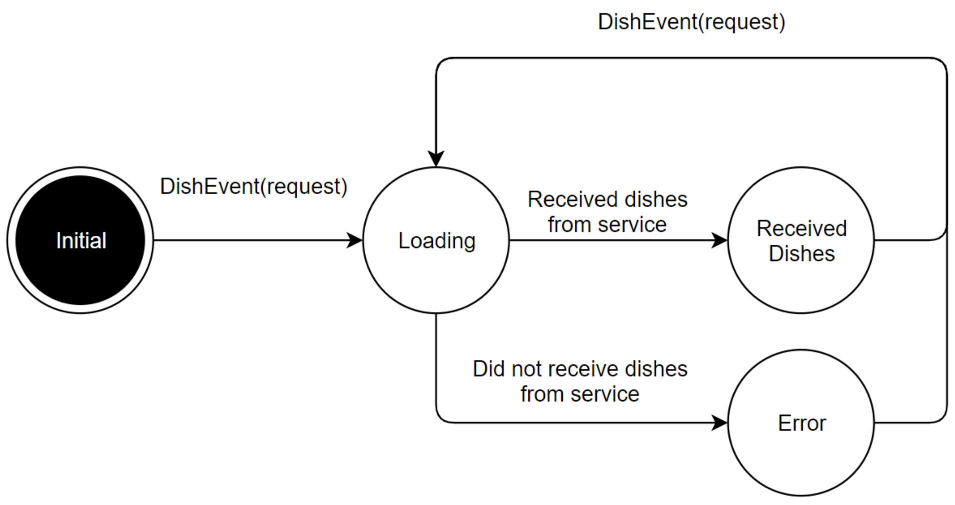


Figure 18 State transitions of the *Dish BLoC*

How the states of the *Dish BLoC* are implemented is shown in the following snippet. It is a common practice to use inheriting classes to implement states and events [31], [74]. This way the name of the class can communicate what the given state/event is and the members of the class can carry any related data. The reason that all states are immutable is explained in *Section 4.5*.

```
///Describes how the process of fetching [Dish]es is going
@immutable
abstract class DishState extends Equatable {}

@immutable
class InitialDishState extends DishState {
  @override
  List<Object> get props => [toString()];

  @override
  String toString() => 'Initial';
}

@immutable
class ReceivedDishState extends DishState {
  final List<Dish> dishes;

  ReceivedDishState(this.dishes);

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

  @override
  String toString() => 'Received/NumberOfDishes: '
      + dishes.length.toString();
}

@immutable
class ErrorDishState extends DishState {
  final String errorMessage;

  ErrorDishState(this.errorMessage);

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

  @override
  String toString() => 'Error/Message: ' + errorMessage;
}

@immutable
class LoadingDishState extends DishState {
  @override
  List<Object> get props => [toString()];

  @override
  String toString() => 'Loading';
}
```

Code Snippet 9 Implementation of the *Dish State* classes [43]

4.2.3.2 The UI Layer

This layer contains the widget tree of the application. As explained in *Subsection 4.2.1*, it can only communicate with the business logic layer through streams of events and state. Instead of displaying every widget that makes up the application in *Figure 17*, it was chosen to group the relevant widgets for a given feature by the pages they are used in: *"Booking Page"*, *"Order Page"*, and *"Menu Page"*. How the UI of these pages looks is shown in *Figure 19*. Each of the pages consists of a tree of multiple nested widgets as explained in *Section 3.3*. Every BLoC of this application is initialized in the UI layer. Whenever a BLoC is needed by more than one widget in the tree, it is initialized and provided from the lowest possible point in the widget tree that is still a common ancestor of all widgets that need access to it.
For example, The *Dish BLoC* is needed by multiple widgets in the *Menu Page*: the *"Apply Filter Button"* needs access to the *Dish BLoC* so that it can emit events to it that force the BLoC to fetch more dishes and the body of the page also needs access to the *Dish BLoC* so that it can display itself to reflect the current state of the *Dish BLoC*. The possible states of the *Dish BLoC* are shown in *Snippet 9*. Visual representations of these states are a list of *Dish Cards* when dishes where fetched successfully, an error text when the fetching was unsuccessful or a loading animation when the fetching is still in progress. To enable these two widgets of the *Menu Page* to access the *Dish BLoC*, it is provided from the lowest common ancestor of those widgets: The *Menu Page*. As previously explained, the *Dish BLoC* is injected with the *Current Search BLoC* on creation. To make this injection possible, they both need to be created and provided from the same point in the widget tree. A detailed explanation of dependency injection is given in *Section 4.6*.

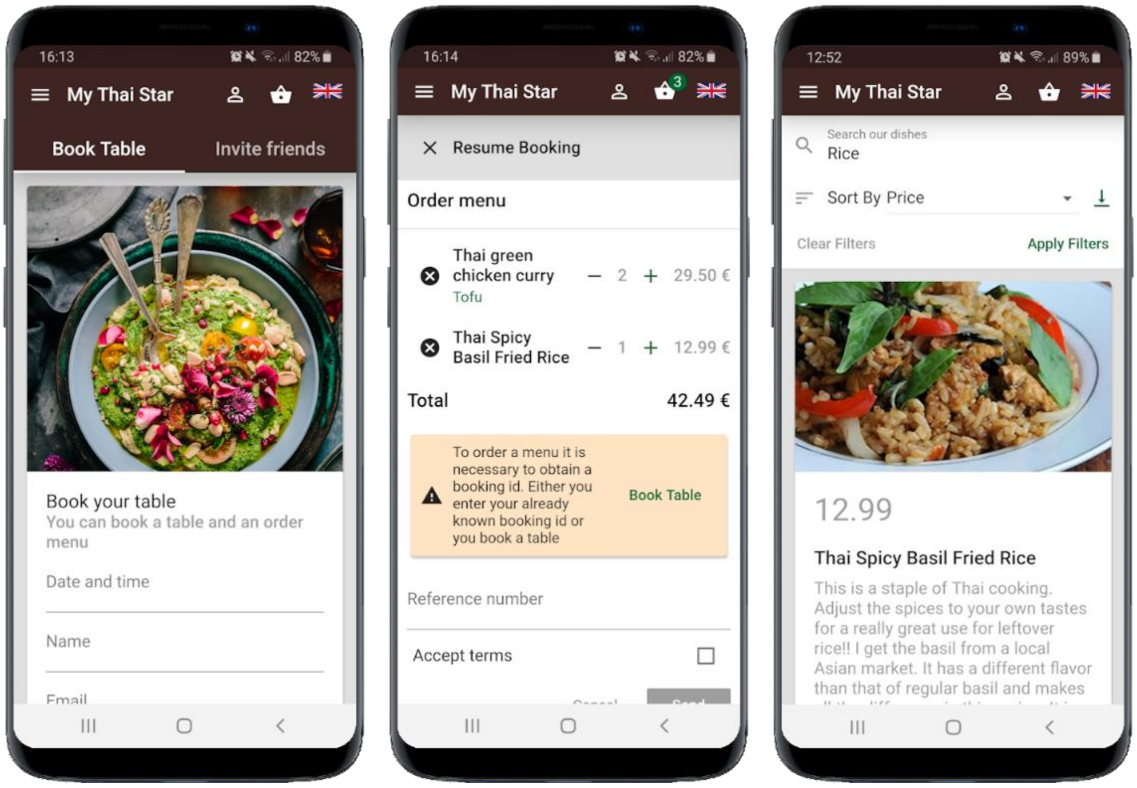

16:13
82%
My Thai Star
Book Table
Invite friends
Book your table
You can book a table and an order menu
Date and time
Name
Email
16:14
82%
My Thai Star
3
Resume Booking
Order menu
Thai green chicken curry
Tofu
2
29.50 €
Thai Spicy Basil Fried Rice
1
12.99 €
Total
42.49 €
To order a menu it is necessary to obtain a booking id. Either you enter your already known booking id or you book a table
Book Table
Reference number
Accept terms
12:52
89%
My Thai Star
Search our dishes
Rice
Sort By Price
Clear Filters
Apply Filters
12.99
Thai Spicy Basil Fried Rice
This is a staple of Thai cooking. Adjust the spices to your own tastes for a really great use for leftover rice!! I get the basil from a local Asian market. It has a different flavor than that of regular basil and makes

Figure 19 *My Thai Star* Flutter *Booking*, *Order* and *Menu – Page* (left to right) [43]

To illustrate the communication of BLoCs and the widget tree, *Figure 20* shows an abstracted version of the widget tree that forms the *Menu Page* and how it interacts with the *Current Search* and *Dish BLoC*.

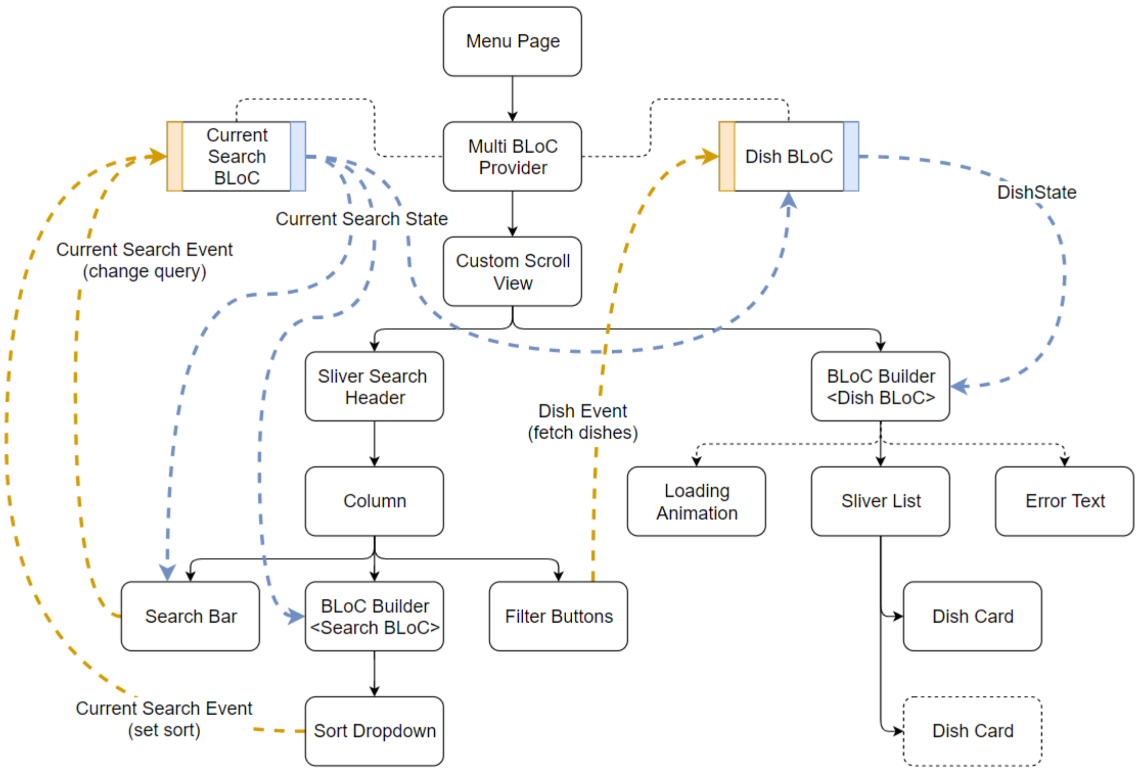


Figure 20 *Menu Page* widget tree with *Current Search* & *Dish BLoC*

The following code snippet illustrates how these BLoCs are initialized in the *Menu Page*. Once the BLoCs are initialized, they can be provided to the rest of the widget tree. That tree can then emit new events and change how it displays itself based on the state of those BLoCs as outlined above. *Code Snippet 11* shows how the widget tree in *Figure 20* is defined in-code.

```
class MenuPage extends StatefulWidget {
  @override
  _MenuPageState createState() => _MenuPageState();
}

class _MenuPageState extends State<MenuPage> {
  CurrentSearchBloc _searchBloc = CurrentSearchBloc();
  DishBloc _dishBloc;

  @override
  void initState() {
    //Injecting dependencies
    _dishBloc = DishBloc(
      searchBloc: _searchBloc,
      dishService: RepositoryProvider.of<RepositoryBundle>(context).dish,
    );
    ...
  }
  ...
}
```

Code Snippet 10 Initialization of the *Dish BLoC* in the *Menu Page* [43]

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: Header(),
    drawer: AppDrawer(),
    body: MultiBlocProvider(
      //Providing Blocs to descendants
      providers: [
        BlocProvider<CurrentSearchBloc>(
          builder: (BuildContext context) => _searchBloc,
        ),
        BlocProvider<DishBloc>(
          builder: (BuildContext context) => _dishBloc,
        ),
      ],
      child: CustomScrollView(
        //CustomScrollView because it contains Widgets of different types
        slivers: <Widget>[
          SliverSearchHeader(),
          BlocBuilder<DishBloc, DishState>(
            builder: (context, DishState state) {
              //Display list based on state
              if (state is ErrorDishState) {
                return _error(state);
              } else if (state is ReceivedDishState) {
                return _list(state);
              } else {
                return _loading();
              }
            },
          ),
         ...
        ],
      ),
    ),
  );
}

///Displays a loading animation wrapped in [SliverFillRemaining]
Widget _loading() {…}

///Displays an error message wrapped in [SliverFillRemaining]
Widget _error(ErrorDishState state) {…}

///Displays a [SliverList] of [DishCard]s
Widget _list(ReceivedDishState state) {
  return SliverList(
    delegate: SliverChildBuilderDelegate(
      (BuildContext context, int index) => DishCard(
        dish: state.dishes[index],
      ),
      childCount: state.dishes.length,
    ),
  );
}
```

Code Snippet 11 Implementation of the *Menu Page* [43]

By extracting all state changes and business logic out of the widget tree like this, the architectural advantages described in *Section 4.1.3* are obtained.

4.2.3.3 The Repository Layer

The repository layer of this application only consists of one interface as all classes in the data layer are similar enough to extend one common interface. The *“Service”* interface only provides one function that exchanges objects of type *Input* for objects of type *Output* asynchronously [98]. The implementation of this interface is shown in the following code snippet:

```
///Use-case independent interface that exchanges
///[Input] for [Output] asynchronously
abstract class Service<Input, Output> {

  ///Exchange one [Input] for a [Future<Output>]
  Future<Output> post(Input input);
}
```

Code Snippet 12 *Service* interface [43]

4.2.3.4 The Data Layer

Each class in the data layer extends a version of the *Service* interface. For example, the *Dish Service* extends the *Service* interface with and *Input* of type *Search* and an *Output* of type *List<Dish>*. This means that the *Dish Service* must provide a *post* function that takes in a *Search* object and returns a list of *Dish* objects asynchronously. The implementation of the *Dish Service* is shown in *Code Snippet 13*. Each of the classes in the data layer is responsible for communicating with one of the services provided by the *My Thai Star* back-end, which are outlined in *Section 2.3*. The *Dish Service, for example*, is responsible for communicating with the “*Dish Management Service*” of the back-end. All classes in the data layer are in their essence an abstraction for back-end services; they provide a high-level interface for the BLoCs to access the services provided by the back-end. For example, the *Dish BLoC* can *post* a *Search* object to its *Dish Service* and then obtain a list of *Dishes* matching that *Search* without needing to know how the actual communication with the back-end is implemented. By implementing the *Repository Pattern* in this manner, three advantages for the application are obtained:

(1) The communication with external services is exclusively happening in one location: the data layer.
(2) BLoCs are solely responsible for business logic and state changes and do not need to additionally implement the communication with external services.
(3) Different implementations of the interface that the BLoCs depend on can be injected into the BLoCs without them changing in behaviour. More on this in *Section 4.6*.

```
@immutable
class DishService extends Service<Search, List<Dish>> {
  static const String _route =
    'mythaistar/services/rest/dishmanagement/v1/dish/search';
  final String _baseUrl;

  static const Map<String, String> _requestHeaders = {
    'Content-type': 'application/json',
    'Accept': 'application/json',
  };

  DishService({@required String baseUrl}) : _baseUrl = baseUrl;

  ///Posts one [Search] to the API and returns [Dish]es
  ///matching that [Search].
  ///
  ///Will throw [Exception]s if the communication with the API fails or
  ///if no matching [Dish]es were found.
  @override
  Future<List<Dish>> post(Search input) async {
    //Turn Search into an object that can be read by the API
    SearchRequest requestBody = SearchRequest.fromSearch(input);
    //Send it to the API
    http.Response response = await http
        .post(
          _baseUrl + _route,
          headers: _requestHeaders,
          body: jsonEncode(requestBody.toJson()),
        )
        .timeout(Duration(seconds: Configuration.defaultTimeOut));

    if (response.body == '') throw Exception("No dishes match that query");

    Map<dynamic, dynamic> respJson = json.decode(response.body);
    SearchResponse searchResponse = SearchResponse.fromJson(respJson);

    if (searchResponse.content == null)
      throw Exception("Received invalid response from Server");

    //If the answer was valid, return the response as a List<Dish>
    return searchResponse.toDishList();
  }
}
```

Code Snippet 13 *Dish Service* implementation [43]

4.3 Model Classes

Every major class type relevant for this project is described in the architecture section except for classes commonly known as *"models"* [83], [99]. These classes are used for modelling objects present in a specific domain. The domain of this project is outlined in *Section 2.2*. The following figure shows the models of this project and their relationships to one another in a UML diagram [100].

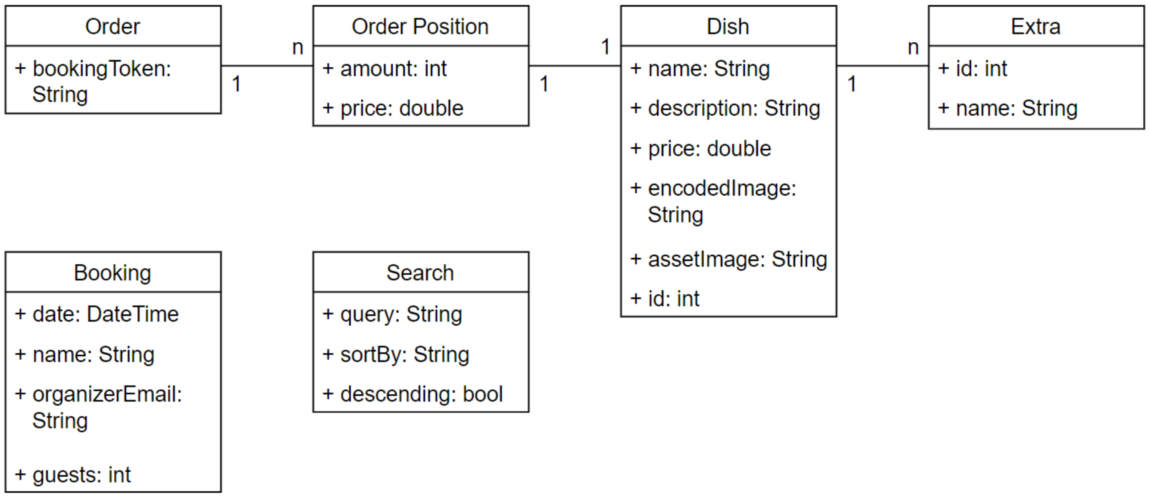


Figure 21 Model classes of the *My Thai Star* Flutter implementation

One might ask why the *Booking* and *Order* model classes have no relation to each other. The domain described in *Section 2.2* clearly states that these two models should be linked through a *booking token.* This linking is, however, a responsibility of the back-end, and it is only relevant for the implementation of the back-end, or in other words, there is no feature implemented in the front-end component that requires the knowledge of this connection. Thus, the front-end model does not need to reflect this relationship.

Models are passed around within and in between the layers described in *Section 4.2*. When the *Menu Page* receives a list of dishes to display, for example, these dishes are models known by the UI and business logic layer. To ensure one-way dependencies [93] models should never be dependent on classes in any of the layers described in *Section 4.2*. To conform with the BLoC pattern described in *Section 4.1.3*, model classes should never contain any business logic.

Paolo Soares, the designer of the BLoC pattern, states that even the formatting of values should be considered business logic [32]. During the development of this project however, multiple occasions presented themselves where it was advantageous to allow models to provide their own values in different formats. Take the implementation of the *Dish* model class, as shown in *Code Snippet 14*. A given *Dish* needs to provide a list of possible *Extras* it can include and whether these *Extras* are selected or not. This behaviour is implemented through a map that links all possible *Extras* to a Boolean value which indicates if it is selected or not. When the UI displays a *Dish* in the current order screen, it needs access to a comma-separated list of all selected *Extras* of a given *Dish*. The solution envisioned by Soares would be to have a BLoC that is solely responsible for turning a *Dish* into a comma-separated list, but this would lead to a lot of boilerplate and worsen the readability of the code. Thus, it was decided to treat such formatting not as business logic in this project.

```
@immutable
class Dish extends Equatable {
  final String name;
  final String description;
  final double price;
  ///Image file encoded as a string
  final String encodedImage;
  ///Location of an asset in the local storage of the app
  final String assetImage;
  final int id;
  final Map<Extra, bool> extras;

  Dish({
    @required this.name,
    @required this.description,
    @required this.price,
    @required this.extras,
    @required this.id,
    this.encodedImage,
    this.assetImage,
  });

  @override
  List<Object> get props => [name, id, extras, selectedExtras()];

  @override
  toString() {
    String res = '\'$name\'';
    res += hasSelectedExtras() ? ' with ' + selectedExtras() : '';
    return res;
  }

  bool hasSelectedExtras() => extras.values.contains(true);

  ///Provides formatted [String] containing the selected [Extra]s
  ///with comma separation
  String selectedExtras() {
    if (!hasSelectedExtras()) return null;

    String res = '';
    extras.forEach((extra, picked) => res +=
      picked ?
      extra.name + ', ' :
      ''
    );
    //Removes trailing comma
    return res.replaceRange(res.length - 2, res.length - 1, '');
  }

  Dish copyWith(…){…}

}
```

Code Snippet 14 *Dish* model class [43]

The reasons for why all model classes extend *"Equatable"* [101] and why all model classes are immutable are given in *Section 4.4* and *Section 4.5* respectively.

4.4 Object Equality

By default, any object in Dart [59] is compared by its reference. The following code snippet contains an example of this behaviour:

```
Extra extra1 = Extra(name: "Tofu", id: 1);
Extra extra2 = Extra(name: "Tofu", id: 1);

print(extra1 == extra2); //False
```

Code Snippet 15 Equality by reference in Dart

This can lead to unwanted behaviour when comparing states or models with one another. For example, in the *My Thai Star* Flutter implementation, a list of available *Dish* objects is created when the user enters the *Menu Page*. When the user adds a *Dish* from that list to the current order, leaves the page, returns to the page and adds what they believe to be the same *Dish* to the order again, these *Dishes* would now be treated as separate entities despite having the same values. Luckily, Dart gives the option to overwrite the comparison behaviour of any object. The *"equatable"* package provides a simple interface to use this language feature [101]. A class that extends *"Equatable"* is compared by its *"props"* array instead of its location in memory. An example of this package being used can be seen in the *Dish* model implementation in *Code Snippet 14*.

All model and state classes in this project extend *Equatable*. This design decision was made for model classes because models are fundamentally about storing information. When comparing information with one another, it is irrelevant where that information is stored, it is only relevant if that information is equal. Or as Rich Hickey puts it in his keynote talk at JaxConf 2012: *"place has no role in an information model"* [102].

State classes extend *Equatable* for the following three reasons: firstly, the BLoC package only notifies the listeners of a given BLoC on state changes. If the package asserts that two states are the same, it will not notify its listeners and the UI will not update. This can lead to problems when comparing states by reference because BLoCs mutate their current state based on events and then emit that mutated state as new state. When comparing states by reference these two states are treated as the same state. Secondly, comparing states based on their values instead of their location improves the testability of the code [89], [102]. And lastly, extending *Equatable* is the recommended approach for creating state classes described in the BLoC package documentation [103].

4.5 Immutability

An object is considered *"immutable"* when its values are unable to change after the object is initialized [61]–[63]. In an object-oriented programming context, this translates to all the member fields of a given class being marked as *"final"*. Such a class will then only produce immutable objects. This section will briefly summarize the main advantages of using immutable objects and then highlight were immutable objects were used for this project.

4.5.1 Advantages

Using immutable objects yields a wide range of advantages [61]–[63], [102]. Some of the most important ones are outlined here. Immutability is in its essence about safety [61], [62], [102]. Once an immutable object is initialized, it can never be altered. It will remain the same no matter through how many functions it passes. Functions that take in an immutable object will produce fewer side effects, as they are unable to influence other functions that take in that same immutable object [61], [62], [102].
Furthermore, immutable objects can never enter an invalid state [61], [62], [102]. Every time a value in a mutable object changes, an assertion must be made whether this change put the object into an invalid state. This is unnecessary for immutable objects, as they never undergo such state changes. The only time such an assertion has to be made for immutable objects is when that object is initialized.

Joshua Bloch, the chief Java architect at Google, advocates in his book *"Effective Java"* [63] to always aim for software, where the highest possible amount of data structures are immutable.

4.5.2 Immutability in this Project

As per Joshua Bloch's recommendation, immutable objects were used wherever possible for this project. This includes but is not limited to models, states, events and the classes in the data layer. Essentially the only classes where immutability was impossible to enforce where the states of *Stateful Widget*s and the BLoC classes. The states of *Stateful Widget*s are mutable by design as outlined in *Subsection 3.4.2*. They are used to initialize/dispose of BLoCs and to preserve BLoCs during rebuilds. And secondly, immutability was impossible to enforce for BLoCs, because they must modify their own state when they consume new events [32].

4.6 Dependency Injection

The term *"Dependency Injection"* (DI) was originally coined by Bob Martin in 1996 [104]. It has since then been a widely covered topic in software design [105], [106]. Paolo Soares even included it as one of his rules for BLoCs: *"All dependencies must be injectable (...)"* [32]. Dependency injection means that if a class A is dependent on an object of class B, class A does not create that object of class B itself. Class A is instead injected with an object of class B. A basic example of DI is shown in the following snippet:

```
class A {
  B b;

  A(this.b);
}
```

Code Snippet 16 Dependency injection example

Firstly, this section outlines what advantages dependency injection provides for a given project. Secondly, a description of one commonly cited way to implement DI in Flutter is given, together with an explanation of why it was not chosen for this project. And lastly, this section outlines which approach was used instead.

4.6.1 Advantages

This design pattern provides the following advantages. Firstly, using DI leads to more reusable code, as the injection of dependencies breaks *"transitive dependencies"* [106]. *Transitive dependencies* occur when a given class A depends on a given class B and class A creates an instance of class B. If this class B depends on a given class C for its initialization, class A is now transitively dependant on class C, because it needs an instance of class C for the initialization of class B. This problem is illustrated in *Figure 22*. The more dependencies a given class has, the less reusable it becomes, thus such *"transitive dependencies"* are to be avoided in most cases.

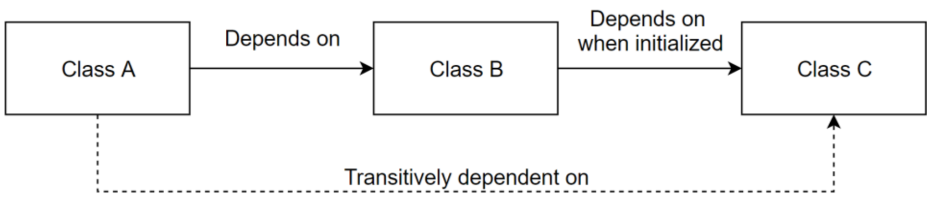


Figure 22 Example of transitive dependencies

Reusability is further amplified when using DI in conjunction with the *Repository Pattern* [94] outlined in *Subsection 4.2.1.* with this pattern, not only are the *transitive dependencies* broken, but also the dependency on the actual implementation of the injected class. Take the *Dish BLoC* as an example, it depends on the *Service* interface outlined in *4.2.3.3*. The actual implementation of the *Service* interface is injected into the *Dish BLoC* on creation. Because all implementations that can be injected into the BLoC share a common interface, the BLoC is independent of how those implementations function. Any injected implementation is treated the same way. In this project, there are two versions of the *Service* that can be injected into the *Dish BLoC*; the actual *Dish Service* that communicates with the back-end and a *Mock Dish Service*, that returns local mock data for testing purposes. No matter which of those implementations is injected into the *Dish BLoC*, it functions the same way. This example is illustrated in the following figure:

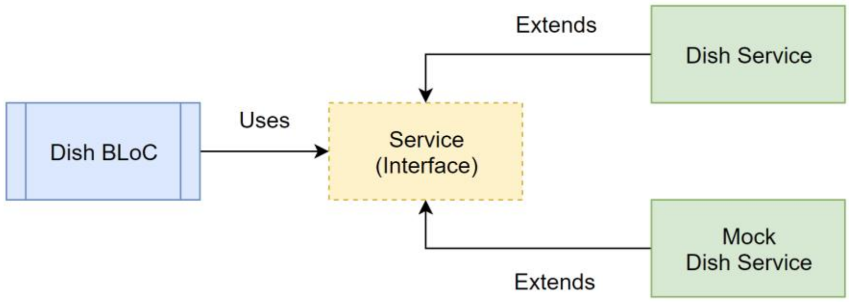


Figure 23 Example of the *Repository Pattern*

In summary, using DI in a given project primarily yields the following two advantages [106]:

(1) More reusable code.
(2) More testable code, as injected dependencies can easily be mocked.

4.6.2 The Inject Repository

There is a multitude of approaches for realizing DI in Flutter. One approach commonly cited by the community [107], [108] is the *"inject"* repository [109] published by Google. It utilizes a series of annotations [42] to generate code while the application is compiling. Classes that should be injected are marked with annotations and the classes that they should be injected into are also marked with annotations. Which classes are injected into which classes is defined in a configuration file.
This approach seems to be promising, as it would be able to move the injection of dependencies completely out of the widget tree. But it was not chosen as the DI solution for this project because it is not yet fully released. Google made a package they use internally public because the Flutter community was eager for a compile-time DI solution [109]. But the inject repository is not at all documented and likely to undergo large changes before it will be fully released.

4.6.3 Dependency Injection in this Project

As mentioned in *Subsection 4.6.1*, dependency injection was primarily used in this project to inject repository implementations into BLoCs. This way BLoCs could be kept completely independent of the data layer. DI was realized by utilizing the *"Repository Provider"* class of the BLoC package [31]. The *Repository Provider* functions very similarly to the *BLoC Provider* described in *Section 4.1.3.3*, in that it provides classes from its position in the widget tree to all its descendants. In the *My Thai Star* Flutter implementation, all repositories are initialized in one central location: inside of the *"Repository Bundle"*. The implementation of the *Repository Bundle* is shown in the following code snippet:

```
///Generates and hold all repositories of the application.
///
///The [RepositoryBundle] will generate all relevant repositories once on
///creation and then hold them in its member variables.
///The [RepositoryBundle] exists to have all repository generation in
///one central location. It will be provided globally using a
///[RepositoryProvider] so it can easily be used to inject dependencies
///into the [Bloc]s of the application.
@immutable
class RepositoryBundle {
  final Service dish;
  final Service booking;
  final Service order;

  RepositoryBundle({@required bool mock, @required String baseUrl})
      : dish = _buildDishRepository(mock, baseUrl),
        booking = _buildBookingRepository(mock, baseUrl),
        order = _buildOrderRepository(mock, baseUrl);

  static Service _buildDishRepository(bool mock, String baseUrl) {
    if (mock) {
      return MockDishService();
    } else {
      return DishService(baseUrl: baseUrl);
    }
  }

  static Service _buildBookingRepository(bool mock, String baseUrl) {
    if (mock) {
      return MockBookingService();
    } else {
      return BookingService(baseUrl: baseUrl);
    }
  }

  static Service _buildOrderRepository(bool mock, String baseUrl) {
    if (mock) {
      return MockOrderService();
    } else {
      return OrderService(baseUrl: baseUrl);
    }
  }
}
```

Code Snippet 17 *Repository Bundle* class [43]

This *Repository Bundle* decides which version of the repositories should be used for the runtime of the application based on the *mock* parameter it is passed on creation. The *mock* parameter is defined in a configuration file, together with some other global configurations for the application. Once the *Repository Bundle* is initialized, it is provided globally to the rest of the widget tree as shown in the next code snippet:

```
@override
Widget build(BuildContext context) {
  return RepositoryProvider<RepositoryBundle>(
    //Provide repositories globally
    builder: (context) => RepositoryBundle(
      mock: Configuration.useMockData,
      baseUrl: Configuration.baseUrl,
    ),
    child: MyThaiStar(), //Create rest of the widget tree
  );
}
```

Code Snippet 18 Providing the *Repository Bundle* globally [43]

Whenever a BLoC with a dependency on a repository is initialized, the relevant widget can access the related repository and inject it as shown in the following snippet:

```
class _OrderFormButtonsState extends State<OrderFormButtons> {
  OrderBloc _orderBloc;

  @override
  void initState() {
    _orderBloc = OrderBloc(
      currentOrderBloc: BlocProvider.of<CurrentOrderBloc>(context),
      orderService: RepositoryProvider.of<RepositoryBundle>(context).order,
    );

    _setUpOrderResponses(_orderBloc);
    super.initState();
  }

  ...
}
```

Code Snippet 19 Injecting dependencies into a BLoC [43]

The approach of having one central location where all injectable dependencies are created is commonly known as *"Service Locator"* [105]. This approach comes with a set of advantages over the approach outlined in *4.6.1*:

(1) It does not rely on an unfinished and undocumented package.
(2) It is easier to understand and reason about because it implements DI without the need for code generation.
(3) It relies on basic Flutter technologies (*Inherited Widget*s [69]) and does not require an additional understanding of other concepts.

The downside is that this approach leads to the UI layer being responsible for the injection of dependencies into the BLoCs. Considering the list of advantages, however, this disadvantage can be considered negligible. Especially because the principle of one-way dependencies [93] is still kept intact as shown in the next figure:

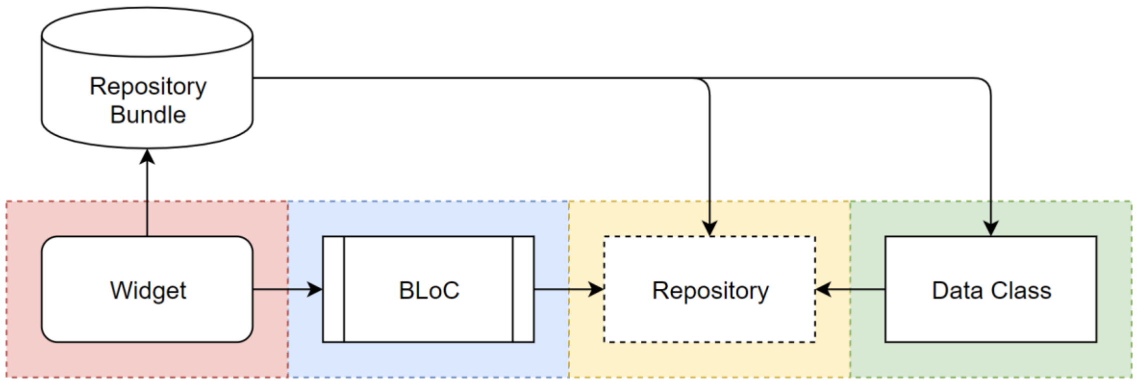


Figure 24 Dependencies of the *Repository Bundle*

4.7 File Structure

The file structure of a project dictates how easily that project can be navigated and how easily different aspects of it can be found. With large codebases, the necessity of these attributes is amplified significantly. It should be obvious to all collaborators where any piece of code is saved and where new pieces of code should be added. This section compares a few different approaches for structuring the files of a Flutter project and explains why one of those approaches was chosen for this thesis.

Bob Martin said in his talk on the *"Principles of Clean Architecture"* in 2015 that the file structure of a given project should reflect the most important aspect of that project [93]. During the research of this thesis, a hand full of different approaches to file structure in Flutter have been identified; each aligning a different aspect of the application with the file structure of that application. Three of these approaches are outlined and compared in the following table:

Name	Structure	Description	Pros	Cons

Component-Based	`lib`
	`├── componentA`
	`├── componentB`
	`├── componentC`
	`├── componentD`
	`│   ├── blocs`
	`│   ├── data`
	`│   ├── models`
	`│   ├── repositories`
	`│   └── ui`
	`└── main.dart`
		This is an approach commonly used in React [53], [110]. The idea is to align the file structure with the different components present in the application. Each component folder contains all BLoCs, models, widgets etc. related to that component.	Makes the application easier to modularize; folders can be turned into separate packages with little effort; all files related to a given feature are in one place.	Not suitable for small applications;

Tree Like	`lib`
	`├── root`
	`│   ├── blocs`
	`│   ├── data`
	`│   ├── models`
	`│   ├── repositories`
	`│   ├── pageA`
	`│   ├── pageB`
	`│   └── pageC`
	`│       ├── blocs`
	`│       ├── data`
	`│       ├── models`
	`│       ├── repositories`
	`│       ├── featureC1`
	`│       └── featureC2`
	`└── main.dart`
		This approach is used by Felix Angelov's team at BMW [23]. In this approach, the file structure is aligned with the widget tree of the application. The different files are placed at a position in the file structure that relates to their position in the widget tree.	The position from where a given BLoC is provided is easily identifiable; developers that know the structure of the widget tree, also know the structure of the project.	High nesting; widgets that are used in multiple locations of the app cannot strictly follow the structure.

Layered	`lib`
	`├── blocs`
	`├── data`
	`├── models`
	`├── repositories`
	`├── ui`
	`└── main.dart`
		This approach is the most commonly cited within the Flutter community [83], [99]. It aligns the layered architecture of the application with its file structure.	Simple to extend; very little nesting.	Many files in a given folder; files relating to one feature are split over multiple locations.

Table 7 Comparison of file structure approaches

In the talk mentioned at the beginning of this section, Martin identified the architecture of an application as its most important aspect [93]. The implementation of any aspect of an application is directly dependent on the architecture that that application follows. Because of this, a layered file structure was chosen for this project. In addition to the approach outlined in *Table 7*, the UI folder was structured to reflect the different pages of the application as shown in the following figure. The header of the application also received its own folder, as it consists of multiple widget files. This way the main disadvantage of the layered file structure approach was minimized. Additionally, a *"shared widgets"* folder was added to house all widgets that are used in multiple locations of the application. The files shown in the following figure, of which the purpose might not be immediately obvious, are explained in *Table 8*.

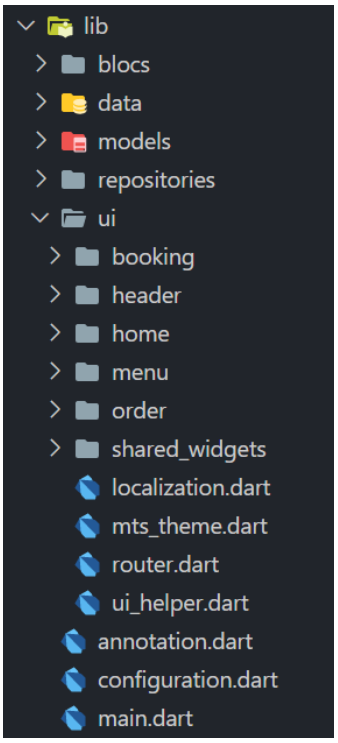


Figure 25 Project structure of the Flutter *My Thai Star* implementation [43]

File Name	Purpose
Localization	Holds the *Localization Delegate* and *Translation* class that are responsible for the localization of texts in this project. This is explained in detail in *Section 4.10*.
MTS Theme	Holds Custom theme data. Defines primary colours, fonts etc.
Router	Responsible for the navigation in-between the pages of the application.
UI Helper	Holds a set of static constant values for margins between widgets.
Annotation	Defines custom annotations [42] for the application so that it is easier to find classes that serve the same purpose or are in similar stages of development.

Configuration	Holds a set of static constant values that make up a list of global configurations for the application as shown in the following snippet:
	 `@immutable`
	 `class Configuration {`
	 `  static final String baseUrl = "http://138.197.218.225:8082/";`
	 `  static final bool useMockData = false;`
	 `  static final bool logging = true;`
	 `  static final int defaultTimeOut = 5; //In Seconds`
	 `}`

Main	Defines the root of the widget tree and the starting point of the application.

Table 8 The purposes of some of the files in the project

4.8 Modularization

Modularization in the context of software development refers to the separation of a system into largely independent components that communicate with each other through strictly defined interfaces [5], [111]. Each component of such a system is then responsible for a specific aspect of that application. For example, one component could handle user authentication and another component could be responsible for data visualization and so on. This section will highlight the advantages of modularization in large-scale applications and how modularization was used in this thesis. Furthermore, this section will explore some additional modularization possibilities that were not utilized for this project.

4.8.1 Advantages

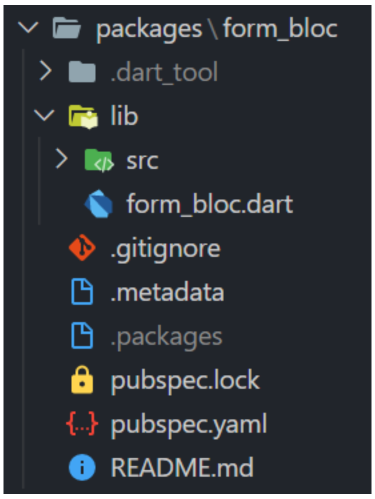
The modularization of code is a proven and well-established way to improve the quality of large-scale software [112]. Separating software into independent components gives the following advantages:

(1) Smaller components are easier to maintain [111].
(2) Components can be re-used [111].
(3) Because components are largely independent, they can be developed by separate development teams [112].
(4) The program can be divided based on its functional aspects [111].

Advantages 1-3 are especially crucial in a large-scale context. Felix Angelov went so far and said that his number one tip for using Flutter in a large-scale application would be to modularize the app as much as possible [23].

4.8.2 Modularization in this Project

Dart gives the ability to modularize an application by extracting a functionality into a separate Dart package [113]. A Dart package is in its structure very similar to a Flutter application. The following figure shows the file structure of the *“Form BLoC”* package that was created during this thesis.



```
packages\form_bloc
  .dart_tool
  lib
    src
    form_bloc.dart
  .gitignore
  .metadata
  .packages
  pubspec.lock
  pubspec.yaml
  README.md
```

Figure 26 *Form BLoC* package file structure [43]

The exact functionality of the *Form BLoC* package is explained in *Section 4.9*. Suffice it to say that the package encapsulates the implementation of form validation in this project. Any Dart file that needs access to the functionality provided by the package can import it like any other package as shown in the following snippet:

```dart
import 'package:form_bloc/form_bloc.dart';
```

Code Snippet 20 Importing the *Form BLoC* package [43]

Form validation is needed for multiple aspects of the *My Thai Star* Flutter application. By extracting it into a separate package, the advantages outlined in *Section 4.8.1* are achieved for this functionality.

4.8.3 Further Possibilities

By using the method described in the previous subsection, it would be possible to extract every feature of the *My Thai Star* Flutter implementation into separate packages. The ordering process could be a package, the menu could be a package and so on. This was not implemented for this thesis however, as it would greatly increase the redundancy between the separate packages. Each package would need its own implementation of the model classes it uses. The menu and order package, for example, would both need their own *Dish* model implementations and so on. One of the central goals of this thesis is to provide a large-scale reference application. The redundancy added by making every feature a separate package would make it a lot more difficult to understand how that reference application functions. Thus, this possibility is merely highlighted here.

4.9 Form Validation

Form validation, in the context of software development, describes the process of validating if a set of values entered by a user into a digital form adhere to a set of predefined rules [114]. Some examples of such values are plain texts, email addresses, phone numbers, integers etc. Any application containing digital forms will have some sort of form validation. Thusly, form validation is a relevant topic for a large number of applications. This project is no exception; form validation is relevant for multiple aspects of the *My Thai Star* Flutter implementation. These aspects being the booking of a table, and the placing of an order, because the user needs to fill out a form with related information before they can complete either of those transactions.
This chapter outlines why Flutter’s inbuild form validation approach was deemed not ideal for this project or any other large-scale Flutter application. Furthermore, this chapter presents an alternative approach for implementing form validation and how this approach was then implemented in this project.

4.9.1 Flutter’s Inbuild Approach

Form validation in any context largely consists of three steps: validating the input of a form field, validating if all fields of a given form are valid and thus the form is now completely filled out and lastly, providing all values entered into the form so that they can be further processed by other parts of the application. Flutter provides tools to realize each of those steps.

Firstly, *"Form Field"* [115] widgets can be used to realize text input. The validation of these *Form Fields* is realized through *"validator"* functions, that take in the inputted value in check if that value adheres to the pre-defined rules [115]. This behaviour is illustrated in the following snippet:

```dart
TextFormField(
  //The validator receives the text that the user has entered
  validator: (value) {
    if (value.isEmpty) {
      return 'Please enter some text'; //Invalid
    }
    return null; //Valid
  },
);
```

Code Snippet 21 Validator function in a *Text From Field* [115]

Secondly, *Form Field* widgets can be assigned to a form through a key [115]. All *Form Fields* with the same key belong to the same form. This key can be used to assert if the whole form is valid.
And lastly, values in the *Form Fields* can be retrieved using *"Text Editing Controllers"* [115]. These controllers are assigned to each field of the form and give access to the current value entered in their associated field. Flutter's inbuild approach has the following disadvantages:

(1) Validator functions cannot be easily reused for multiple fields adhering to the same rules.
(2) This approach breaks the architecture outlined in *Section 4.2*, as state changes and business logic are located in the UI layer.

Because of these disadvantages, a different approach to form validation was developed during the writing of this thesis. This approach is outlined in the following subsection.

4.9.2 The Form BLoC Package

As mentioned in *Subsection 4.8.2*, the *"Form BLoC"* package was developed to realize form validation in this project. The aim of this package is to remove validation logic from the UI layer and instead realize it inside of BLoCs. *Form Field* widgets will still be used in conjunction with this package, but now they are no longer responsible for validation and only serve as a way for the user to input values. *Text Editing Controllers* are no longer used with this approach. Instead BLoCs are responsible for tracking the values inside of the *Form Fields.* As mentioned in the previous section, in order for this package to reflect the entirety of form validation it has to reflect three major steps: validating inputs of *Form Fields*, validating the state of the whole form and providing the inputted values. For this reason, the *Form BLoC* package provides two types of BLoCs: *"Field BLoCs"* and *"Form BLoCs"*, which are described in the following table:

BLoC	Description
Field	Responsible for two things: validating if the input of a *Form Field* is valid and holding on to that entered value. Consumes the value inputted into a *Form Field* and emits state that describes if the current input is valid or not and what that input was.
Form	Has a set of associated *Field BLoCs.* Responsible for two things: asserting if a whole form is valid and providing all values that are currently held by each of its associated *Field BLoCs*. Emits state that describes if the whole form is valid and what values are currently in the form. State is emitted every time any of its *Field BLoCs* changes state. If all associated *Field BLoCs* are valid, the *Form BLoC* considers the whole form valid.

Table 9 BLoCs provided by the *Form BLoC* package [43]

The package provides a set of *Field BLoCs* for common validation use cases such as email, non-empty fields, dates etc. but it also gives the option to create new *Field BLoCs* for other use cases. The following figure illustrates the data flow between *Field BLoCs, Form BLoCs* and the UI layer:

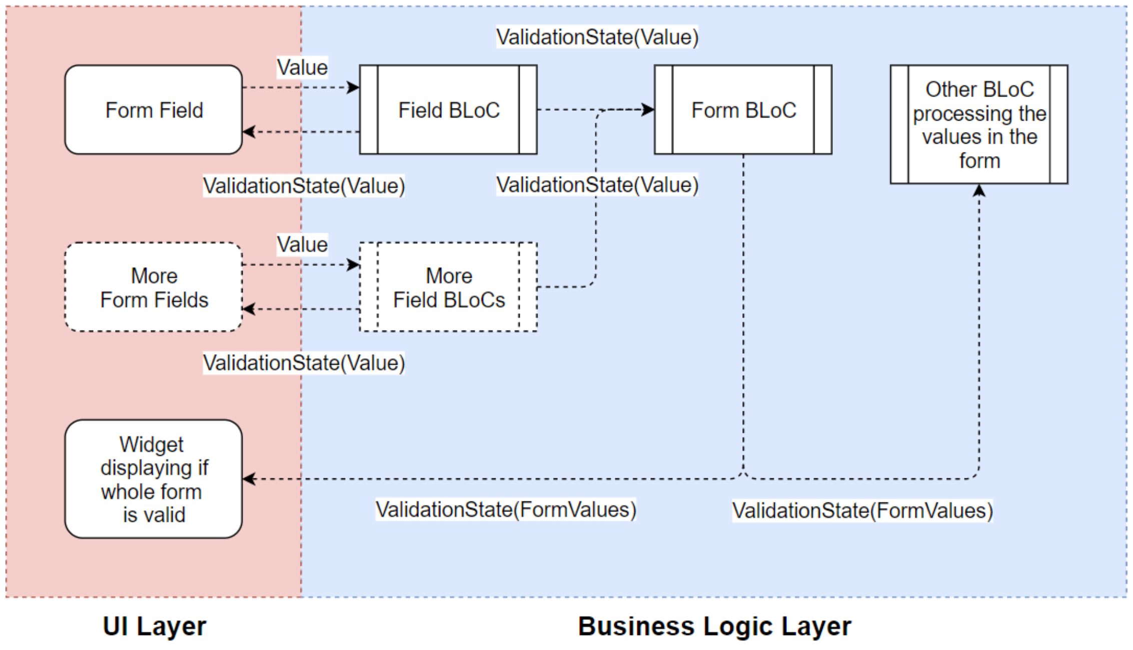


Figure 27 Data flow of the BLoCs provided by the *Form BLoC* package [43]

This approach to form validation is superior to the standard approach provided by Flutter for the following three reasons:

(1) Validation no longer takes place in the UI.
(2) *Form Field BLoCs* can easily be reused for multiple *Form Fields.*
(3) This approach adheres to the architecture described in *Section 4.2* as state changes and business logic are now contained inside of BLoCs.

The main disadvantage of this approach is, that it is more complex then Flutter's inbuilt version. But the advantages listed above make that disadvantage negligible.

4.9.3 Form Validation in this Project

This subsection showcases the two types of BLoCs provided by the *Form BLoC* package in a concrete example: The *Booking Form* of the *My Thai Star* Flutter implementation. As described in *Section 2.2*, the *Booking Form* is filled out when a user wants to book a table at the fictional *My Thai Star* restaurant. The following figure shows how the relevant form validation BLoCs are orchestrated for that specific use case:

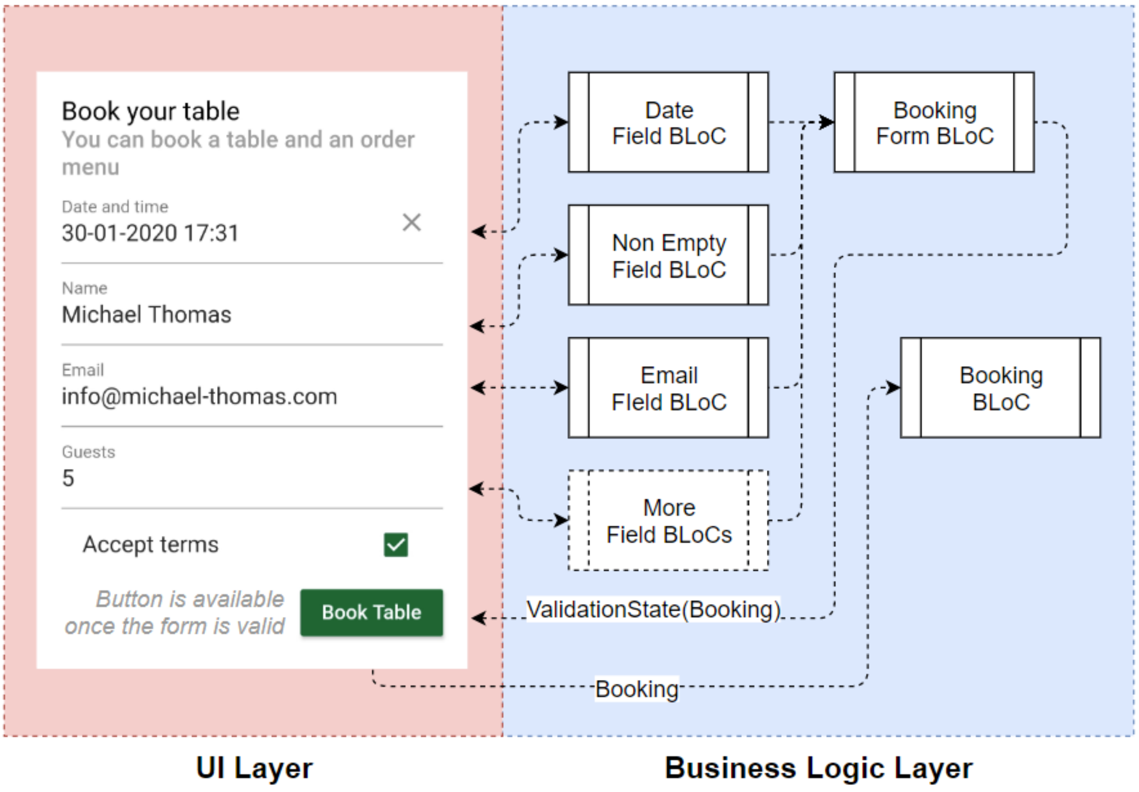


Figure 28 *Booking Form* using the *Form BLoC* package [43]

A new *Form BLoC* needs to be created for each specific use case. This is because *Form BLoCs* emit all current values of the form as an object. How those values are turned into an object differs greatly from use case to use case. For example, the *Booking Form BLoC* emits its values as a *Booking* model object. The implementation of this *Booking Form BLoC* is shown in the following snippet:

```
class BookingFormBloc extends FormBaseBloc<Booking> {
  final EmailFieldBloc _emailBloc;
  final DateFieldBloc _dateBloc;
  final NonEmptyFieldBloc _nameBloc;
  final NumberFieldBloc _guestBloc;

  BookingFormBloc({
    @required emailBloc,
    @required dateBloc,
    @required nameBloc,
    @required guestBloc,
    @required termsBloc,
  })  : _emailBloc = emailBloc,
        _dateBloc = dateBloc,
        _nameBloc = nameBloc,
        _guestBloc = guestBloc,
        super([emailBloc, dateBloc, nameBloc, guestBloc, termsBloc]);

  @override
  ValidationState<Booking> get initialState =>
     InitialState(Booking());

  ///Called every time one of the [FieldBloc]s
  ///of the [FormBaseBloc] changes state
  @override
  Stream<ValidationState<Booking>> mapEventToState(FormEvent event)
    async* {
    ...

    if (isFormValid()) {
      yield ValidState(Booking(
        name: _nameBloc.currentState.data,
        organizerEmail: _emailBloc.currentState.data,
        date: date,
        guests: guests,
      ));
    } else {
      yield InvalidState(Booking(
        name: _nameBloc.currentState.data,
        organizerEmail: _emailBloc.currentState.data,
        date: date,
        guests: guests,
      ));
    }
  }
}
```

Code Snippet 22 *Booking Form BLoC* implementation [43]

The function *"is form valid"* is provided by the package and checks if all related *Field BLoCs* of the *Form Field BLoC* are valid. If they are, the values of the form are emitted as a *Booking* object wrapped in a *Valid State*. If they are not, the incomplete *Booking* object is emitted as an *Invalid State*.

How all form validation BLoCs for the *Booking Form* are initialized in the UI layer is shown in the following snippet. The set of *Field BLoCs* that make up the *Booking Form* are injected into the *Booking Form BLoC* on creation.

```
//Validation: FieldBlocs
EmailFieldBloc _emailBloc = EmailFieldBloc();
DateFieldBloc _dateBloc = DateFieldBloc(DateFormat('dd-MM-yyyy HH:mm'));
NonEmptyFieldBloc _nameBloc = NonEmptyFieldBloc();
NumberFieldBloc _guestBloc = NumberFieldBloc();
CheckboxFieldBloc _termsBloc = CheckboxFieldBloc();

//Validation: FormBloc
YourFormBloc _formBloc;

@override
void initState() {
  _formBloc = YourFormBloc(
    emailBloc: _emailBloc,
    dateBloc: _dateBloc,
    nameBloc: _nameBloc,
    guestBloc: _guestBloc,
    termsBloc: _termsBloc,
  );

  super.initState();
}
...
```

Code Snippet 23 Initializing form validation BLoCs [43]

A *Form Field* Widget implementing this package is shown in the following snippet. In contrast to the approach outlined in *Section 4.9.1*, the validation no longer takes place in the UI. Each *Form Field* has a dedicated *Field BLoC* responsible for its validation.

```
BlocBuilder<FieldBloc, ValidationState>(
  bloc: _formFieldBloc,
  builder: (context, ValidationState state) {
    return TextFormField(
      decoration: InputDecoration(
        labelText: _label,
        errorText: validate(state),
      ),
      onChanged: (String input) => _formFieldBloc.dispatch(input),
    );
  },
);

String validate(ValidationState state) {
   if (state is InvalidState)
      return _errorHint;
   else
      return null;
}
```

Code Snippet 24 *Form Field* using the *Form BLoC* package [43]

Every time the value in the *Form Field* changes, that value is emitted to the *Form Field BLoC*. Every time the *Form Field BLoC* emits new state, the *Form Field* is rebuilt and displays an error text if that state is invalid.

Once the *Booking Form BLoC* asserts that the *Booking Form* is valid, all values entered into the form can be accessed through the state emitted by the *Booking Form BLoC* as a *Booking* object. This object can then be sent to the *Booking BLoC* to place a booking at the fictional *My Thai Star* restaurant as shown in *Figure 28*.

4.10 Localization

Localizing an application for multiple countries is of crucial importance when trying to reach the broadest possible audience with that application. The former analytics company *"Distimo"* [116] tracked the downloads of 200 applications before and after they were localized for an undisclosed number of countries. The results *Distimo* published in 2012 showed that localizing an application for multiple countries can increase the downloads of that application by an average of 128% in only one week [117]–[119]. Because localization is of such importance for the reach of any given application, it is transitively also crucial for large-scale applications.
This section will explore how localization was realized in this project and showcase why the default Flutter approach was chosen instead of rebuilding this feature from scratch with the BLoC pattern.

4.10.1 Localization In this Project

Flutter has an inbuilt method for handling localization [120]. This approach uses *"Localization Delegates"* [120], which are responsible for producing objects that hold the localized data for a given country. In this project, the objects produced by the delegate are called *"Translations"*. The *load* function of the *Localization Delegate* is called whenever the application switches the country it should be localized to. The *load* function then returns a new *Translation* object containing the translated texts that match that new country. The translated texts of this application are stored in JSON files that map an English key to its translation in the given language as shown in the following snippet. It is important that all these JSON files follow the same structure and use the same keys, otherwise accessing different translations of the same text is very difficult.

```
"table": {
    "rowsPage": "Zeilen pro Seite",
    "of": "von",
    "noResults": "Keine Ergebnisse"
  },
...
```

Code Snippet 25 JSON file mapping English keys to German translations [43]

For example, when the user switches the application from English to German, the *load* function would be called with a German *Locale* [121] object, the delegate would then load the German JSON file and return a new *Translation* object containing the information in that JSON file. This behaviour is illustrated in the following snippet:

```
///Generates and then provides a set of [Translation] objects to
///the Widget Tree.
///
///Flutter’s in-build way of handling Localization is through
///[LocalizationsDelegate]s. A custom one can be created by
///extending the [LocalizationsDelegate] class.
@immutable
class MtsLocalizationDelegate extends LocalizationsDelegate<Translation> {
  static const List<String> supportedLanguages =
      ['en','de','bg','es','fr','nl','pl','ru'];

  ///Generates one [Translation] for a given [Locale]
  @override
  Future<Translation> load(Locale locale) async {
    Map<dynamic, dynamic> translationMap = await _loadFromAssets(locale);
    return Translation(locale, translationMap);
  }

  ///Loads a translation JSON file of a given [Locale] form the assets
  ///and parses it to a [Map<dynamic, dynamic>]
  static Future<Map<dynamic, dynamic>> _loadFromAssets(Locale locale) async {
    String jsonContent = await rootBundle
        .loadString('assets/languages/${locale.languageCode}.json');

    return json.decode(jsonContent);
  }
  ...
}
```

Code Snippet 26 *Localization Delegate* of *My Thai Star* Flutter [43]

Localization Delegates are registered in the *Material App* widget [122] at the root of the application as shown in *Code Snippet 27*. The *Material App* widget has the property *“locale”*, which defines the current country the application is localized to. When the *locale* property of the *Material App* is changed, the *load* function of all registered delegates is called with that new *Locale*.

The changing of *Locales* is handled by the *“Localization BLoC”* in this project. Whenever the *Locale* is supposed to be switched, the UI emits an event to the *Localization BLoC* containing the *Locale* that the app should be localized to. This causes that BLoC to emit this new *Locale* as state. The *Material App* widget is subscribed to the state of the *Localization BLoC* and updates its *locale* property whenever this BLoC emits a new *Locale*. How this looks in the *Material App* widget is shown in the following snippet:

```
class MyThaiStar extends StatelessWidget {
  static const String title = 'My Thai Star';

  @override
  Widget build(BuildContext context) {
    return RootProvider(
      child: BlocBuilder<LocalizationBloc, Locale>(
        builder: (context, locale) => MaterialApp(
           //Rebuild whenever the Localization BLoC emits new state
          title: title,
          theme: MtsTheme.data,
          locale: locale, //Update locale
          initialRoute: Router.home,
          onGenerateRoute: (RouteSettings settings) =>
              Router.generateRoute(settings),
          localizationsDelegates: _buildLocalizationDelegates(),
          supportedLocales: MtsLocalizationDelegate.supportedLanguages
              .map((code) => Locale(code))
              .toList(),
        ),
      ),
    );
  }

  ///Build a set of [LocalizationsDelegate]s.
  ///
  ///The list contains both Flutter’s own [LocalizationsDelegate]s for
  ///widgets etc. and the My Thai Star specific delegate.
  Iterable<LocalizationsDelegate<dynamic>> _buildLocalizationDelegates() => [
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
        //This is the My Thai Star specific delegate for handling localized text
        MtsLocalizationDelegate(),
      ];
}
```

Code Snippet 27 *Localization Delegates* in the *Material App* widget [43]

The *Translation* object corresponding to the current *Locale* set in the *Material App* can be accessed through the *Build Context* of any widget that sits below the *Material App* in the widget tree. The call a given widget can use to access the current *Translation* is shown in the following snippet:

```
Localizations.of<Translation>(context, Translation);
```

Code Snippet 28 Accessing the current *Translation* using a *Build Context* [43]

This way any widget can get access to the translated text corresponding to the currently selected *Locale*. Because the current *Translation* is accessed frequently, some helper functions were implemented to shorten the call shown in *Snippet 28* as much as possible while still keeping the meaning of the call obvious. These helper functions are part of the *Translation* class shown in the following snippet:

```dart
///Holds a translation of all My Thai Star texts for one given [Locale]
@immutable
class Translation {
  ///JSON holding the translations
  final Map<dynamic, dynamic> _translationMap;
  final Locale locale;

  const Translation(this.locale, this._translationMap);

  String get languageCode => locale.languageCode;

  ///Returns value in the [_translationMap] of a given path
  String get(String path) {
    String result;
    try {
      List<String> query = path.split('/');
      result = _findInMap(_translationMap, query);
    } catch (e) {
      result = 'Did not find \'$path\'';
    }
    if (result == null) result = 'Did not find \'$path\'';
    return result;
  }

  ///Gives one JSON value that matches the given [query].
  ///
  ///Recursively searches through the map.
  String _findInMap(Map<dynamic, dynamic> map, List<String> query) {
    if (query.length == 1) {
      return map[query.first];
    } else {
      return _findInMap(map[query.first], query.sublist(1, query.length));
    }
  }

  ///Enables a shorter way of accessing the current [Translation]
  static Translation of(BuildContext context) =>
      Localizations.of<Translation>(context, Translation);
}
```

Code Snippet 29 *Translation* class of the *My Thai Star* Flutter implementation [43]

The *"of"* function shortens the call needed to access the current *Translation* object. The *"get"* function provides a simple interface to access data within the JSON map provided by the *Translation* object. The size difference in calls is made obvious by the following snippet. Because these calls are made every time any widget displays any text, every character saved is important.

```
Localizations.of<Translation>(context, Translation);    //Without "of"
Translation.of(context);                                 //With "of"

Translation.of(context).map['menu']['header']['title']; //Without "get"
Translation.of(context).get('menu/header/title');       //With "get"

Localizations.of<Translation>(context, Translation)     //No helpers
  .map['menu']['header']['title'];

Translation.of(context).get('menu/header/title');       //All helpers
```

Code Snippet 30 Space saved by helper functions in the *Translation* class

4.10.2 Localization using exclusively BLoCs

This subsection will briefly explore why utilizing a purely BLoC based approach for localization was deemed not ideal for this or any other large-scale project. Before the solution outlined in *Subsection 4.10.1* was implemented, an attempt was made to realize localization without *Localization Delegates*. In this approach, the *Localization BLoC* emitted *Translation* objects directly and the entire UI subscribed to the state of the *Localization BLoC* in order to update its texts.

One could argue that in the approach described in *4.10.1* some business logic and state changes are handled by the underlying framework and it thusly breaks the architecture described in *Subsection 4.2*. With a purely BLoC based approach, these state changes would instead happen inside of a BLoC and it would thusly be more in line with that architecture.

The problem with a purely BLoC based approach, however, is that *Stateful Widgets* still preserve their state even if the entire widget tree is forced to update. Thusly placing a *BLoC Builder* that subscribes to the state of the *Localization BLoC* at the root of the application would only force all *Stateless Widgets* to update their texts and *Stateful Widgets* would remain unchanged. The only solution would be to additionally place *BLoC Builders* inside of all *Stateful Widgets* which would cause a large amount of additional overhead. Because of this overhead, a purely BLoC based solution was deemed not ideal.

5 Conclusion

This chapter outlines how the central goals of this thesis were fulfilled and what the exact contributions of this project to the Flutter community are. Secondly, this chapter will reflect on the work that was done and evaluate the strengths and weaknesses of this project. And lastly, a few suggestions for future related work are made.
This thesis had two central goals as outlined in *Section 1.1*: to document crucial steps in the development process of a large-scale Flutter application and to generate such a large-scale application as a reference project. In order to fulfil these goals a number of steps were taken.

This project was prefaced with the creation of a guide on using Flutter in a large-scale context. The guide condensed a wide range of community sources and scientific papers into a streamlined guideline for developing scalable Flutter applications. The guide was well received by the Flutter community after its release through Capgemini's DevonFw open-source initiative. The knowledge generated during the writing of the guide was then extended upon by an interview with Felix Angelov, an expert in the field. After a wide enough knowledge base was established, and the domain of the *My Thai Star* reference application was analysed, the development of a large-scale Flutter application began. During the development process, a wide range of obstacles were faced, analysed and overcome.

Ten particularly interesting aspects of the development process were identified and then described in detail in this thesis. The descriptions of these aspects were written in a way that makes them easily transferable into other domains and other large-scale Flutter applications. The design decisions presented are crossroads that most developers using Flutter in a large-scale context will face. The possible options for overcoming those crossroads were evaluated, analysed and finally, one of the options was chosen. These evaluations will save peers facing the same problems: time and resources.

Secondly, the *My Thai Star* Flutter application was developed and published during the writing of this thesis. It is an open-source, large-scale project that is fully documented and follows all the implementation recommendations made by this thesis. Peers can use it as a reference when building their own Flutter applications on a similar scale.

Additionally, even if it was not one of the original goals, a novel solution for realizing form validation with the BLoC pattern was created and published as a Dart package. Peers may utilize this package when implementing the same architectural style and can thusly yield the benefits of this approach as outlined in *Section 4.9.2*.

5.1 Reflection

This section is a reflection on the work process, and it highlights some of the strengths and weaknesses of the project. All in all, this thesis satisfies the goals set at the beginning of the writing process. Three contributions to the Flutter community are made and with the community continuing to grow, these contributions will only become more relevant.
However, this project does have a few limitations.

Firstly, only one interview with an expert in the field could be conducted. In a perfect scenario, multiple experts using Flutter in a wide range of domains and contexts would have been interviewed and their different recommendations would have been compared. But because Flutter is still such a new framework, it was difficult to find people using it in a large-scale context, as very few companies have fully adapted this new technology.

Secondly, because Flutter is a comparatively new solution, next to no scientific research has so far been conducted on it. As of the writing of this thesis, there has not been a single publication on the Flutter Framework through IEEE [123] or ACM [124]. Thus, the scientific sources of this thesis are limited to more general topics such as cross-platform development, software architecture, the benefits of immutable data types etc.

Lastly, during the writing of this thesis, it became obvious that the term *"large-scale"* is ill-defined within the software development community. Different outlets seem to have a vastly different understanding of what *"large-scale"* is referring to exactly [125], [126]. Looking back at the conducted work now, the term *"scalable"* might have been a better choice to describe the created application, as this term is more clearly defined and might fit the context of this thesis better. This problem was mitigated, however, by a detailed glossary entry and an explanation of what *"large-scale"* is referring to in the context of this project in *Section 2.4.*

5.2 Future Work

Two topics outside of the scope of this thesis that might be interesting for future research endeavours were discovered during the writing process: firstly, so far there have not been any empirical studies of the performance of Flutter applications in comparison to other cross-platform approaches. Flutter claims to deliver native performance and on a subjective level the applications do *feel* native, but a proper empirical study backing up this claim is still necessary.
Secondly, the possibility of fully modularizing a Flutter application was discussed during the Angelov interview [23]. The approach he suggested is very similar to the currently highly popular micro-service approach [127]. Each of the modules of the application would be responsible for all aspects of one feature of the application: user interface, business logic, communication with external services etc. Angelov stated that his team at BMW is currently working on modularizing their Flutter application to such a degree as to where each feature could theoretically run on its own in an independent application. Fully exploring this topic and its feasibility could be very interesting for large-scale Flutter applications in the future.

References

[1] TU Dresden, ‘Declaration of originality’, *TU Dresden*. [Online]. Available: https://tu-dresden.de/bu/bauingenieurwesen/studium/im-studium/access/copy2_of_index. [Accessed: 17-Sep-2019]

[2] T. Y. Adinugroho, Reina, and J. B. Gautama, ‘Review of Multi-platform Mobile Application Development Using WebView: Learning Management System on Mobile Platform’, *Procedia Comput. Sci.*, vol. 59, pp. 291–297, Jan. 2015, doi: 10.1016/j.procs.2015.07.568.

[3] Google LLC, ‘Android Canvas’, *Android Developers*, 2007. [Online]. Available: https://developer.android.com/reference/android/graphics/Canvas?hl=de. [Accessed: 30-Dec-2019]

[4] Apple, ‘Quartz 2D’, 2007. [Online]. Available: https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html. [Accessed: 30-Dec-2019]

[5] P. Krutchen, ‘Component Definition’, 2012. [Online]. Available: https://wiki.c2.com/?ComponentDefinition. [Accessed: 06-Jan-2020]

[6] B. Hookway, *Interface*. The MIT Press, 2014 [Online]. Available: https://www.vitalsource.com/de/products/interface-branden-hookway-v9780262322638. [Accessed: 24-Jan-2020]

[7] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, ‘A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development’, *ACM Comput Surv*, vol. 51, no. 5, pp. 108:1–108:34, Nov. 2018, doi: 10.1145/3241739.

[8] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, ‘Comparing Cross-Platform Development Approaches for Mobile Applications’, presented at the International Conference on Web Information Systems and Technologies, 2012, vol. 2, pp. 299–311, doi: 10.5220/0003904502990311 [Online]. Available: http://www.scitepress.org/documents/2012/39045. [Accessed: 27-Dec-2019]

[9] Techopedia, ‘What is a Platform?’, *Techopedia.com*. [Online]. Available: https://www.techopedia.com/definition/3411/platform. [Accessed: 22-Jan-2020]

[10] D. A. Spuler and A. S. M. Sajeev, ‘Abstract Compiler Detection of Function Call Side Effects’, James Cook University To wnsville, QLD 4811 Australia, 1994 [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.2096&rep=rep1&type=pdf

[11] Flutter Dev Team, ‘Flutter State’, 2019. [Online]. Available: https://flutter.dev/docs/development/data-and-backend/state-mgmt. [Accessed: 22-Sep-2019]

[12] Joint working group NAErg/NIA, ‘DIN EN ISO 9241-210 Ergonomics of human-system interaction’. Din, 2019 [Online]. Available: https://www.din.de/de/mitwirken/normenausschuesse/naerg/entwuerfe/wdc-beuth:din21:302206360. [Accessed: 24-Jan-2020]

[13] D. Boelens, ‘Widget — State — BuildContext — InheritedWidget’, *Medium*, 2018. [Online]. Available: https://medium.com/flutter-community/widget-state-buildcontext-inheritedwidget-898d671b7956. [Accessed: 23-Sep-2019]

[14] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, ‘Cross-platform Model-driven Development of Mobile Applications with Md2’, in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2013, pp. 526–533, doi: 10.1145/2480362.2480464 [Online]. Available: http://doi.acm.org/10.1145/2480362.2480464. [Accessed: 27-Dec-2019]

[15] A. Ribeiro and A. R. da Silva, ‘Survey on Cross-Platforms and Languages for Mobile Apps’, in *2012 Eighth International Conference on the Quality of Information and Communications Technology*, 2012, pp. 255–260, doi: 10.1109/QUATIC.2012.56.

[16] T. A. Majchrzak, A. Biørn-Hansen, and T.-M. Grønli, ‘Progressive Web Apps: the Definite Approach to Cross-Platform Development?’, *Hawaii Int. Conf. Syst. Sci. 2018 HICSS-51*, Jan. 2018 [Online]. Available: https://aisel.aisnet.org/hicss-51/st/mobile_app_development/7

[17] M. Latif, Y. Lakhrissi, E. H. Nfaoui, and N. Es-Sbai, ‘Review of mobile cross platform and research orientations’, in *2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS)*, 2017, pp. 1–4, doi: 10.1109/WITS.2017.7934674 [Online]. Available: https://ieeexplore.ieee.org/document/7934674

[18] A. Ebone, Y. Tan, and X. Jia, ‘A Performance Evaluation of Cross-Platform Mobile Application Development Approaches’, in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 92–93 [Online]. Available: https://ieeexplore.ieee.org/document/8543442

[19] Google LLC, ‘Google Home Page’, 1998. [Online]. Available: https://about.google/?fg=1&utm_source=google-DE&utm_medium=referral&utm_campaign=hp-header. [Accessed: 22-Jan-2020]

[20] Flutter Dev Team, *The Flutter Framework*. Google LLC, 2018 [Online]. Available: https://flutter.dev/. [Accessed: 20-Sep-2019]

[21] Google LLC, ‘Google Trends: Flutter & React Native’, *Google Trends*, 2019. [Online]. Available: https://trends.google.de/trends/explore?date=today%205-y&q=React%20Native,Flutter. [Accessed: 16-Oct-2019]

[22] BMW, ‘BMW Official Website’, 1916. [Online]. Available: https://www.bmw.co.uk/en/index.html. [Accessed: 07-Jan-2020]

[23] F. Angelov, ‘Interview about the BLoC Pattern and Flutter in large-scale applications’, 12-Sep-2019 [Online]. Available: https://www.dropbox.com/s/cq31trdnlm0noe2/BLoC%20Pattern%20and%20Flutter%20in%20large-scale%20applications%20%28Felix%20Angelov%2009-12-2019%29.zip?dl=0

[24] D. Ramel, ‘LinkedIn Data Says Flutter Is Fastest-Growing Skill Among Software Engineers’, *ADTmag*, 29-Mar-2019. [Online]. Available: https://adtmag.com/articles/2019/03/29/linkedin-skills.aspx. [Accessed: 27-Dec-2019]

[25] Technical University Cologne, ‘Technical University Cologne’, 2019. [Online]. Available: https://www.th-koeln.de/en/homepage_26.php. [Accessed: 20-Sep-2019]

[26] Capgemini, ‘Capgemini - Home Page’, 2019. [Online]. Available: https://www.capgemini.com/us-en/. [Accessed: 20-Sep-2019]

[27] Capgemini, ‘Devonfw’, 2019. [Online]. Available: https://devonfw.com/index.html. [Accessed: 13-Nov-2019]

[28] S. J. Linares, D. R. Gonzalez, and Contributors, *My Thai Star*. devonfw, 2019 [Online]. Available: https://github.com/devonfw/my-thai-star. [Accessed: 13-Oct-2019]

[29] S. Faust, ‘Flutter Guide’, *GitHub*, 2019. [Online]. Available: https://github.com/devonfw-forge/devonfw4flutter. [Accessed: 14-Oct-2019]

[30] GitHub Inc., ‘Github’, *GitHub*, 2008. [Online]. Available: https://github.com. [Accessed: 17-Oct-2019]

[31] F. Angelov and Contributors, ‘Bloc Package’, Oct-2018. [Online]. Available: https://felangel.github.io/bloc/#/. [Accessed: 12-Sep-2019]

[32] P. Soares, ‘Flutter / AngularDart – Code sharing, better together’, Google Campus, LA, 25-Jan-2018 [Online]. Available: https://www.youtube.com/watch?v=PLHln7wHgPE. [Accessed: 12-Sep-2019]

[33] Oracle, *Java JDK*. Oracle, 1996 [Online]. Available: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. [Accessed: 26-Sep-2019]

[34] Ryan Dahl, ‘Node.js’, *Node.js*, 2011. [Online]. Available: https://nodejs.org/en/. [Accessed: 22-Jan-2020]

[35] Microsoft, .‘NET’, 2001. [Online]. Available: https://dotnet.microsoft.com/. [Accessed: 22-Jan-2020]

[36] ECMA, *JavaScript ECMA Standard*. ECMA, 1997 [Online]. Available: https://www.ecma-international.org/publications/standards/Ecma-262.htm. [Accessed: 26-Sep-2019]

[37] World Wide Web Consortium (W3C), ‘Extensible Markup Language Definition’, 26-Nov-2008. [Online]. Available: https://www.w3.org/TR/xml/. [Accessed: 28-Dec-2019]

[38] Google LLC, ‘Angular’, 2016. [Online]. Available: https://angular.io/. [Accessed: 06-Oct-2019]

[39] T. Müller, *H2 Database*. 2005 [Online]. Available: https://www.h2database.com/html/main.html. [Accessed: 28-Dec-2019]

[40] Docker, Inc., *Docker*. 2013 [Online]. Available: https://www.docker.com/. [Accessed: 28-Dec-2019]

[41] R. Fielding, U. Irvine, and J. Gettys, ‘Hypertext Transfer Protocol -- HTTP/1.1’, *Hypertext Transfer Protocol -- HTTP/1.1*. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616.html. [Accessed: 19-Sep-2018]

[42] Dart Team, ‘Meta library’, 2018. [Online]. Available: https://api.flutter.dev/flutter/meta/meta-library.html. [Accessed: 25-Jan-2020]

[43] S. Faust, *Flutter implementation of My Thai Star*. Germany, 2019 [Online]. Available: https://github.com/devonfw-forge/devonfw4flutter-mts-app. [Accessed: 29-Dec-2019]

[44] Computerphile, *HTML IS a Programming Language (Imperative vs Declarative)*. University of Nottingham, 2016 [Online]. Available: https://www.youtube.com/watch?v=4A2mWqLUpzw. [Accessed: 25-Sep-2019]

[45] W. Leler, ‘What’s Revolutionary about Flutter’, *hackernoon*, 2017. [Online]. Available: https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514. [Accessed: 22-Sep-2019]

[46] R. Nunkesser, ‘Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App Development’, in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 214–218.

[47] Apple, *iOS SDK*. Apple, 2010 [Online]. Available: https://developer.apple.com/ios/. [Accessed: 25-Sep-2019]

[48] Google LLC, *Android SDK*. Google LLC, 2008 [Online]. Available: https://developer.android.com/. [Accessed: 25-Sep-2019]

[49] Google LLC, *How is Flutter different for app development*. Google Developers Official Youtube Channel, 2019 [Online]. Available: https://www.youtube.com/watch?v=l-YO9CmaSUM&feature=youtu.be. [Accessed: 19-Sep-2019]

[50] S. Stoll, ‘In plain English: So what the heck is Flutter and why is it a big deal?’, *Medium*, 2018. [Online]. Available: https://medium.com/flutter-community/in-plain-english-so-what-the-heck-is-flutter-and-why-is-it-a-big-deal-7a6dc926b34a. [Accessed: 22-Sep-2019]

[51] World Wide Web Consortium (W3C), ‘HTML’, 1994. [Online]. Available: https://www.w3.org/html/. [Accessed: 15-Oct-2019]

[52] World Wide Web Consortium (W3C), ‘Cascading Style Sheets’, 1994. [Online]. Available: https://www.w3.org/Style/CSS/Overview.de.html. [Accessed: 15-Oct-2019]

[53] Facebook, *React Native Framework*. Facebook, 2015 [Online]. Available: https://facebook.github.io/react-native/. [Accessed: 22-Sep-2019]

[54] N. Couvrat, ‘What Happens When My React Native Application Starts?’, 2018 [Online]. Available: https://www.youtube.com/watch?v=rReCzR6DMEM. [Accessed: 22-Jan-2020]

[55] T. Kol, ‘Performance Limitations of React Native and How to Overcome Them’, Amsterdam, 2017 [Online]. Available: https://www.youtube.com/watch?v=psZLAHQXRsI. [Accessed: 22-Sep-2019]

[56] Flutter Dev Team, ‘Platform Channels’, 2018. [Online]. Available: https://flutter.dev/docs/development/platform-integration/platform-channels. [Accessed: 22-Jan-2020]

[57] Flutter Dev Team, ‘FAQ - Flutter’, 2019. [Online]. Available: https://flutter.dev/docs/resources/faq. [Accessed: 22-Sep-2019]

[58] Flutter Dev Team, ‘Flutter Widgets’, 2019. [Online]. Available: https://flutter.dev/docs/development/ui/widgets-intro. [Accessed: 25-Sep-2019]

[59] Dart Team, ‘Dart programming language’, 2019. [Online]. Available: https://dart.dev/. [Accessed: 20-Sep-2019]

[60] Flutter Dev Team, ‘StatelessWidget class’, 2018. [Online]. Available: https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html. [Accessed: 01-Oct-2019]

[61] W. Sheikh, ‘Understanding Immutability’, *Medium*, 16-Jul-2018. [Online]. Available: https://medium.com/tribalscale/understanding-immutability-fdd627b66e58. [Accessed: 09-Jan-2020]

[62] R. Jin, ‘5 Benefits of Immutable Objects Worth Considering for Your Next Project’, 2017. [Online]. Available: https://hackernoon.com/5-benefits-of-immutable-objects-worth-considering-for-your-next-project-f98e7e85b6ac. [Accessed: 09-Jan-2020]

[63] J. Bloch, *Effective Java*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2008.

[64] Google LLC, *How to Create Stateless Widgets*, vol. Ep. 1. 2018 [Online]. Available: https://www.youtube.com/watch?v=wE7khGHVkYY. [Accessed: 23-Sep-2019]

[65] I. Krankka, ‘Putting build methods on a diet’, *iirokrankka.com*, 2018. [Online]. Available: https://iirokrankka.com/2018/06/18/putting-build-methods-on-a-diet/. [Accessed: 28-Aug-2019]

[66] Dart Team, ‘Performance best practices’, 2018. [Online]. Available: https://flutter.dev/docs/testing/best-practices. [Accessed: 11-Oct-2019]

[67] Flutter Dev Team, ‘StatefulWidget class’, 2018. [Online]. Available: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html. [Accessed: 01-Oct-2019]

[68] Google LLC, *How Stateful Widgets Are Used Best*, vol. Ep. 2. 2018 [Online]. Available: https://www.youtube.com/watch?v=AqCMFXEmf3w. [Accessed: 23-Sep-2019]

[69] Flutter Dev Team, 'InheritedWidget class', 2018. [Online]. Available: https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html. [Accessed: 01-Oct-2019]

[70] Flutter Dev Team, 'BuildContext class', 2018. [Online]. Available: https://api.flutter.dev/flutter/widgets/BuildContext-class.html. [Accessed: 01-Oct-2019]

[71] Flutter Dev Team, 'State Management Recommendations', 2019. [Online]. Available: https://flutter.dev/docs/development/data-and-backend/state-mgmt/options. [Accessed: 25-Nov-2019]

[72] Wikipedia, 'State management', *Wikipedia*. 04-Sep-2019 [Online]. Available: https://en.wikipedia.org/w/index.php?title=State_management&oldid=914027862. [Accessed: 15-Oct-2019]

[73] ISO, 'ISO/IEC/IEEE 42010:2011 Systems and software engineering — Architecture description'. ISO, 2011 [Online]. Available: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/05/50508.html. [Accessed: 15-Oct-2019]

[74] B. Egan and Contributors, 'Flutter Architecture Samples', *fluttersamples*, 2017. [Online]. Available: https://fluttersamples.com/. [Accessed: 28-Aug-2019]

[75] R. Rousselet and Flutter Dev Team, 'provider | Flutter Package', *Dart packages*, 2018. [Online]. Available: https://pub.dev/packages/provider. [Accessed: 06-Oct-2019]

[76] D. Boelens, 'Flutter - BLoC - ScopedModel - Redux - Comparison', *Didier Boelens*, 2019. [Online]. Available: https://www.didierboelens.com/2019/04/bloc---scopedmodel---redux---comparison/. [Accessed: 09-Sep-2019]

[77] V. Savjolovs, 'Flutter app architecture 101: Vanilla, Scoped Model, BLoC', *Medium*, 2019. [Online]. Available: https://medium.com/flutter-community/flutter-app-architecture-101-vanilla-scoped-model-bloc-7eff7b2baf7e. [Accessed: 28-Aug-2019]

[78] F. Hracek and M. Sullivan, *Pragmatic State Management Using Provider*, vol. 24. 2019 [Online]. Available: https://www.youtube.com/watch?v=HrBiNHEqSYU. [Accessed: 09-Sep-2019]

[79] M. Sullivan and F. Hracek, 'Pragmatic State Management in Flutter', Mountain View, CA, 09-May-2019 [Online]. Available: https://www.youtube.com/watch?v=d_m5csmrf7I. [Accessed: 28-Aug-2019]

[80] B. Egan, 'Flutter Redux Package', *Dart packages*, 2017. [Online]. Available: https://pub.dev/packages/flutter_redux. [Accessed: 06-Oct-2019]

[81] D. Abramov, 'Three Principles of Redux', 2015. [Online]. Available: https://redux.js.org/. [Accessed: 08-Oct-2019]

[82] D. Abramov, ‘Redux’, 2015. [Online]. Available: https://redux.js.org/. [Accessed: 06-Oct-2019]

[83] S. Suri, ‘Architect your Flutter project using BLOC pattern’, *Medium*, 2019. [Online]. Available: https://medium.com/flutterpub/architecting-your-flutter-project-bd04e144a8f1. [Accessed: 09-Sep-2019]

[84] D. Boelens, ‘Flutter - Reactive Programming - Streams - BLoC’, *Didier Boelens*, 2018. [Online]. Available: https://www.didierboelens.com/2018/08/reactive-programming---streams---bloc/. [Accessed: 12-Sep-2019]

[85] M. Sullivan and F. Hracek, *Technical Debt and Streams/BLoC*, vol. 4. 2018 [Online]. Available: https://www.youtube.com/watch?v=fahC3ky_zW0. [Accessed: 09-Sep-2019]

[86] A. Bizzotto, ‘Widget-Async-Bloc-Service: A Practical Architecture for Flutter Apps’, *Medium*, 2019. [Online]. Available: https://medium.com/coding-with-flutter/widget-async-bloc-service-a-practical-architecture-for-flutter-apps-250a28f9251b. [Accessed: 10-Oct-2019]

[87] Google LLC, ‘AngularDart’, 2018. [Online]. Available: https://angulardart.dev/. [Accessed: 07-Oct-2019]

[88] Dart Team, ‘Dart Streams’, 2019. [Online]. Available: https://dart.dev/tutorials/language/streams. [Accessed: 20-Sep-2019]

[89] F. Angelov, ‘Unit Testing with “Bloc”’, *Medium*, 2019. [Online]. Available: https://medium.com/flutter-community/unit-testing-with-bloc-b94de9655d86. [Accessed: 09-Oct-2019]

[90] E. W. Dijkstra, ‘The structure of the “THE”- multiprogramming system’. .

[91] J. Savolainen and V. Myllarniemi, ‘Layered architecture revisited — Comparison of research and practice’, presented at the 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, Cambridge, UK, 2009 [Online]. Available: https://ieeexplore.ieee.org/document/5290685. [Accessed: 06-Jan-2020]

[92] P. Walpita, ‘Software Architecture Patterns — Layered Architecture’, *Medium*, 10-Jul-2019. [Online]. Available: https://medium.com/@priyalwalpita/software-architecture-patterns-layered-architecture-a3b89b71a057. [Accessed: 06-Jan-2020]

[93] B. Martin, ‘The Principles of Clean Architecture’, King Street, Norwich, 2015 [Online]. Available: https://www.youtube.com/watch?v=o_TH-Y78tt4. [Accessed: 13-Sep-2019]

[94] D. Garlan and M. Shaw, ‘An Introduction to Software Architecture’, Carnegie Mellon University, Pittsburgh, PA, USA, 1994 [Online]. Available: https://dl.acm.org/citation.cfm?id=865128

[95] Dart Team, ‘Dart Enum’, 2017. [Online]. Available: https://www.tutorialspoint.com/dart_programming/dart_programming_enumeration.htm. [Accessed: 08-Jan-2020]

[96] Dart Team, ‘Dart Exception’, 2017. [Online]. Available: https://api.dart.dev/stable/2.7.0/dart-core/Exception-class.html. [Accessed: 08-Jan-2020]

[97] P. J. Ramadge and W. M. Wonham, ‘Supervisory Control of a class of discrete-event processes’, *SIAM J Control Opt*, vol. 25, pp. 206–230, 1987.

[98] Dart Team, ‘Asynchronous programming in Dart’, 2018. [Online]. Available: https://dart.dev/codelabs/async-await. [Accessed: 01-Oct-2019]

[99] C. Aldrich, ‘Flutter: Code Organization’, *Medium*, 04-May-2019. [Online]. Available: https://medium.com/flutter-community/flutter-code-organization-de3a4c219149. [Accessed: 13-Nov-2019]

[100] ISO, ‘ISO/IEC 19505-2:2012 Unified Modeling Language’, *ISO*, 2012. [Online]. Available: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/28/52854.html. [Accessed: 09-Jan-2020]

[101] F. Angelov, ‘equatable | Dart Package’, *Dart packages*, 2019. [Online]. Available: https://pub.dev/packages/equatable#-example-tab-. [Accessed: 10-Oct-2019]

[102] R. Hickey, ‘The Value of Values’, 2012 [Online]. Available: https://www.youtube.com/watch?v=-6BsiVyC1kM. [Accessed: 10-Jan-2020]

[103] F. Angelov and Contributors, ‘Flutter Infinite List BLoC Tutorial’, 2019. [Online]. Available: https://felangel.github.io/bloc/#/flutterinfinitelisttutorial. [Accessed: 13-Sep-2019]

[104] B. Martin, ‘The Dependency Inversion Principle’, in *The C++ Report*, 1996, pp. 8(6):61–66 [Online]. Available: https://www.labri.fr/perso/clement/enseignements/ao/DIP.pdf

[105] M. Fowler, ‘Inversion of Control Containers and the Dependency Injection pattern’, *martinfowler.com*, 2004. [Online]. Available: https://martinfowler.com/articles/injection.html. [Accessed: 11-Jan-2020]

[106] H. Y. Yang, E. Tempero, and H. Melton, ‘An Empirical Study into Use of Dependency Injection in Java’, in *19th Australian Conference on Software Engineering (aswec 2008)*, 2008, pp. 239–247, doi: 10.1109/ASWEC.2008.4483212.

[107] Z. Rehman, ‘Dependency Injection In Flutter’, *Medium*, 19-Aug-2019. [Online]. Available: https://medium.com/flutter-community/dependency-injection-in-flutter-f19fb66a0740. [Accessed: 11-Jan-2020]

[108] S. Suri, ‘Compile time Dependency Injection in Flutter’, *Medium*, 22-Jul-2019. [Online]. Available: https://blog.usejournal.com/compile-time-dependency-injection-in-flutter-95bb190b4a71. [Accessed: 11-Jan-2020]

[109] Google LLC, *Inject Repo*. Google, 2019 [Online]. Available: https://github.com/google/inject.dart. [Accessed: 13-Oct-2019]

[110] A. Agarwal, ‘Structuring a React Project - a Definitive Guide’, *Medium*, 29-Mar-2019. [Online]. Available: https://blog.bitsrc.io/structuring-a-react-project-a-definitive-guide-ac9a754df5eb. [Accessed: 14-Jan-2020]

[111] Tutorialspoint, ‘Software Design Basics’, 2018. [Online]. Available: https://www.tutorialspoint.com/software_engineering/software_design_basics.htm. [Accessed: 15-Jan-2020]

[112] D. Wickramaarachchi and R. Lai, ‘Software modularization in global software development’, in *2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–6, doi: 10.1109/ICODSE.2014.7062705.

[113] Dart Team, ‘Creating packages’, 2018. [Online]. Available: https://dart.dev/guides/libraries/create-library-packages. [Accessed: 15-Jan-2020]

[114] MDN Contributors, ‘Client-side form validation’, *MDN Web Docs*, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation. [Accessed: 20-Jan-2020]

[115] Flutter Dev Team, ‘Build a form with validation’, 2018. [Online]. Available: https://flutter.dev/docs/cookbook/forms/validation. [Accessed: 21-Jan-2020]

[116] App Annie, ‘App Annie acquired Distimo in 2014’, 2014. [Online]. Available: https://www.appannie.com/de/about/. [Accessed: 26-Jan-2020]

[117] R. Kim, ‘Distimo: App translation can pay off, especially in Asia’, 01-Oct-2012. [Online]. Available: https://gigaom.com/2012/10/01/distimo-app-translation-can-pay-off-especially-in-asia/. [Accessed: 17-Jan-2020]

[118] J. Nouch, ‘Distimo: 90% of the top apps delivered in English, but localisation increasingly’, 2012. [Online]. Available: https://www.pocketgamer.biz/data-and-research/45474/distimo-90-of-the-top-apps-delivered-in-english-but-localisation-increasingly-important/. [Accessed: 26-Jan-2020]

[119] A. Wong, ‘Distimo: Localization Increases Downloads by 128x for iOS Apps’, 2012. [Online]. Available: http://www.oneskyapp.com/blog/localization-increases-downloads-by-128-on-average-for-iphone-apps/. [Accessed: 26-Jan-2020]

[120] Flutter Dev Team, ‘Internation-alizing Flutter apps’, 2018. [Online]. Available: https://flutter.dev/docs/development/accessibility-and-localization/internationalization. [Accessed: 17-Jan-2020]

[121] Flutter Dev Team, ‘Locale class’, 2018. [Online]. Available: https://api.flutter.dev/flutter/dart-ui/Locale-class.html. [Accessed: 17-Jan-2020]

[122] Flutter Dev Team, ‘MaterialApp class’, 2018. [Online]. Available: https://api.flutter.dev/flutter/material/MaterialApp-class.html. [Accessed: 17-Jan-2020]

[123] IEEE, ‘IEEE Xplore Digital Library’, 1963. [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp. [Accessed: 20-Sep-2018]

[124] ACM, ‘ACM Digital Library’, 1947. [Online]. Available: https://dl.acm.org/. [Accessed: 20-Sep-2018]

[125] T. Dingsøyr, T. Fægri, and J. Itkonen, ‘What is large in large-scale? A taxonomy of scale for agile software development’, 2014, vol. 8892, doi: 10.1007/978-3-319-13835-0_20.

[126] O. Levy and D. G. Feitelson, ‘Understanding Large-Scale Software – A Hierarchical View’, in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, Montreal, QC, Canada, 2019, pp. 283–293, doi: 10.1109/ICPC.2019.00047 [Online]. Available: https://ieeexplore.ieee.org/document/8813291/. [Accessed: 28-Jan-2020]

[127] M. Fowler and J. Lewis, ‘Microservices: A definition of this new architectural term’, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 27-Jan-2020]

Appendix

A.1 Transcript of the Interview with Felix Angelov

On the 9th of December 2019, a one-hour long interview was held with Felix Angelov under the topic of *"The BLoC Pattern and Flutter in large-scale applications"*. This is a summary of the contents of that interview. Timestamps are given for crucial parts of the interview.

- **[Before the recording started]**
  - Introduction and asking for Angelov's consent to be recorded.
- **[00:00]** Preamble.
- **[00:44]** Q: *"Exactly for what kind of application are you using the BLoC-Pattern?"*
  - His team is reimplementing consumer-facing application at BMW.
  - The app is responsible for interaction with the car and remotely controlling the car.
  - He says that during the development process problems arose while maintaining separate codebases (IOS/Android).
  - The Flutter app will be replacing the current application soon (early 2020).
- **[02:57]** Q: *"Why did you choose BLoC as a state management solution over other options?"*
  - His team started with Redux & the BLoC Pattern.
    - His team was originally of the opinion that redux is superior for global state.
    - His team used BLoC only for local state.
    - His team thought Redux has too much boilerplate.
    - His team thinks the learning curve was high for Redux.
  - His team eventually switched to exclusively use the BLoC pattern.
    - He says that this is when they built their own BLoC package.
    - His team wanted a high-level abstraction for state management.
    - His team did not want their developers to do low-level stream handling.
    - His team thinks that their state-management solution should be easy to understand for new developers.
    - His team liked that each BLoC was handling one use-case.
    - His team did not want one global store handling the complete app-state.
    - His team is in the process of completely modularizing their app (micro-service approach).

- **[09:20]** Q: *"Regarding the BLoC-Package, why did you choose to limit the number of input- and output-streams of BLoCs to one?"*
    - He says that this was done to force single responsibility for a given BLoC.
    - He says that this way it becomes easy to see when a given BLoC is doing too much.
    - He recommends that developers should avoid an *"application BLoC"*.
    - He recommends that a given BLoC should only handle one aspect of a given application.
- **[13:24]** Q: *"Rule 1. for BLoCs (as defined by Paolo Soares) is 'Each complex enough component has a corresponding BLoC' - How do you define 'complex enough' in your project(s)?"*
    - He says that it usually ends up being one BLoC per feature.
    - [14:35] His team loosely defines one feature as: *"one piece of value given to the customer"*.
    - In his project, the standard BLoC usually has four states: finished, loading, failure, success.
    - [16:31] The approach of his team is to start with one BLoC and if they are starting to have trouble with that BLoC (*"it becomes awkward"),* they refactor and split it into two BLoCs.
    - [17:00] He recommends drawing finite state machine diagrams before implementing a given BLoC.
    - [18:18] *"Prefer smaller more focused BLoCs instead of larger BLoCs"* this way testability and reusability are improved.
    - He also states that there is no perfect rule for this.
- **[18:50]** Q: *"What are your thoughts on naming conventions for BLoCs, Events, States etc.?"*
    - His team started off not having any.
    - His team then started to run into problems because of this.
    - His team is now using one at BMW.
    - [20:10] He explains their conventions.
    - [23:23] He says that enums as states are nice when states are very simple.
- **[24:21]** Q: *"What is your opinion on having multiple widget classes in one file?"*
    - He thinks it is okay as long as there is only one public widget.
    - This forces you to separate responsibilities.

- **[25:25]** Q: *"What does your typical Flutter project file structure look like?"*
  - His team uses a tree-like approach.
    - He says that the folder structure mimics the widget tree of the application (almost).
    - His team uses one folder for a given feature. These folders are then nested in one another.
    - His team uses this approach because it enables developers that know how the app is structured to easily find the code for a specific part of the app.
    - He says that this way features are already modularized.
    - He says that the downside of the approach is a lot of nesting.
  - His team does not use the layered file structure approach, because with that approach responsibilities are split over multiple locations and modularizing is more difficult.
  - He thinks that any approach is fine as long as it is used consistently.
- **[29:55]** Q: *"Didier Boelens states in one of his articles on the BLoC Pattern, that restricting BLoC to only use streams as inputs and outputs can be a little 'heavy'. What are your thoughts on this statement?"*
  - He does agree when talking about implementing BLoC from scratch.
  - He says that BLoCs add complexity to the program.
  - He says that the BLoC package is meant to counteract this.
- **[33:23]** Q: *"Do you have any tips for people using Flutter in a large-scale application?"*
  - He says that it is important to understand the scope of the application.
  - He says that the team should take the time to plan the structure of the project.
  - He says that many people say you should start with *"set state"* and then refactor once you run into problems with that way of managing state;
    - He disagrees with this.
    - He says that refactoring to a new state-management solution is a lot more work later in the project.
    - He says that teams then end up combining BLoC & *"set state"*.
    - He thinks this is not a good approach.
  - He adds that teams should be open to addressing problems soon.
  - [36:36] He states that the best projects are the ones where the people constantly question the decisions and are not afraid to refactor.
  - He says that the most important thing for large-scale applications is modularization.
    - He says it is not as difficult as people think.
    - The BMW app is modularized in such a way that each feature can theoretically run as its own separate app (similar to the popular microservice approach).

- **[41:29]** Q: *“Do you have any tips for people using the BLoC-Pattern in a large-scale application?”*
    - He says that it is important to understand the concept and its advantages.
    - He says to not only use BLoC because everyone else is doing it.
    - He recommends using tooling made by other developers (BLoC generation, linting).
    - He says it is important to write tests.
- **[43:08]** Why they chose Flutter (not a question, the dialogue naturally came to this topic).
    - His team ended up choosing Flutter because of the amount of tooling Flutter offers for free (3 levels of testing, widget tree visualization, hot reload).
- **[46:30]** Round up (This is when Angelov could ask questions of his own).
- **[01:06:14]** End of the interview.

A.2 Milestones of the Thesis

Month	November (11)				December (12)			
Week Relative	1	2	3	4	5	6	7	8
Week Number	45	46	47	48	49	50	51	52
Thesis	-4	-3	-2	-1	1	2	3	4
Milestones	**Hand-in expose,** May Thai Star Domain analysis	**1. Meeting with Prof, Meeting with MTS architect,** MTS: Architecture Planning, MTS: Front-end 1/2	**Meeting with Capgemini,** MTS: Front-end 2/2	**Thesis Outline, Hand-in Requirements,** MTS: BLoCs 1/2, MTS: Connect to back-end 1/2	**2. Meeting with Prof, Register Thesis 30.11,** MTS: Connect to back-end 2/2, MTS: BLoCs 2/2	**Angelov Interview,** MTS: *Refactoring*	MTS: *Refactoring* , MTS: In-code docu	Plan Writing process, Docu: Intro, Docu: Ref Project, Docu: Flutter
Other Events			Guide Publication			MI Showcase		Christmas
Capgemini	Working Student							

Month	January (01)					February (02)			
Week Relative	9	10	11	12	13	14	15	16	17
Week Number	1	2	3	4	5	6	7	8	9
Thesis	5	6	7	8	9			Colloquium	
Milestones	Docu: State-management, Docu: Architecture 1/4	**3. Meeting with Prof,** Docu: Architecture 3/4, Docu: Models, Docu: Object Equality, Docu: Immutability, Docu: DI	Docu: File Structure, Docu: Modularization, Docu: Locale 4/5	Docu: Locale 5/5, Docu: Forms, Docu: *Iterate*	Docu: Conclusion, Docu: Abstract, Docu: *Iterate* , Feedback from peers, Spellchecking, Grammar, Print out 3x, **Hand-In 01.02**			**Colloquium 17.02**	
Other Events	New Years			Guide Presentation					
Capgemini	Working Student								

A.3 Project Plan of Guide and Thesis

Month	August (08)				September (09)				October (10)					November (11)			
Week Relative	1	2	3	4	5	6	7	8	9	10	11	12	13	1	2	3	4
Week Number	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Guide	-2	-1	1	2	3	4	5	6	7	8	9	10	11			Publish	
Thesis														-4	-3	-2	-1
Capgemini					Working Student									Working Student			
Semester								WS 19/20									

Month	December (12)				January (01)					February (02)			
Week Relative	5	6	7	8	9	10	11	12	13	14	15	16	17
Week Number	49	50	51	52	1	2	3	4	5	6	7	8	9
Guide								Present					
Thesis	1	2	3	4	5	6	7	8	9			Colloqui.	
Capgemini	Working Student												
Semester	WS 19/20												29.02

Guide	7 - 11 Weeks
Thesis	9 Weeks
WS 19/20	23.09 - 29.02