FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Coffee Time Mobile Application in Flutter
Student:	Bc. Petr Nymsa
Supervisor:	Ing. David Šenkýř
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

By the end of 2018, the first stable version 1.0 of Flutter was released. The current stable version is 1.12.13 and the technology is still developing. The goal of this thesis is to design, implement, and test a multiplatform mobile application developed by this new technology. For this purpose, an example mobile application focused on the search for cafes will be implemented.

Steps to follow:
1. Current Flutter provides different state management approaches. Compare 2 approaches and select the one that you prefer.
2. Based on the previous decision, design the architecture of the application.
3. Implement the application that provides the following functionalities: search for a nearby café, details of the café, add/remove tags of the café, e.g. pet-friendly, child-friendly, etc.
4. Provide user testing.
5. Summarize and evaluate the results you have achieved.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 16, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Coffee Time Mobile Application in Flutter

Bc. Petr Nymsa

Department of Software Engineering
Supervisor: Ing. David Šenkýř

May 27, 2020

Acknowledgements

I would like to thank my supervisor *Ing. David Šenkýř* for his guidance and valuable advices. Sincere gratitude belongs to my partner and family for their support during work on this thesis.

Declaration

In Prague on May 27, 2020 . . . .

Citation of this thesis

Nymsa, Petr. *Coffee Time Mobile Application in Flutter.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Práce se zabývá multiplatformním frameworkem Flutter, a to nejen pro tvorbu mobilních aplikací. V práci je popsán samotný framework, jeho použitelnost a využití při vývojí aplikací. Následně je famework použit při návrhu a implementaci aplikace Coffee Time pro operační systém Android. Aplikace vyhledává kavárny v blízkém okolí s možností filtrování dle různých kritérií. Uživatelé aplikace si mohou zobrazit detailní informace, spustit navigaci nebo přečíst hodnocení. Aplikace byla navrhnnuta pomocí prototypu uživatelského rozhraní a jeho postupném otestování. Nakonec byla aplikace zpřístupněna a nasazena ke stažení pro mobilní telefony se systémem Android.

Klíčová slova Flutter framework, reaktivní programování, Android aplikace, vyhledávač kaváren, serverless, Firebase.

Abstract

This thesis is focused on multi-platform framework Flutter for creating not only mobile applications. In the thesis, Flutter framework and its usability during application development are described. Flutter is used to design and implement the Coffee Time application for Android devices. This application is able to search nearby cafes with options to filter them by different criteria. Application users can display cafe details, launch navigation or read reviews. The application was designed using a prototype user interface and its gradual testing before the actual implementation. In the end, the application was released for download to Android devices.

Keywords Flutter framework, reactive programming, Android application, cafe search, serverless, Firebase.

Contents

Introduction **1**
Motivation . . . 1
Structure . . . 2

1 Flutter Foundations **3**
1.1 Technical overview . . . 4
1.1.1 Reactive Programming . . . 6
1.2 Everything Is a Widget . . . 7
1.2.1 Widgets Are Not Only Visible Parts . . . 8
1.2.2 Stateless vs. Stateful Widget . . . 9
1.3 State Management Approaches . . . 14
1.3.1 Case Study Note . . . 14
1.3.2 Inherited Widget . . . 15
1.3.3 Provider Package . . . 17
1.3.4 Business Logic Component . . . 19
1.3.5 Conclusion . . . 23
1.4 Flutter Internals . . . 23
1.4.1 RenderObject and RenderTree . . . 23
1.4.2 Everything Is a Widget Revisited . . . 24
1.4.3 Deciding What to Redraw . . . 25
1.4.4 Notion of Keys . . . 26
1.4.5 Const Optimisation . . . 28
1.5 Conclusion . . . 29

2 Coffee Time Analysis **31**
2.1 Considered Application . . . 31
2.2 Use Cases . . . 32
2.2.1 UC1: Display Nearby Cafes As a List . . . 33
2.2.2 UC2: Display Nearby Cafes As a Map . . . 33

2.2.3 UC3: Start Navigation . . . 34
2.2.4 UC4: Toggle Cafe As a Favourite . . . 34
2.2.5 UC5: Setting the Filter . . . 34
2.2.6 UC6: Display Favourite Cafes . . . 35
2.2.7 UC7: Review the Cafe's Tags . . . 35
2.2.8 UC8: Suggest a New Tag . . . 35
2.3 Existing Alternatives . . . 36
2.3.1 Gastromapa Lukáše Hejlíka . . . 36
2.3.2 Pivní Deníček . . . 37
2.3.3 Restu . . . 39
2.3.4 Zomato . . . 39
2.3.5 Google Maps . . . 40
2.4 Application Prototype . . . 41
2.4.1 Coffee Time Prototype . . . 42
2.4.2 Nielsen Heuristic . . . 45
2.4.3 User Testing . . . 48
2.5 Back-end Analysis . . . 51
2.5.1 Google Places API . . . 52
2.5.2 Coffee Time API . . . 55
2.5.3 The Selection of the Right Technology . . . 56
2.6 The Conclusion . . . 58

3 Implementation 59
3.1 Clean Architecture . . . 59
3.2 Coffee Time API . . . 61
3.2.1 The API Endpoints . . . 61
3.2.2 Express.js Pipeline . . . 62
3.2.3 Routing . . . 64
3.2.4 Integration with Firestore . . . 65
3.2.5 Functions Deployment . . . 65
3.3 Coffee Time . . . 66
3.3.1 Domain Layer . . . 66
3.3.2 Data Layer . . . 70
3.3.3 Representation Layer . . . 73
3.4 SOLID and Dependency Injection . . . 76
3.5 Unit Testing . . . 78
3.6 Conclusion . . . 79

4 Application Release 81
4.1 Development Workflow . . . 81
4.1.1 Continuous Integration . . . 83
4.2 Internal Testing . . . 83
4.3 Crashlytics . . . 84
4.4 User Testing Re-Evaluation . . . 84

4.5 Release 85
4.5.1 Missing Feature from Prototype 87
4.5.2 Conclusion 87

Conclusion 89
Next Steps 89
Personal Author's Note in the End 90

Bibliography 91

A Acronyms 97

B Content of Enclosed CD 99

List of Figures

0.1 Flutter Trend Against Other Popular Frameworks [4]. 2

1.1 Widget Composition Example. 3
1.2 Dart Platforms [11]. 5
1.3 Flutter System Overview [8]. 5
1.4 Compose Widgets to Create Layout [16]. 8
1.5 Compose Widgets to Create Layout – Left Column [16]. 8
1.6 User Interface Formula [17]. 9
1.7 Counter Application. 11
1.8 Counter Application's Widget Tree. 11
1.9 Expected Rebuilt vs. Actual Rebuilt. 13
1.10 InheritedWidget Approach and Its Widget Tree. 17
1.11 Provider Approach and Its Widget Tree. 20
1.12 A BLoC Pattern [18]. 20
1.13 BLoC Approach and Its Widget Tree. 21
1.14 Three Trees – Widget, Element and Render Tree [25]. 24
1.15 Square Widgets and Associated Elements with States. 26
1.16 Square Widgets After Swap and Associated Elements With Wrong States. 26

2.1 Application Use Case Diagram. 32
2.2 Gastromapa *L. Hejlíka* [26]. 37
2.3 Pivní deníček [27]. 38
2.4 Restu [28]. 38
2.5 Zomato [29]. 40
2.6 Google Maps [30]. 40
2.7 Coffee Time Task Graph. 43
2.8 Low Fidelity (Lo-Fi) Prototype. Cafe List (Left) and Detail Screen (Right). 44

2.9 High Fidelity (Hi-Fi) Prototype. Cafe List (left) and Detail Screen (right). 45
2.10 Client and API Communication Flow. 55
2.11 Cafe’s Tag and Cafe Relation. 56
2.12 Coffee Time Domain Model. 57

3.1 Clean Architecture Defined By *B.C. Martin* [47]. 60
3.2 API Components. 66
3.3 Cafe Repository Interface. 68
3.4 Tag Repository Interface. 69
3.5 Repositories and Services Association. 71
3.6 Presentation Layer Screens Organization. 73
3.7 Screens and Blocs Association. 78

4.1 Released Version – Cafe List and Detail Screens. 86
4.2 Coffee Time QR Code for Downloading the Application. 86

List of Tables

2.1 Analysed Applications User Interface Summarization. 41
2.2 Find Place Parameters. 52
2.3 Find Place Fields. 53
2.4 Nearby Search Parameters. 53
2.5 Place Details Parameters. 53
2.6 Place Details Fields. 53

3.1 Places Endpoints. 62
3.2 Nearby Parameters. 62
3.3 Find Parameters. 62
3.4 Tags Endpoints. 63

Introduction

The mobile applications usage is growing. The native development of an application for each, individual platform is well used across many companies. However, many of them, predominantly smaller ones, concludes that developing application for each platform and maintenance is not cheap and comes with much of work. On the other hand, there are technologies which offer cross-platform development with one code-base. Every cross-platform technology takes its own approach to how the code is compiled to the native platform.

Some of them use the concept of bridging the cross-platform user interface primitives to its counterpart in the native platform. Well-known examples are React Native [1] or Xamarin Forms [2]. The opposite approach is the form of a progressive web application (PWA) where the application is written as a web-based application with support of native features. This approach uses, for example, Ionic framework [3].

During 2017, the concept of another approach was proposed, where the application uses low-level platform API to draw over the whole screen with keeping high performance and access to native features. Later on, from this concept open-source framework Flutter, made by Google, was created [5].

Flutter for the last three years until now (first half of 2020) started to gain developers attention, and it was highly promoted by Google. One indication of its growing popularity is *Stack Overflow Developer Survey 2019* [6] where it took third place of "Most Loved" framework directly after *.NET Core* and *Torch/PyTorch* and highly growing trend among questions created during the last years (Figure 0.1). However, like with every new technology, the Flutter can become well-known and well used or will be left as a dead-end.

Motivation

There are many reasons why the author chose this topic for the thesis. First of all, his bachelor thesis [7] already focused on mobile application development, although it used different cross-platform technology – Xamarin. During his

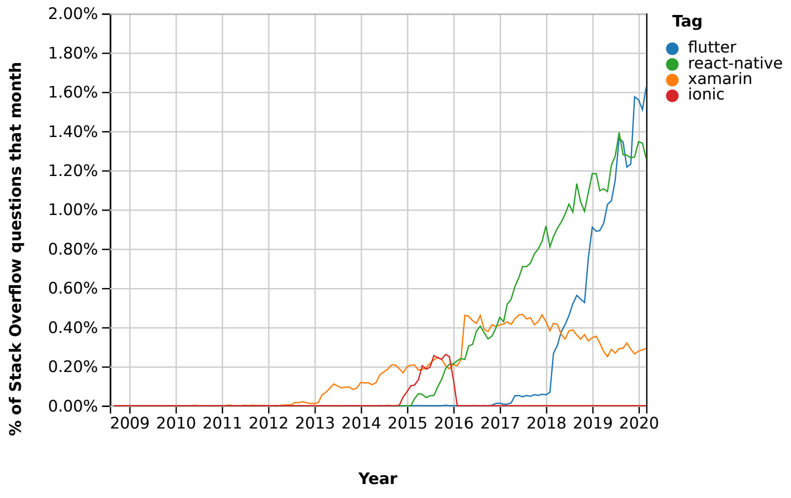


Figure 0.1: Flutter Trend Against Other Popular Frameworks [4].

ongoing studies, the author discovered and started to use new, by his opinion, promising framework Flutter. So his first goal and motivation was to study Flutter more in-depth and bring comprehensive study material of this framework for others. The second reason was to conclude if Flutter can become a framework which can be used to create production-ready applications or if it is still an experimental framework. Last reason was motivation to create complex, and yet, simple to use mobile application for everyone who seeks to find new cafes to visit.

Structure

The thesis is divided to four chapters:

- Chapter 1 deals with introduction to Flutter framework, its concept and internal functionality.
- Chapter 2 introduces proposed Coffee Time application. It describes the created prototype and its user testing. The analysis of back-end services is also described.
- Chapter 3 describes a process of back-end implementation as well as of Coffee Time application implementation. In the chapter, details how its implemented, which approaches was taken and how development process was done.
- Chapter 4 describes final application release and its testing.

In conclusion, the results are compared with the goals of this thesis.

Chapter 1

Flutter Foundations

In the introduction, a new, promising, cross-platform framework was introduced. The Flutter's primary goal is to provide the ability to build high-performance, high-fidelity apps for *iOS*, *Android*, web, and desktop systems (Windows, MacOS) from a single code-base [8]. In this chapter, the framework philosophy will be described. Used programming language and theory of reactive programming is briefly introduced. The chapter describes the concept of widgets as a base building block for every application. Later on, one of the most critical topics – *state management* is discussed in the form of existing approaches and recommendation which to prefer when building applications. At the end of this chapter, the brief look under the framework's hood is discussed.

Flutter includes a modern react-style framework, a 2D rendering engine, predefined widgets and development tools. The primary premise is a motto "everything is a widget". A widget is an immutable building block of application which is part of the user interface. Each widget can define structural elements such as a button, stylistic elements such as colour or it can define the interface's layout, such as padding. Widgets are composed as a tree hierarchy with a possibility of composing each widget to another. If any event occurs (such as user interaction), the framework can rebuild part of this tree to redraw the screen.

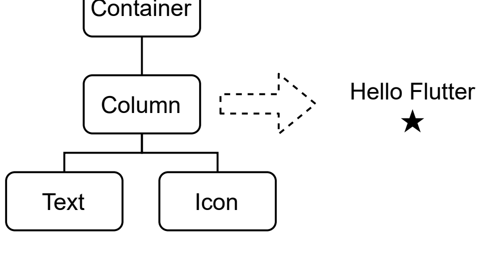


Figure 1.1: Widget Composition Example.

Flutter encourages developers to create and use small, single-purpose widgets and compose them to create complex interfaces and layouts. Take an example from Listing 1, where the root widget, *Container*, is used to create a rectangular visual element. The *Container* is something like a *div* element in HTML. Under the *Container* there is *Column* widget which composes children widgets into the vertical direction. Finally, *Text* widget displaying text "Hello Flutter" and *Icon* widget showing star icon. The composition hierarchy along with a result is shown in Figure 1.1.

```dart
Container(
    padding: const EdgeInsets.all(5.0),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Hello Flutter'),
        Icon(Icons.star),
      ]
    ),
)
```

Listing 1: Widget Composition Code Example.

1.1 Technical overview

Flutter uses programming language Dart (specification v2.0 [9]). It is also made by Google and it is inspired by languages such as JavaScript. Dart using statically typed system with runtime checks, but like many other languages highly use type inference [10]. Dart can be used from writing simple scripts to full-featured applications. Dart has flexible compiler technology where the compiler can decide running code in different ways, depending on the targeted platform [11].

- **Dart Native** – For programs targeting devices (mobile, desktop, server, and more), Dart Native includes both a Dart VM with Just-in-Time (JIT) compilation and an Ahead-of-Time (AOT) compiler for producing machine code.
- **Dart Web** – For programs targeting the web, Dart Web includes both a development time compiler (dartdevc) and a production time compiler (dart2js).

Flutter performs the use of both ways. If the targeted platform is web, the Dart Web is used. For other platforms Dart Native is chosen. The Dart Native's JIT compilation is highly used to support fast development process with

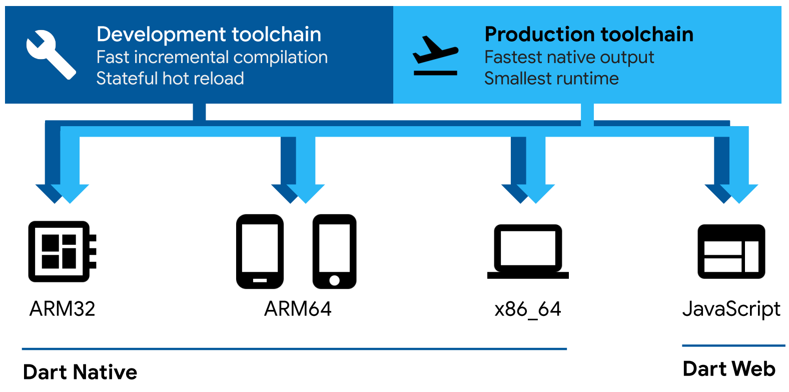


Figure 1.2: Dart Platforms [11].

"hot-reload" functionality. Then the AOT compilation is used for the best-optimised production-ready result on the native platform.

Flutter framework is organised into several layers (Figure 1.3), where each layer makes usage of the previous one. The upper layers are more frequently used by developers on a daily basis, and lower layers are used only if the developers need to create particular customizations.

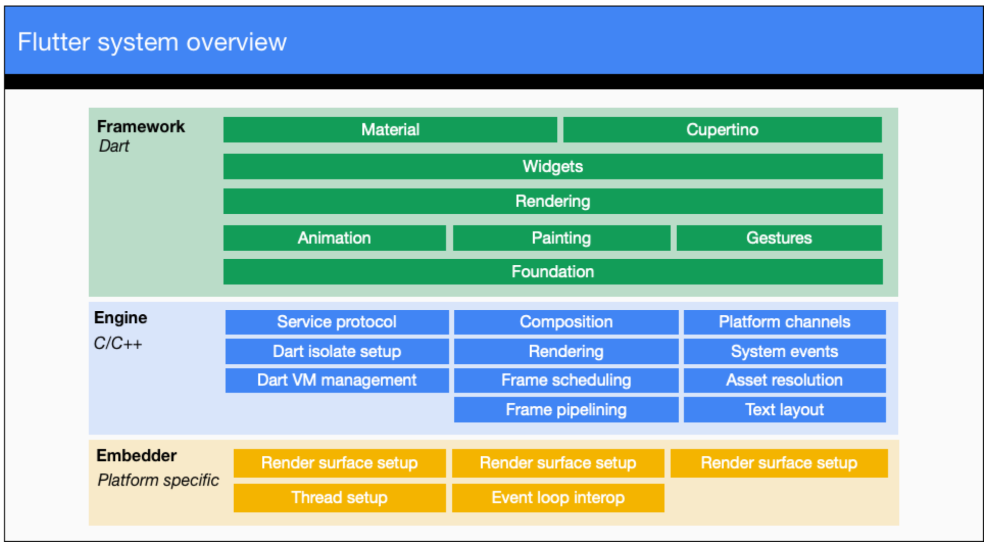


Figure 1.3: Flutter System Overview [8].

Unlike the other frameworks, Flutter uses high-performance 2D rendering engine and draws everything onto the screen directly. That means pixel-perfect control over what and how it is displayed. The most top layers, *Mate-*

rial and *Cupertino*, are set of widgets which defines Material Design (Android systems) and Apple Design components respectively. To highlight that, Flutter does not use native components, but everything draws by itself. These two layers support developers to bring the standardised look and feel to the targeted platform.

1.1.1 Reactive Programming

Flutter makes significant usage from the concept of reactive programming. There is nearly always a requirement to update data in response to user interaction or any other event such as getting data from the server. More than that, sometimes it is necessary to update different parts of the user interface in response to these events.

Flutter creates user interface by composing *immutable* widgets. The immutability is the key point here. Whenever user interface needs to "redraw" screen, the part of the widget tree is replaced by *new* widget instances (in fact, it is not simple as that, and this topic is more deeply discussed later in this chapter). In many other User Interface (UI) frameworks, such as *Xamarin*, is usually taken the approach of coupling UI components with view-models through concepts such as data binding [12]. That means that whenever UI needs to change, the components mutate application's state. Flutter takes an entirely different approach. It can be said "here is the current state of the application, draw something on the screen accordingly" – there is no way to state `widget.property = new value` as widgets are immutable.

1.1.1.1 The Notion of Streams

A Stream can be described as "A pipe with two ends, only one allowing to insert something into it. When something is inserted into the pipe, it flows inside the pipe and goes out by the other end" [13]. The Stream can convey any data type, from simple values to events, complex object or even another stream. The data can come to the Stream, for example, from an external data source such as server connection or from events such as user interactions. In Dart, the Streams support manipulating them, filtering, re-grouping, modify data before they are send and much more. This functionality can be used to build reactive UI. Flutter has several widgets supporting streams to rebuild part of the UI whenever new data arrived into the Stream.

The answer to the question "What is reactive programming?" could be "Reactive programming is programming with asynchronous data streams“ [13][14]. Within Flutter framework, anything from an interaction event (a tap, a gesture), changes of a variable, messages, everything that may change is conveyed and triggered by streams.

It means that with reactive programming, according to [13], the application:

- becomes asynchronous,
- is architectured around the notion of Streams and their listeners,
- when something happens somewhere (an event, a change of a variable) a notification is sent to a Stream,
- if "somebody" listens to that Stream, it will be notified and will take appropriate action(s), whatever its location in the application.

From Widgets perspective – Widget does not longer need to know:

- what is going to happen next,
- who might use this information (no one, one or several Widgets),
- where this information might be used (nowhere, same screen, another one, several ones),
- when this information might be used (almost directly, after several seconds, never).

Later on in this chapter, the Business Logic Component (BLoC) pattern is introduced. This pattern uses Streams to manage application life-cycle and they are used for state management.

1.2 Everything Is a Widget

In this section, we will discuss in more detail how the UI is built. Every UI consists of the layout and individual components. The layout defines the screen's base structure, such as a menu on the top and subsequent actions on the bottom. Then the layout is composed of individual components, such as a menu, buttons or icons. Together they create a final interface.

These building blocks in Flutter are called "Widgets". Whatever it is simple text, a button, or complex parts of the layout, such as a grid with multiple columns and rows – everything is a widget. Widgets describe what their view should look like given their current configuration and state. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next one [15]. As the composition to the tree implies, each widget has at most one parent and zero or more children widgets. This tree, called "widget tree", is in fact, one of the three trees involved. The framework has a sophisticated way of decision about how the trees should be rebuilt and the screen updated. This behaviour is in more detail described later in this chapter.

The Flutter framework uses only one language to define both the user interface and business logic as well. Widget is a Dart class which inherits from some of the widget's base class (typically *StatelessWidget* or *StatefulWidget*). Each widget has a `build()` method which defines how the widget should be built (and drawn on the screen).

1.2.1 Widgets Are Not Only Visible Parts

Widgets are not only visible parts of the UI such as clickable buttons, text or icons. The widgets also define layouts such as columns, rows, grids, the margin between other widgets, padding around them and more.

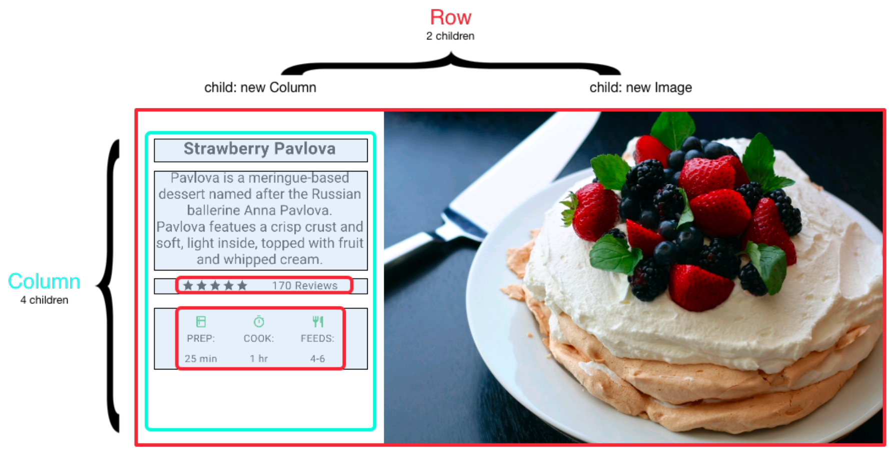

Row
2 children
child: new Column
child: new Image
Column
4 children
Strawberry Pavlova
Pavlova is a meringue-based dessert named after the Russian ballerine Anna Pavlova. Pavlova featues a crisp crust and soft, light inside, topped with fruit and whipped cream.
★★★★★ 170 Reviews
PREP: 25 min
COOK: 1 hr
FEEDS: 4-6

Figure 1.4: Compose Widgets to Create Layout [16].

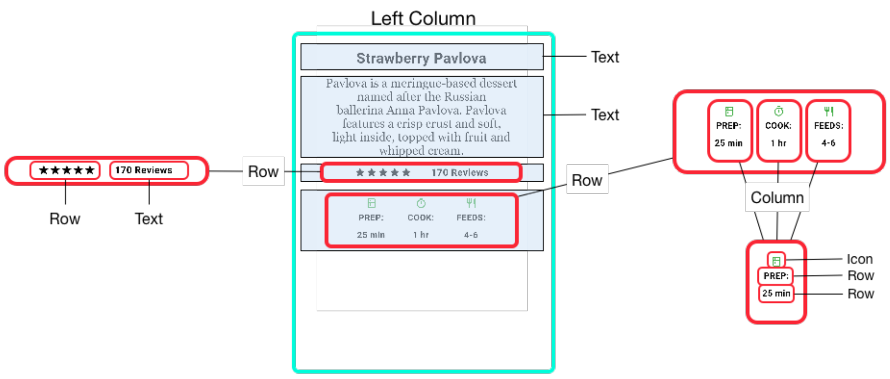


Figure 1.5: Compose Widgets to Create Layout – Left Column [16].

An example of widget composition creating a layout is shown in Figure 1.4. The root widget, a *Row* widget, contains two nested widgets. On the left there is a *Column* which contains more nested widgets and on the right, *Image* widget which displays a product image. The break-down of the left column widget can be seen in Figure 1.5.

1.2.2 Stateless vs. Stateful Widget

In the introduction it was said that Flutter's approach of displaying current user interface is declarative – "here is the current state of the application, draw something on the screen accordingly". In Flutter, whenever application's state changes, the user interface is redrawn. There is no imperative way to change the UI, such as `textWidget.text = 'new text'`. The advantage of the declarative approach is that there is only one code path for any state of the UI. Developers just describe how the screen should look for a given state, and that is it [17]. The UI can be described as a formula where UI is equal to function which takes a state and returns new UI (Figure 1.6).

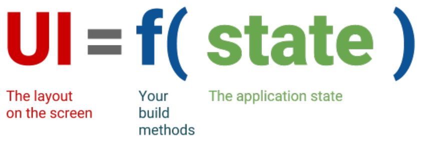


Figure 1.6: User Interface Formula [17].

1.2.2.1 Build Context

An essential part of the widgets is *BuildContext*. A context is a reference to the location of the widget within the part of the tree [18]. Each widget has its own context. As widgets are composed to the tree, the contexts are as well. The widget has access to its own context and its parent context.

The *BuildContext* is provided to each widget through the `build()` method and is used to find the widget's ancestors. This is commonly used to obtain a defined application theme or get a reference to a navigator widget, which is used to do navigation between screens.

1.2.2.2 Local vs. Application State

The state is anything that forms what should be displayed. The state is any data what are needed in order to rebuild UI at the moment [19]. The state can be separated into two concepts – local state and application state.

- **Local state** – Local state is which can be tied into one widget. It can be, for example, current tab in the "tab selector" widget, current progress of animation or state of checkbox (checked or unchecked).

- **Application state** – Application State is which can not be local, whenever some information is needed to share across multiple widgets, the state which should be kept during a user session. An example of application state can be a logged user information, loaded articles from the server or chat messages.

1.2.2.3 Stateless Widget

A *Stateless Widget* is a widget which does not manage its own state. Once it gets its parameters, and it was built through the `build()` method, it cannot be changed. Remember, that whenever Flutter decides to redraw the screen, part of the tree is rebuilt, but with new instances of the widgets. Typical examples of the *Stateless Widget* can be *Container*, *Text* or *Icon.* These widgets accept many parameters which can alter their look (and behaviour), but they cannot be changed later on by themselves.

1.2.2.4 Stateful Widget

Whenever widget needs to manage its state and wants to mutate it for example, in case of an event, the widget should be stateful. The widget as a Stateless accepts parameters which can be used to configure this widget but also has an associated object, called state. This state object is an active part of the widget and is used to change widget and force framework rebuiltUI. An example of a *Stateful Widget* can be a checkbox with “checked” state.

Stateful Widget does not have only `build()` method but has associated *State* object which defines several methods to support widget’s lifecycle. These methods are `initState()` for any state initialisation and `dispose()` to clear any allocated resources.

The state object is associated with widget’s *BuildContext.* This association is permanent, and state object will never change it [18]. Even if the widget’s *BuildContext* can be moved around the tree structure, the state will remain associated with that context. This implies that the *Stateful Widget* can be replaced during the tree rebuild with a new instance, but the state object is persisted.

1.2.2.5 Force Rebuild with setState()

As was mentioned, *Stateful Widge*t can tell the framework to rebuild, and the widget can be redrawn based on the changed state. The *Stateful Widget* has method `setState(callback)` which is used to do such rebuild. Inside `callback` a developer should change the widget’s state to the new value and framework will rebuild the widget based on that new state.

1.2.2.6 Case Study: Counter Application

Suppose application where are two buttons. One button increments a value (`counter`) and other decrements. The counter value is displayed within two *Text* widgets located on different places within the application. Whenever any of the buttons are clicked, and the value is changed, all *Text* widgets should reflect this change.

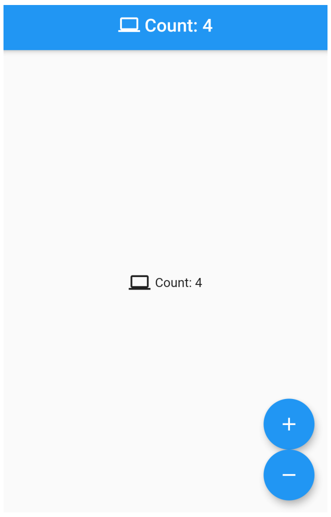


Figure 1.7: Counter Application.

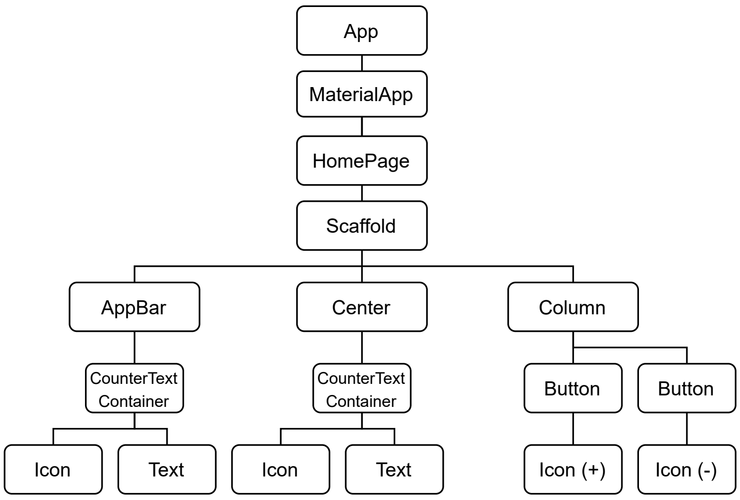


Figure 1.8: Counter Application's Widget Tree.

Application's layout is shown in Figure 1.7 and the corresponding widget tree in Figure 1.8 (shortened for brevity). In the *AppBar* and in the centre of the screen, there is *Text* widget which displays current value. On the bottom of the screen there are *FloatingActionButtons* widgets, which increment (decrement) counter value. The value needs to be accessible to the *Text* and to the button as well. Hence, the state is declared within the whole application's widget *HomePage*.

```dart
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}
```

Listing 2: HomePage Widget Definition.

Listing 2 shows the definition of the *HomePage* widget. The widget inherits from `StatefulWidget` and declares *HomePageState* which is an associated state object.

```dart
class _HomePageState extends State<HomePage> {
  int _counter = 0;
  void _incrementCounter() => setState(() => _counter++);
  void _decrementCounter() => setState(() => _counter--);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: CounterTextContainer(_counter)),
      body: Center(child: CounterTextContainer(_counter)),
      floatingActionButton: Column(
        children: <Widget>[
          FloatingActionButton(
            onPressed: _incrementCounter,
            child: Icon(Icons.add),
          ),
          FloatingActionButton(
            onPressed: _decrementCounter,
            child: Icon(Icons.remove),
          )]));
   }
}
```

Listing 3: HomePageState – setState() Example.

Listing 3 shows definition of *HomePageState* where state is represented as `int _counter = 0` variable. There are also two private methods for incrementing and decrementing `_counter` value. In each of the methods, the

```
class CounterTextContainer extends StatelessWidget {
  final int count;
  CounterTextContainer(this.count);

  @override
  Widget build(BuildContext context) {
    return Row(
        children: [
          const Icon(Icons.computer),
          const SizedBox(width: 5),
          Text('Count: $count')
        ],
    );
  }
}
```

Listing 4: CounterTextContainer – Accepting State As Parameter

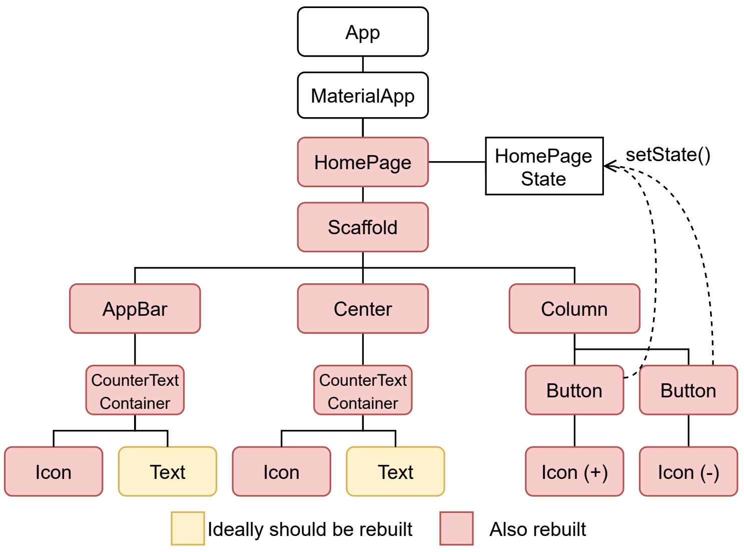


Figure 1.9: Expected Rebuilt vs. Actual Rebuilt.

`setState()` is called with the appropriate state change. These two methods are bound to `onPressed()` callback within *FloatingActionButton*. The state (`_counter` variable) is passed down to `CounterTextContainer` Listing 4 where the value is used to display within a *Text* widget.

The expectation is, that if the user pressed any of the buttons, the value is incremented or decremented respectively and part of the UI which depends on `_counter` value will be rebuilt. This part should be two *Text* widgets. In fact, the state is defined within the *HomePage* widget, and so, the whole *HomePage* and its children are rebuilt (Figure 1.9). In this small example, it is not really a problem and performance should not be affected. However, if the tree is deeply nested with heavy performance widgets (for example animations), it could lead to reduced performance and application can lags.

How to define the application-wide state and prevent the necessary rebuilding part of the widget tree is a subject of “state management” section.

1.3 State Management Approaches

Using `setState()` method is the perfect solution (and recommended) for local state. However, as soon as the state needs to be shared across multiple widgets, the managing this state becomes cumbersome. If the state should be shared, there is only one viable solution – lifting state up [20]. The state should reside in the widget, which is the parent for all widgets that needs access to the state. Such a solution can be used (and works) but creates a few problems.

The very first problem is if some of the interested widgets are kept in deep layer of the tree and others are not, the necessary amount of widgets is rebuilt whenever the state changes. The second issue is that the code of the widgets becomes very quickly hard to manage due to its tight dependencies with upper widgets. The only doable solution to provide access to the state is through references or function callbacks. These callbacks have to be passed to every widget down to the tree. This solution adds the necessary complexity to maintenance and readability. Last but not least, the application state usually has some business logic which should have been tested. However, if the state (and its logic) is put directly into the widget, this logic is hard to test.

The “State Management” is a term for patterns and design solutions which helps to prevent those issues – reducing necessary rebuilds, keeping code maintainable and testable.

1.3.1 Case Study Note

In our case study, there are two places where the state (counter value) is needed – first place is in the AppBar’s text and the centre of the screen. Secondly, the buttons for incrementing and decrementing need access to the state in order to modify it. The first version of counter application used solution with `setState()` and made the *HomePage* (a root widget of the whole application) as *StatefulWidget*. The state was then accessible from every HomePage’s children. Although this code is straightforward, there is already an issue with the testability of the business logic as it is tightly coupled to widget’s class.

Following lines in this section introduce some used patterns and solutions for state management. Every solution will use the same case study but with appropriate implementation. Each solution’s full code is available as an appendix. The design and functionality remain the same as introduced before.

1.3.2 Inherited Widget

One solution offered by *Flutter* framework is the concept of *InheritedWidget* [21] [18]. This concept is used across the framework – for example, obtaining current *Theme* or screen device information through *MediaQuery* object. Both can be accessed through convention "of" method – `Theme.of(context)` returns *Theme* object. Internally these objects makes usage of an *InheritedWidget*. The *InheritedWidget* has two features:

- It can be accessed from any widget directly.
- Whenever the widget changes, the accessing widget is automatically rebuilt.

The second implies that, for example, whenever widget access the *MediaQuery* and it is changed (the device is rotated, resolution changed,...), the widget is rebuilt to handle these changes.

```
class _CounterInherited extends InheritedWidget {
  _CounterInherited({Widget child, this.data})
        : super(child: child);

  final CounterModel data;

  @override
  bool updateShouldNotify(_CounterInherited oldWidget) => true;
}
```

Listing 5: _CounterInherited.

Listing 5 is implementation of *InheritedWidget*. It accepts child widget and `CounterModel` which is state (and business logic) holding class (Listing 7). The `CounterModelProvider` (Listing 6) is a widget which provides `of(context, listen)` method and should be called when widget needs access to the model. The optional parameter `listen` controls if a widget is automatically assigned to listening for changes or not. The `of` method make usage of `BuildContext` and traverses from given widget up to the root until it finds the widget searched for or fails.

The HomePage (Listing 8) is wrapped by `CounterModelProvider` to provide access to any its children. Within the HomePage's `build()` method, `CounterModel` is obtained without listening. It is used for button's callbacks to invoke `increment()` (`decrement()` respectively) method. Inside `CounterTextContainer` (Listing 9) is used *Text* widget and `CounterText` where the `CounterModel` is accessed to get current value. Note that every widget is stateless. The most important thing is that only the CounterText

```
class CounterModelProvider extends StatefulWidget {
  CounterModelProvider({this.child});

  final Widget child;

  @override
  CounterModel createState() => CounterModel();

  static CounterModel of(BuildContext context,
                         {bool listen = true}) {
    return (listen
        ? context.
            dependOnInheritedWidgetOfExactType<_CounterInherited>()
        : context.
            findAncestorWidgetOfExactType<_CounterInherited>()
        )
        .data;
  }
}
```

Listing 6: CounterModelProvider.

```
class CounterModel extends State<CounterModelProvider> {
  int _count = 0;
  int get count => _count;

  void increment() => setState(() => _count++);
  void decrement() => setState(() => _count--);

  @override
  Widget build(BuildContext context) {
    return _CounterInherited(data: this, child: widget.child);
  }
}
```

Listing 7: CounterModel.

is rebuilt when the state changes (Figure 1.10) in comparison with `setState()` approach where the whole application was rebuilt.

Although *InheritedWidget* solves many issues, the amount of code needed to achieve these results can be more unreadable and hard to maintain than approach with plain `setState()` method. However, Flutter's community created package which abstracts *InheritedWidget* and simplify this process.

```
// MyApp - wraping HomePage with CounterModelProvider
home: CounterModelProvider(child: HomePage()),
// HomePage's build method
final model = CounterModelProvider.of(context, listen: false);
return Scaffold(
    appBar: AppBar(title: CounterTextContainer()),
    body: Center(child: CounterTextContainer()),
    floatingActionButton: Column(
      children: [
        FloatingActionButton(
          onPressed: model.increment,
          child: Icon(Icons.add),
        ),
        FloatingActionButton(
          onPressed: model.decrement,
          child: Icon(Icons.remove),
        )
      ],
));
```

Listing 8: HomePage Implementation.

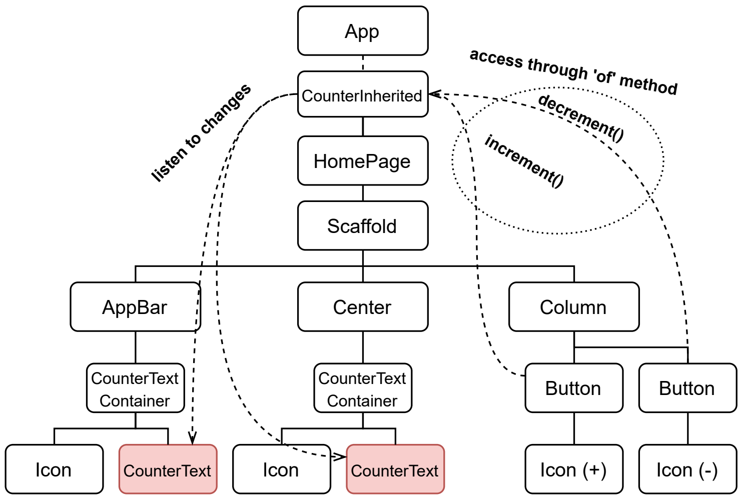


Figure 1.10: InheritedWidget Approach and Its Widget Tree.

1.3.3 Provider Package

Provider is a community package created by *Remi Rousselet* [22] as a simplification over inherited widgets with abstraction and more flexibility. One of the features is the concept of `ChangeNotifier` and its *ChangeNotifier-*

```
// CounterTextContainer's build method
return Row(
    children: [
      const Icon(Icons.computer),
      const SizedBox(width: 5),
      CounterText()
    ]);
//CounterText's build method
final model = CounterModelProvider.of(context);
return Text('Count: ${model.count}');
```

Listing 9: CounterTextContainer and CounterText Widgets.

```
class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }

  void decrement() {
    _count--;
    notifyListeners();
  }
}
```

Listing 10: Provider's CounterModel.

Provider. The concept behind is similar to the approach with the *Inherited-Widget*. The state is represented by the model class `CounterModel`, which uses mixin `ChangeNotifier` (Listing 10). This model contains only business logic and associated data. There are no widgets involved. If any of value is changed, the `notifyListeners()` method should be called to notify its listeners to rebuilt.

The *ChangeNotifierProvider* wraps HomePage widget, where `CounterModel` is created. Within HomePage's build method, the `CounterModel` is accessed through `Provider.of<CounterModel>(context, listen: false);` (note the similarity with *InheritedWidget*) without listening to changes (Listing 11). The *Consumer* widget used within *CounterTextContainer* (Listing 12) allows to automatically listening to changes and re-run its `builder` callback. If needed, the `child` argument can be used to construct widget which can be part of the rebuilding widget without rebuilding itself.

```
// in MyApp: provides CounterModel to descendants
home: ChangeNotifierProvider(
    create: (_) => CounterModel(),
    child: HomePage(),
),
// HomePage's build method
final model = Provider.of<CounterModel>(context, listen: false);
// ... rest of the HomePage's build method
// ... which is same as within InheritedWidget approach
```

Listing 11: Provider's HomePage.

```
//CounterTextContainer build method
return Consumer<CounterModel>(
  builder: (context, model, child) {
    return Row(
        children: [
          child,
          const SizedBox(width: 5),
          Text('Count: ${model.count}') // CounterText
        ]);
  },
  // child widget is not rebuilt when model changes
  child: Container(
    padding: const EdgeInsets.all(8.0),
    child: Icon(Icons.computer),
  ),
);
```

Listing 12: CounterTextContainer with Consumer.

The result is the same as with *InheritedWidget* (Figure 1.11), but with more readable code. The *Provider* package became very popular and encouraged by the Flutter team as a solution for state management [20]. Package offers more than the `ChangeNotifier` – it can be used for dependency injection and it is used by many other packages such as *flutter_bloc* [23].

1.3.4 Business Logic Component

Business Logic Component (BLoC) is a pattern originally introduced by *Paolo Soares* and presented during *DartConf 2018* conference [24]. The pattern was popularized by *Didier Boelens* [13] and it was inspiration to *Felix Angelov* for creation of *flutter_bloc* package [23] – a popular BLoC based state management solution.

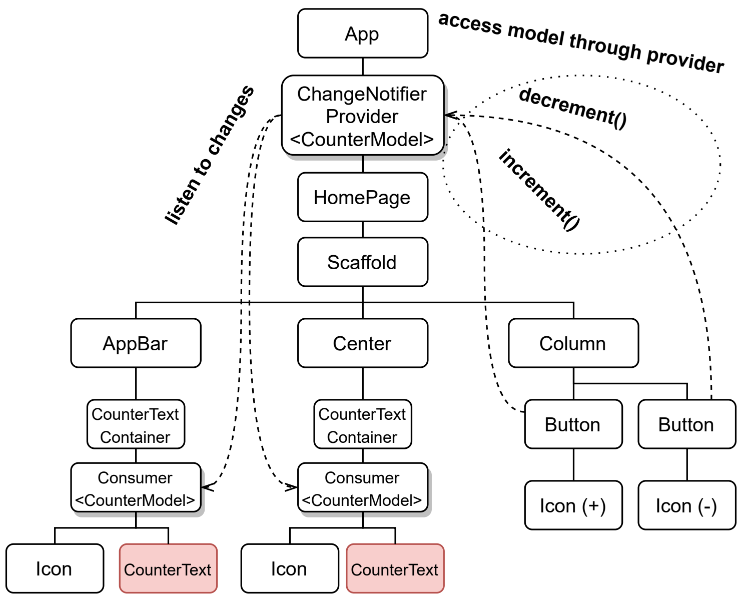


Figure 1.11: Provider Approach and Its Widget Tree.

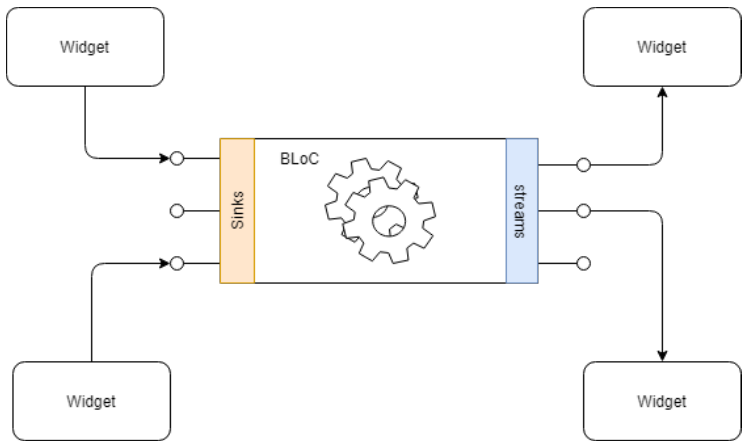


Figure 1.12: A BLoC Pattern [18].

The BLoC builds on the streams. The BLoC stands for class, holding business logic where on one side accepts stream of events (event sink) and on the other side provides stream of states (Figure 1.12). Through event sinks, BLoC accepts events which are processed and based on them a new state is put to the state stream. In terms of Flutter – widgets send events and listens for new state coming from state stream. The business logic itself is hidden from them and widgets (the UI) are concerned only about sending event and rebuilding themselves based on coming state.

The BLoC solves the responsibility separation where the logic is centralised within the BLoC class. This leads to better and easier testability. And third, thanks to the independence of UI with business logic, changing and organizing layouts can be done without changes within the application's logic. The UI is only concerned about building widgets based on the current state and eventually sending events if needed. This also imply that events can be sent from any place within the application without any complexity.

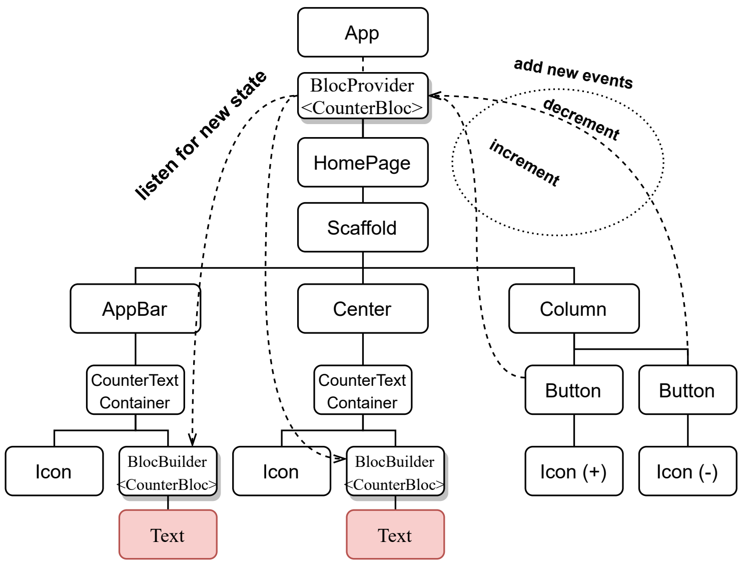


Figure 1.13: BLoC Approach and Its Widget Tree.

The BLoC also force centralized place where the particular state can be changed. With other solutions such as `setState()` and eventual callback passing, these changes can be spread across many places and it can be hard to maintain and test them. On the other hand, the usage of streams adds code complexity and robustness – a code boilerplate and the application has to be designed with asynchronous execution in mind as the BLoC relies on the streams. The mentioned package *flutter_bloc* is BLoC pattern with abstraction over the stream's complexity without downgrading their usefulness and advantages.

1.3.4.1 flutter_bloc Package

As before, the case study "counter application" was also written with the BLoC approach. Widget tree and its rebuild (Figure 1.13) is practically identical as with *Provider*. The difference lies in the how the code is organised, how the state is managed and the whole philosophy of the application's code. With *InheritedWidget* the state was mutated and its listeners were notified.

```
// CounterBloc's events
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  @override int get initialState => 0;

  @override
  Stream<int> mapEventToState(
    CounterEvent event,
  ) async* {
    if (event == CounterEvent.increment)
      yield state + 1;
    else if (event == CounterEvent.decrement) yield state - 1;
  }
}
```

Listing 13: CounterBloc's Implementation.

```
// App' build method -- providing CounterBloc
home: BlocProvider(
        create: (_) => CounterBloc(),
        child: HomePage()),

// inside onPressed() callback - add event to bloc
context.bloc<CounterBloc>().add(CounterEvent.increment),
```

Listing 14: BLoC Approach – Providing CounterBloc and Accessing Bloc.

```
Row(
children: <Widget>[
  const Icon(Icons.computer),
  const SizedBox(width: 5),
  BlocBuilder<CounterBloc, int>(
    builder: (context, state) => Text('Count: $state'),
  )]);
```

Listing 15: BLoC Approach – CounterTextContainer's Implementation.

With BLoC approach, whenever new event arrives, new state is returned and its listeners are rebuilt.

The *flutter_bloc* has base `Bloc<E,S>` class where `E` is an "event type" and `S` is a "state type". Each BLoC class has to inherit this base class and overrides `mapEventToState()` method in order to respond to any event and *yield* new state. How the package works and how it can be used is more discussed in Chapter 3. Listing 13 shows `CounterBloc` implementation. Events are represented as enum `CounterEvent` with *increment* and *decrement* respectively. The state is simple `integer` type. The *flutter_bloc* package uses under the hood *Provider* so it is very similar how the BLoC is provided to the widget tree. The BLoC is accessed with `context.bloc` in order to add new events. An example is shown in Listing 14. `CounterTextContainer` widget (Listing 15) uses `BlocBuilder` widget which rebuilds whenever new state arrives.

1.3.5 Conclusion

In this section, four approaches for state management was introduced. From simple `setState()` approach to stream based BLoC. As the BLoC encapsulates business logic into its own class and makes use of stream notion, this approach was chosen as the state management approach for implementation of our application due to convenient and straightforward way to usage, along with easy testability.

1.4 Flutter Internals

In the last section of this chapter, some of the internal Flutter's work is discussed. First of all, it is described more in-depth on how the framework is able to build widgets and draw them on the screen. Then the notion of Keys is introduced (an unique identification of widgets within widget tree). After that, some optimisations such as *const* constructors which can give better performance are discussed.

At the beginning of this chapter, Figure 1.3 shows how Flutter is made. The middle layer – the engine is responsible for rendering and orchestrating Flutter framework.

1.4.1 RenderObject and RenderTree

As was said earlier, Flutter uses pixel-perfect rendering – each widget is in the end translated into several pixels drawn on the screen. In order to do that, engine has a notion of the *Render Tree*.

The *Render Tree* is composed by objects called *RenderObjects*. These *RenderObjects* are used to [25]:

- define some area of the screen in terms of dimensions, position, geometry but also in terms of "rendered content",
- identify zones of the screen potentially impacted by the gestures,

The root object of the tree is called *RenderView*.

1.4.2 Everything Is a Widget Revisited

From a developer perspective, everything is a widget what is related to the UI in terms of layout and interaction [25]. The widget is an immutable class, where instances (or more precise its derivates) form the widget tree by composition. The widget itself, however, does not know how it can be rendered to the screen.

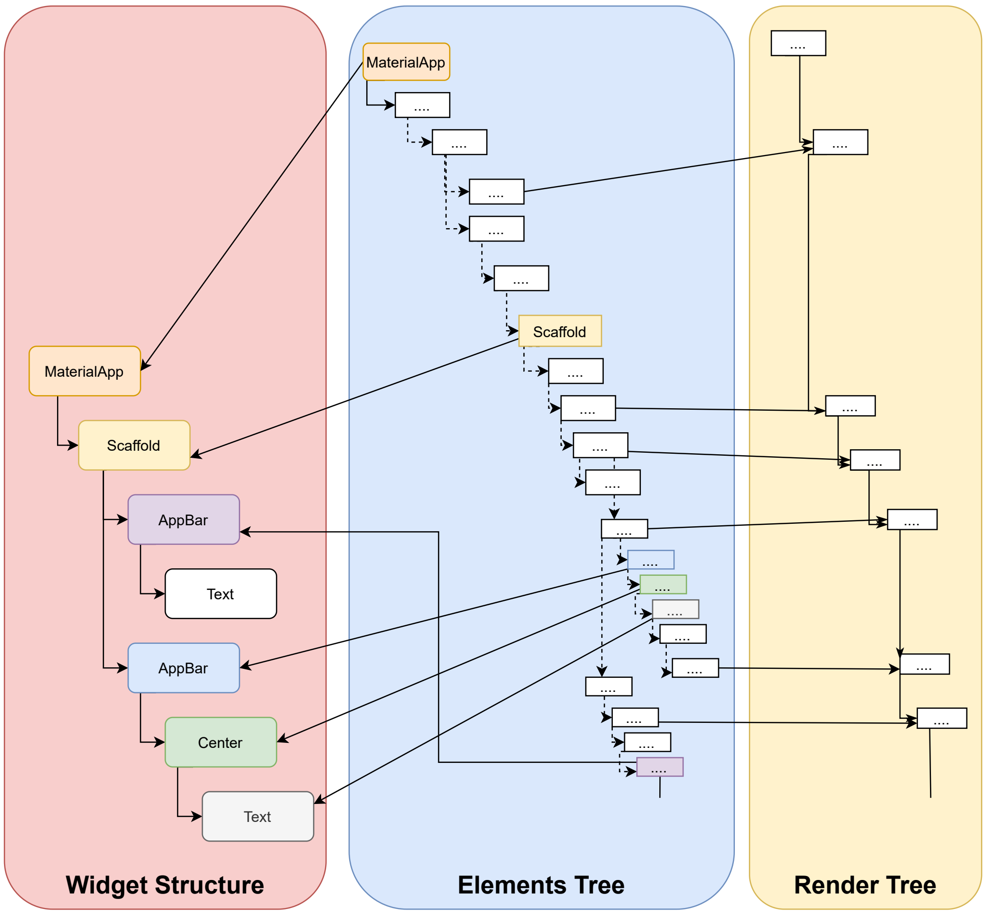


Figure 1.14: Three Trees – Widget, Element and Render Tree [25].

When Flutter needs to render the current state to the screen, the engine will request to inflate all widgets [25]. This can be simplified as unpacking the “box of boxes”. Each widget internally uses more granulated, and low-level API’s widgets in order to precisely describe the layout. Furthermore, from a developers perspective, the widgets create *Widget Tree*. In fact, internally each widget has assigned *Element* object which forms the *Element Tree*.

Each *Element* points to widget which created it, parent and potentially child *Element* and may also point to a *RenderObject* [25]. Figure 1.14 shows notion of *Widget Tree*, *Element Tree* and *Render Tree*.

Every Widget can be assigned to one of the three categories [25]:

1. **The proxies** – these widgets hold information which needs to be available to other widgets – such as *InheritedWidget*.
2. **The Renderer**s – These widgets define the layout of the screen, such as *Row*, *Column*, *Padding*, ...
3. **The Components** – These widgets provide final information related to how the piece of UI should look. An example of such a Widget can be *Text* or *RaisedButton*.

Depending on the widget category, a corresponding *Element* type is associated. There are two main *Element* types – *ComponentElement* and *RenderObjectElement*. The first one does not directly correspond to visual rendering. The latter one has a connection to *RenderObjects*. Also, every widget has its corresponding *Element* object, and speaking of *StatefulWidgets*, each widget has corresponding *StatefulElement* where the state is associated. This statement also implies that *BuildContext* is an *Element* itself.

1.4.3 Deciding What to Redraw

A redraw mechanic relies on invalidating either an *Element* or a *RenderObject* [25]. Whenever the widget should be rebuilt, the corresponding *Element* is marked as dirty. Invalidation of *RenderObject* can happen, for example, when changes to its dimension, position or geometry are made or when the associated *Element* is marked as dirty. When engine decides that new repaint should happen, it iterates over all invalidated (dirty) elements and request them to rebuild. Internally the rebuild works as [25]:

1. The corresponding widget's `build()` method is called which returns a new widget.
2. If the *Element* has no child, the new widget is inflated.
3. Otherwise the new widget is compared to the one referenced by the child *Element* and
   - if they are same (same widget type and Key), the update is made and the child *Element* in the *Element Tree* is kept,
   - if they are not same, the child *Element* is unmounted (and discarded) and the new widget is inflated.
4. The inflating of the widget leads to creating a new *Element*, which is inserted into the *Element Tree* as a new child of the *Element*.

After that, the *Element Tree* is considered as a stable and a similar process is made with *Render Tree* – every *RenderObject* marked as dirty performs its layout (calculating dimension and geometry), every *RenderObject* marked to repaint is repainted. In the end, the device screen is redrawn.

1.4.4 Notion of Keys

When comparing new widget with current one within *Element*, the *Element* decides to rebuild when the widget type and *Key* are different. A *Key* is an object associated with a widget. In the simplest form, a *Key* can be considered as the unique identification of the widget. In most cases, developers should not need to work with the keys as Flutter manage them internally. However, there are cases where the manual definition of the *Key* is necessary.

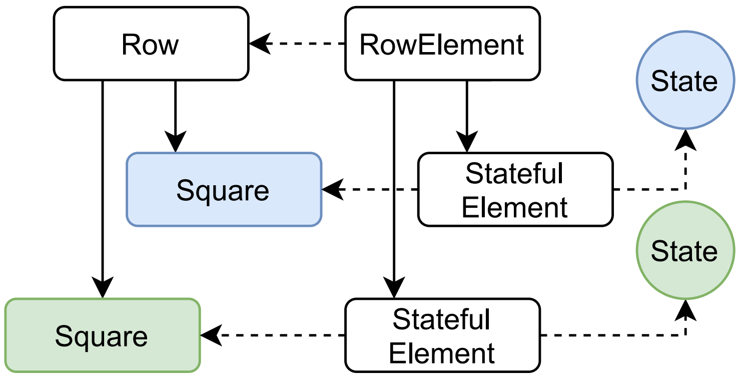


Figure 1.15: Square Widgets and Associated Elements with States.

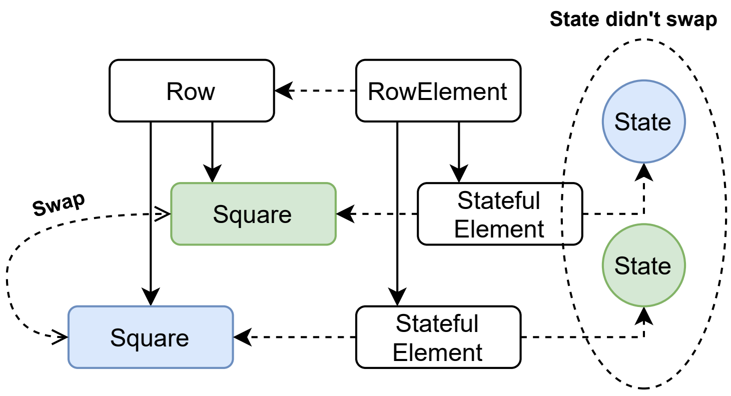


Figure 1.16: Square Widgets After Swap and Associated Elements With Wrong States.

```
// SquarePage holds list of Square widgets
class _SquaresPageState extends State<SquaresPage> {
  final squares = [Square(RandomColor.get()),
                   Square(RandomColor.get())];
  void _shiftSquares() {
    setState(() => squares.insert(1, squares.removeAt(0)));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Row(children: squares),
        floatingActionButton: FloatingActionButton(
          onPressed: _shiftSquares,
          //...
        ),
      );
   }
}
```

Listing 16: SquarePage Widget With Stateless Square Widgets.

```
class Square extends StatefulWidget {
  Square({Key key}) : super(key: key);
  @override
  _SquareState createState() => _SquareState();
}

class _SquareState extends State<Square> {
  Color color;
  @override
  void initState() {
    color = RandomColor.get();
  }

  @override
  Widget build(BuildContext context) {
    return Container(color: color,width: 100,height: 100);
  }
}
```

Listing 17: Square Widget as StatefulWidget.

The problem can occur when some widget uses a collection of widgets of the same type that holds some state. Consider a concrete example where `SquarePage` holds a list of `Square` widgets (Listing 16). Each `Square` (as a *StatelessWidget*) has defined random colour through a constructor. After a button is clicked, the squares are swapped. With `Square` as *StatelessWidgets*, everything works as expected. However, if the `Square` becomes *StatefulWidget* (Listing 17) and the button is clicked, it seems like nothing happened – squares stay on the same place. When the widget is marked to rebuilt, it walks through *Elements* and if the widget type and the *Key* match, the *Element* updates its reference to new Widget. In the case of *StatefulWidget*, the associated state is linked to the *Element* object (Figure 1.15). When squares are shifted, the *Element* is marked as dirty. It walks through square's *StatefulElement* and checks if the widget type and the *Key* match. They match because no *Keys* are assigned to them. Hence, the *Element* updates its widget reference, but the associated state remains the same (Figure 1.16). The key is to add *Key*. There are several types of Keys such as *ValueKey*, where some unique value can be assigned (for example article's id). For `Square` example, the *UniqueKey* which generates unique identification is enough for usage. After the *Keys* are assigned,

```
final squares = [Square(key: UniqueKey()),Square(key: UniqueKey())]
```

the example works again as expected. The full example code is available as before within appendix.

The keys should be put to the most top widget, which is used as a root widget of collection. Otherwise, the rebuilding algorithm fails once again, and wrong behaviour will occur. In practice, *Key* should be used when stateful widgets are used within collections (such as *ListView*, *Row* or *Column*) and they are manipulated – moved, removed and similar. Moreover, sometimes the *GlobalKey* can be used to manage some widget's state "outside". This approach is often used with managing text inputs.

1.4.5 Const Optimisation

On of the Dart's features are *const* constructors which makes instance as a *compile-time constant*. This feature can be used to optimise widget builds and prevent unnecessary rebuilds. Most of the Widgets has *const* constructor, and if possible, the *const* constructor should be used. Such as *Icon* widget or *Text* widget if they accept non-changing values they can be made compile-time constants and Flutter when rebuilding the tree will these Widget reuses instead of creating a new one.

The problem of performance optimisation is a vast subject for discussion, and it could have its own chapter. The *const* constructors are only "tip of the iceberg". In general, it is somewhat common sense to avoid unnecessary

UI rebuilds or avoid using animations carelessly such as animate each line of *Text* within *ListView* whenever the list is updated.

1.5 Conclusion

The first chapter introduced Flutter framework and its philosophy “everything is a widget” as a primary user interface building block. The notion of state and several state management approaches were discussed, how they can be implemented and how they affect the rebuilding of UI. In the end, part of Flutter internals was uncovered and explained to grasp a better understanding of how the framework works.

Chapter 2

Coffee Time Analysis

In this chapter, the specification of the implemented application is outlined. The analysis of similar applications was made to obtain ideas. After analysis, the low fidelity prototype was created to outline the first vision of the final application. Next, the high-fidelity prototype, along with Nielsen's heuristics analysis and user testing, were made. At the end of the chapter, considered tools and services which are used to implement the application are briefly described.

2.1 Considered Application

Coffee Time is an application focused on searching nearby cafes. Users should be able to search and find nearby cafes around them and decide which place to visit. Each cafe is displayed with information such as distance, user's reviews, photos, opening hours and more.

Added value to this standard information is a feature so-called "the tags". These tags are user-added additional info which describes more precisely what given cafe offer or for instance if that cafe allows pets inside.

The set of tags is defined, and users can add these tags to the cafe. Each tag can be reviewed by other users. These reviews are done through "like" and "dislike" functionality. The purpose of tag's reviews is to prevent outdated or misleading information. The example of such review can be "User visited cafe which has tag 'dog friendly', but unfortunately this information was incorrect. Consequently, the user decided to open the application and review the cafe's tag 'dog friendly' with dislike."

Together with likes and dislikes, each tag has a computed score. Each like gives to score plus one and as an opposite, dislike minus one. If any tags reach the score to -4, the tag is removed from the cafe, more precisely is not shown anymore to users. If such removed tag is proposed by any user again, it obtains "like" review. Thus score is incremented to -3 and the tag is shown again.

The application is location-based and offers a map view to support the convenient user experience. Any cafe can be marked as user's favourite to give a faster way to find cafe whenever the user wants.

In conclusion, Coffee Time is application focused on one domain – searching nearby cafes in order to know where to go to study, talk with friends or for example have a great coffee. The application should be simple to use with a clean user interface.

2.2 Use Cases

From the specification above, the use cases and use case scenarios were formed. The use cases diagram is listed in Figure 2.1 and shows every use case from the user perspective. Technically there is the role of application administrator who can check control back-end or available application's tags, but it is skipped due to the lack of importance from application perspective.

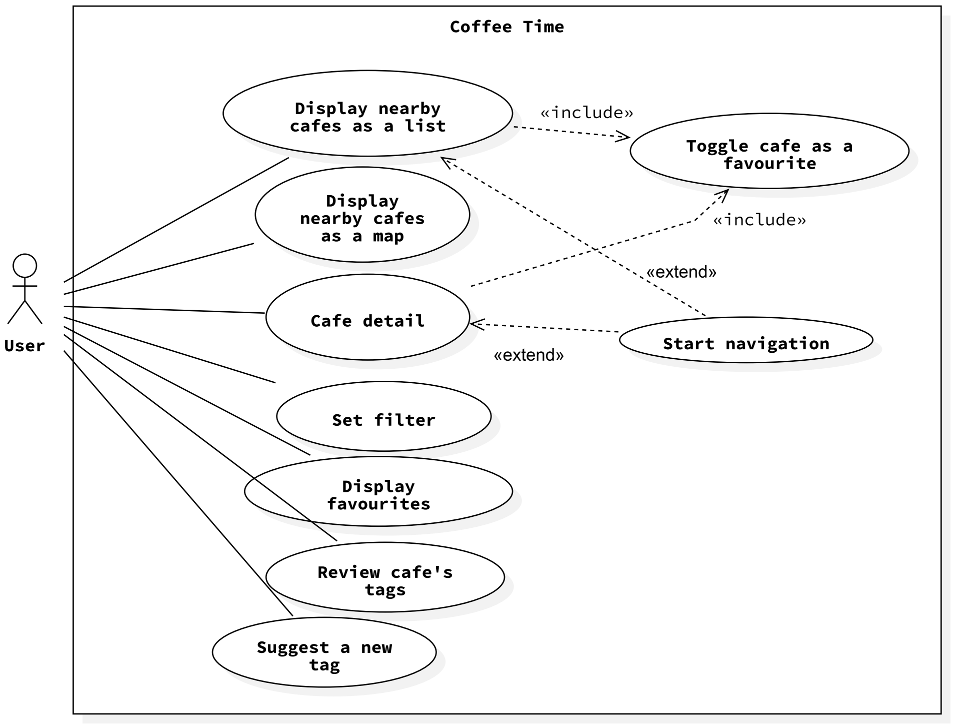


Figure 2.1: Application Use Case Diagram.

As shown on diagram, the application has several use cases:

- UC1: Display nearby cafes as a list.

- UC2: Display nearby cafes as a map.
- UC3: Start navigation.
- UC4: Toggle cafe as a favourite.
- UC5: Setting the filter.
- UC6: Display favourite cafes.
- UC7: Review the cafe's tags.
- UC8: Suggest a new tag.

2.2.1 UC1: Display Nearby Cafes As a List

An user can display nearby cafes in the form of the list view. The result is filtered by setting a filter, which can be altered by *UC5*.

Pre-Conditions An user must be on the cafe list screen.

Basic Flow

1. An user launch Coffee Time and lands on cafe list.
2. Cafe list shows nearby cafes around him. Each cafe is displayed in the form of a card.
3. An user can pull the list down to refresh results.
4. An user taps on cafe's card and is redirected to the detail view.

Alternative Flow 1 An user launch navigation to the selected cafe.

2.2.2 UC2: Display Nearby Cafes As a Map

An user can display nearby cafes in the form of the map view. Each cafe is shown as a map marker.

Pre-Conditions An user must be on the map screen.

Basic Flow

1. An user launch application and change screen to map view.
2. Nearby cafes are shown as markers.
3. An user taps on the marker and is redirected to the detail screen.

Alternative Flow 1 An user taps anywhere on the map to display nearby cafes on the selected location.

2.2.3 UC3: Start Navigation

This use case allows starting navigation to selected cafe through native navigation applications.

Pre-Conditions The navigation services must be enabled.

Basic Flow

1. An user selects the cafe.
2. An user selects navigate action.
3. The system request to open navigation application is opened.
4. An user enables navigation and is redirected to navigation application.

Alternative Flow 1 An user dismisses navigation request and cancels navigation.

2.2.4 UC4: Toggle Cafe As a Favourite

Each cafe can be marked as a favourite to faster future access.

Pre-Conditions The cafe must be loaded, so that it is visible to the user. An user must be either on the cafe list screen, detail screen or favourite screen.

Basic Flow

1. An user has a cafe which wants to toggle as a favourite.
2. An user toggles cafe as a favourite.

2.2.5 UC5: Setting the Filter

An user changes the filter settings to filter out the cafe results.

Pre-Conditions There are results to filter.

Basic Flow

1. An user opens filter settings screen.
2. If it is suitable user changes results ordering from “by distance” (default) to “by popularity”.
3. If it is suitable user changes opening hours filter.

4. Add tags to filter by, if any.
5. Returns back to the previous screen
6. The results are filtered with the chosen filter.

2.2.6 UC6: Display Favourite Cafes

Display every favourite cafe in the form of the list view.

Pre-Conditions An user must be on the map screen.

Basic Flow

1. An user displays favourite cafe list.
2. After the cafe is selected, the user is redirected to the cafe's detail screen.

Alternative Flow 1 An user launches navigation to the selected cafe.
Alternative Flow 2 An user toggles cafe as not-favourite anymore.

2.2.7 UC7: Review the Cafe's Tags

Use case allows users to review the cafe's tags with likes and dislikes.

Pre-Conditions Cafe must have tags to review. User must be on the cafe's detail screen.

Basic Flow

1. An user wants to suggest a change to the selected cafe.
2. An user reviews each tag with "like", "dislike" or skip review for the given tag.
3. An user confirms review.

Alternative Flow 1 An user decides not to do the review and goes back to the detail screen.

2.2.8 UC8: Suggest a New Tag

Use case allows users to suggest a new tag to the selected cafe.

Pre-Conditions Cafe must have tags to review. User must be on the cafe's detail screen.

Basic Flow

1. An user wants to suggest a change to the selected cafe.
2. An user selects new tags for the suggestion.
3. An user confirms the suggestion.

Alternative Flow 1 An user decides not to make the suggestion and goes back to the detail screen.

2.3 Existing Alternatives

The analysis of existing alternatives was made to research already created applications with similar features. Existing applications were searched through Android's official store. Applications with these functionalities were chosen for the review:

- nearby place search,
- application's theme should be cafes or similar.

For comparison the most five inspiring and distinguish applications were chosen. The following lines briefly describe one of each, their target audience, the advantages and drawbacks.

2.3.1 Gastromapa Lukáše Hejlíka

Published in the first quarter of 2019 as a new application for exploring restaurants in the Czech Republic. The application's speciality is that restaurants' reviews are not done by users but by gastronomy specialist *Lukáš Hejlík*.

As soon as the application launches, it displays nearby restaurants. Each establishment is shown as a card with important information such as an address, distance, and type of restaurant. The main card's focus is a large photo which should catch the user's eye to take a look.

After the card is clicked, the user is presented with the restaurant's detail, where more information such as opening hours, map location and comprehensive review by *L. Hejlík* can be found. The navigation to the chosen restaurant can be launched from this view as well. The target audience is anyone who seeks to visit unknown places and possess the opportunity to taste great food.

2.3.1.1 The Advantages

- Design is fresh, clean and users can immediately see relevant content.
- Thanks to clean design the application is easy to use and understand.
- The whole application behaves smoothly without any noticeable freezing.

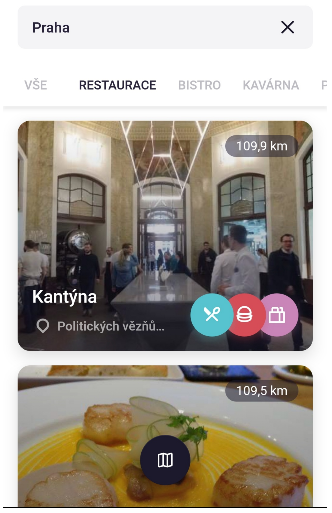


Figure 2.2: Gastromapa *L. Hejlíka* [26].

2.3.1.2 The Drawbacks

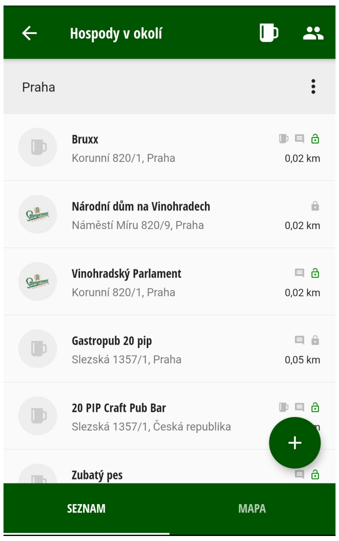
- The navigation button has a blackish colour that after scrolling disappears. If the restaurant has a darker photo, the button is hard to notice.
- When coming back to the main screen, loading of the list is started again, and the previous search is lost.
- Detail screen on entry is fully covered with restaurant photo. From a design point of view, it is a nice touch, but users must scroll to see any information.

2.3.2 Pivní Deníček

Application *Pivní deníček* is used to search nearby pubs in the Czech Republic and their beer offer. The content is created by the community, including served beer and their prices. The application offers searching nearby restaurants filtered by beer brands. Each user can view a history of places they have visited. Furthermore they can mark any pub as their favourite and share their experience.

2.3.2.1 The Advantages

- Pubs are displayed as a list or points on the map.
- The served brands are displayed directly within the list, so it is not needed to visit details.



Bruxx
Korunní 820/1, Praha 0,02 km

Národní dům na Vinohradech
Náměstí Míru 820/9, Praha 0,02 km

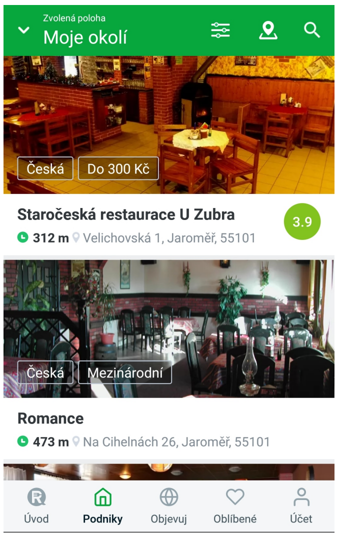
Vinohradský Parlament
Korunní 820/1, Praha 0,02 km

Gastropub 20 pip
Slezská 1357/1, Praha 0,05 km

20 PIP Craft Pub Bar
Slezská 1357/1, Česká republika

Zubatý pes

Figure 2.3: Pivní deníček [27].



Staročeská restaurace U Zubra
312 m Velichovská 1, Jaroměř, 55101

Romance
473 m Na Cihelnách 26, Jaroměř, 55101

Figure 2.4: Restu [28].

- The registration is optional for searching. If users want to contribute, they have to have an account.

2.3.2.2 The Drawbacks

- A registration can be done through Facebook or e-mail. With e-mail registration, the user is forced to leave the application and is redirected to the web page where registration is finished.

- As was said, content is created by the community. During the research, it was clear that many information is outdated or misleading.

- Overall the application design looks outdated and does not meet current, modern, trends.

- On the primary screen, there are displayed user's stats and the most three nearest pubs. The drawback is that on the larger screens, there is plenty of unused space.

- Each restaurant displays only one brand of drafted beer. Nowadays, many pubs offer more than one brand.

- The side menu can be opened only with the hamburger icon but not with slide to the right gesture.

2.3.3 Restu

Restu is another gastronomy guide focused on restaurants in the Czech Republic. Through this application reservations can be made. Application has many unique functionalities. For example “discover” section which shows attractive offers or the best cafe in the city. Another functionality is the “check-in” button which gives credits to the users if they eat at the given restaurant. Target audience is everyone who searches for new places where to eat and make a reservation.

2.3.3.1 The Advantages

- Clean and well-structured layout.
- Opt-in registration.

2.3.3.2 The Drawbacks

- When a restaurant card is selected, window of the restaurant pops up but at the bottom cannot be hidden again.
- To review the restaurant, the user has to be signed in and the restaurant must be open. If the restaurant is closed, the review cannot be added.

2.3.4 Zomato

Zomato started as a web-based restaurant browser. Application has its own database of establishments and its content is edited by users. On the primary screen there are displayed “week hits”, top restaurants, or “happy hours”. Restaurants are divided into categories such as “Nightlife” or “Daily menu” which helps for navigation within the application. Target audience is anyone who wants to try new restaurants or someone who is looking for action offers.

2.3.4.1 The Advantages

- Well solved filtering. The filter setting is intuitive and it displays the most used filters.
- Advanced options for filtering with tags such as “dog friendly” or “Wi-Fi free”.
- Friendships with other users. If another user added a review, a notification is received.

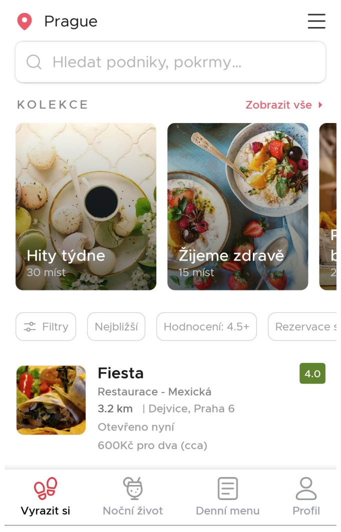


Figure 2.5: Zomato [29].

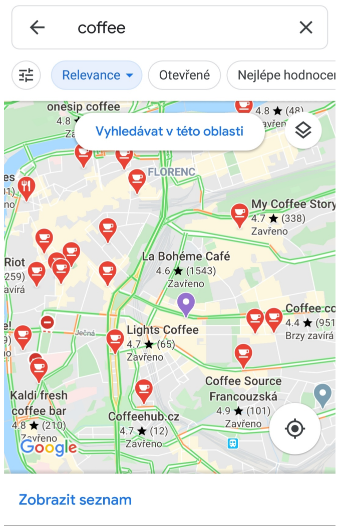


Figure 2.6: Google Maps [30].

2.3.4.2 The Drawbacks

- The primary screen is cluttered with many information at one place.
- Nearby restaurants list is hidden below “favourites restaurants” and “month collections”.
- The full-text search in some circumstances behaves unexpectedly. For example, to search for restaurants which offer “Asian food” user has to type exactly “asian” but not “asia”.
- Readability of some text is worsened by light background and greyish text colour. In some scenarios, it is hard to read the content.

2.3.5 Google Maps

Popular worldwide map service by *Google*. One of the world’s biggest database of places, including restaurants. *Google maps* for each business, establishment displays additional info such as user reviews, photos, prices. Within Android system is already installed. Nearby places can be searched directly from the map.

2.3.5.1 The Advantages

- Well known and tested user interface which is embedded often to another application.
- No registration is required.

- A place detail includes plenty of useful information.
- GPS navigation with one click.

2.3.5.2 The Drawbacks

- Not domain focused, that means it does not offer focused content on particular businesses such as restaurants.
- No advanced filtering and result sorting.

In conclusion, five applications were analysed on the Android system. Three apps are focused mainly on gastronomy. Another one specialises on beer. Last one, *Google maps* is one of the most universal and robust. Each application has its own unique UI and overall user experience differs. In Table 2.1 the targeted audience and user interface usefulness is summarised.

Application	Targeted audience	Overall UI
Gastromapa Lukáše hejlíka	Everyone	Great
Pivní deníček	Beer drinkers	Bad
Restu	Everyone	Bad
Zomato	Everyone	Good
Google	Everyone	Great but complex

Table 2.1: Analysed Applications User Interface Summarization.

2.4 Application Prototype

One crucial step during the creation of software product is prototyping. Prototypes can help introduce different design ideas, can be easily tested, evaluated and changed. Prototyping techniques differ, but the desired output is the same – provide visually concept of the final product. Prototypes do not help only visually, but they are part of user experience research and can find out which parts of the user interface should be changed before it is implemented.

There is no correct definition of how prototypes should look like and how they should be created. The prototype can be made from the form of a simple sketch on paper to sophisticated pixel-perfect application [31]. Prototypes can be created multiple times during the whole creation process.

In the early stages, Low Fidelity (Lo-Fi) prototype is typically created. With Lo-Fi, the application can be evaluated and user-tested if desired design concept is usable and understandable for users. When Lo-Fi is finalised, the next prototype – High Fidelity (Hi-Fi) is created. Hi-Fi comes out from Lo-Fi and should behave as fully functional application on the target platform. With Hi-Fi once again, the application is evaluated with users and tested.

To be more precise, according to [31], Lo-Fi prototype is a way to translate high-level concepts into tangible and testable artefacts. The most significant functionality of Lo-Fi prototypes is to check and test the functionality of the product before visual appearance. Advantage of Lo-Fi is that it is inexpensive, fast way to propose prototype. On the other hand, Lo-Fi lacks complexity and cannot supply advanced interactivity. Lo-Fi should be used to quickly create a prototype and get user feedback in the early stages of the creation process.

After the Lo-Fi prototype, the Hi-Fi is created. This prototype looks and feels as similar as possible to the actually built application. Hi-Fi should be created on the targeted platform and behave as it is the final product. The goal is to have more complex UI interactivity and have better feedback from user testing. Thanks to the fact that prototype looks like a real application, user behaves more naturally and can give more precise, a meaningful feedback than with Lo-Fi prototype.

In conclusion, Lo-Fi prototypes are tested only internally with a small number of users and can be iterated more often and faster. The Hi-Fi should be created and tested after Lo-Fi prototype was accepted, because the Hi-Fi is more expensive to create. On the other hand, Hi-Fi gives better feedback from user testing, thus provides more valuable information.

2.4.1 Coffee Time Prototype

After the specification was written, next step was to create a *task list*. The task list is written from the user's perspective – each task describe an user's action. It should tackle all important functionalities and even obvious one such as "add record" or "remove record".

Because the task list can become very long, it is usually transformed into a *task graph*. The task graph does not have any specific definition, but it should contain every task along with each available screen within the application. Coffee Time's task graph is listed in Figure 2.7. The blue rectangles are screens and the yellow ones are any task what users can do. The application's entry point is highlighted with bold blue rectangle.

A note about *User Interface Design (MI-NUR)* subject which was held during the winter semester of the academic year of 2019/2020. The Coffee Time prototype was crate as a semester work. Sincere gratitude to classmates *Bc. Ondřej John* and *Bc. Vojtěch Polcar* for their co-working on the prototype. Furthermore, much appreciation to *Ing. Jiří Hunka* for feedback and guidance during the work. The classmates helped during prototyping, proposing functionalities, researching and testing. The High-Fidelity prototype was implemented only by the thesis author.

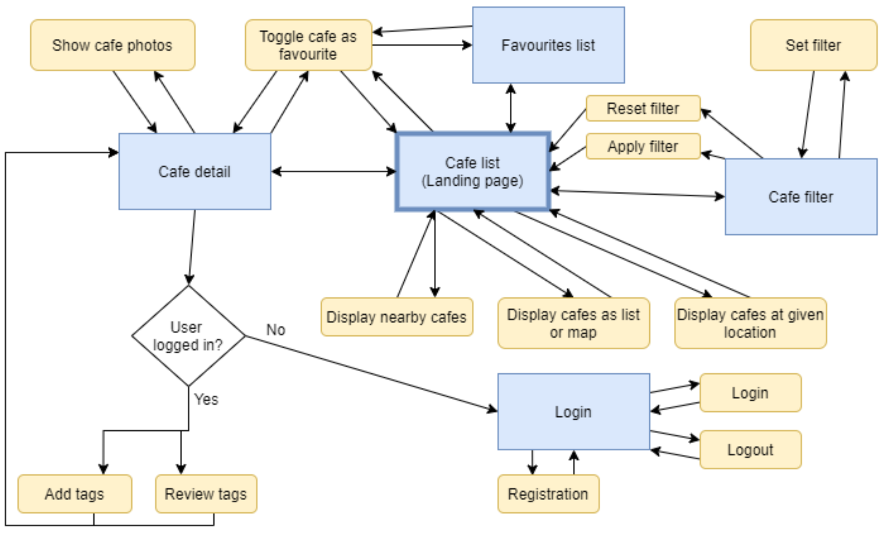


Figure 2.7: Coffee Time Task Graph.

2.4.1.1 Low Fidelity Prototype

As a task graph was defined, the *Lo-Fi* prototype could be made. Although the application is considered as multi-platform application, the prototype was focused on Android, and its Material design [32]. The inspiration was taken from typical Material layouts, such as *AppBar* with title and subsequent actions or tabs the bottom of the screen.

First of all, the rough prototype was drawn on paper. Its purpose was to come up with some ideas and considered layout. After that, the *Balsamiq* [33] prototyping software was used. The *Balsamiq* tries to mimic pencil and paper. The prototype is created with a set of components which looks like they are drawn by hand. The most important feature was the ability to create deep links between screens or components. With a few clicks, the prototype was able to handle actions such as the open application menu or navigate to detail. With that tool, the clickable prototype focused on essential app's features was created. As a result, the PDF was exported. The PDF is enclosed as part of the thesis located at `prototype/lofi.pdf`. The portion of the result is shown in Figure 2.8.

The result was tested with co-workers and closest author's family members. From the testing session, useful feedback was given, which is listed along with short answers.

1. **On the cafe list, what if the cafe has more tags than it can display in the row. How to solve it?**

   The solution is to calculate free screen space and display portion of available tags.

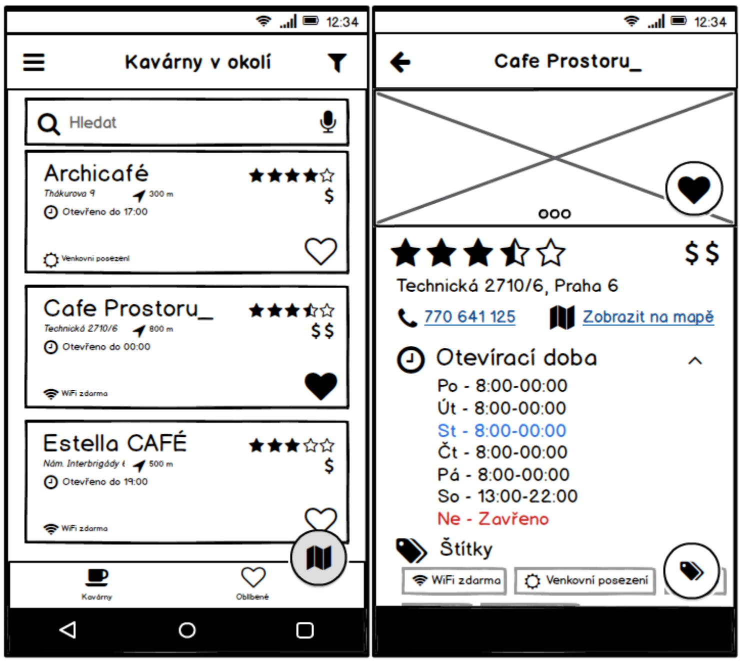


Figure 2.8: Lo-Fi Prototype. Cafe List (Left) and Detail Screen (Right).

2. **Research more on how to solve user's reviews.**

   As a considered data source is Google Places API, it was acknowledged that it is not possible to add custom reviews through their API.

3. **Navigation and contact buttons are too small.**

   Taken into account during implementation.

4. **If the tag in detail is clicked, the cafe list with the given tag is shown.**

   Taken into account as a valuable tip.

5. **Focus on usability with mobile devices, mainly when the user's walk or are in public transport.**

   The usability should be more tested.

6. **Focus more to provide understandable information about what tag is and how to use it.**

   Information and usability should be more revisited.

2.4.1.2 High Fidelity Prototype

The Hi-Fi prototype was created as Flutter application. Because of that, in the future, the already written code could be reused. The aim was to create a fully functional prototype for Android devices. As was said earlier in this chapter, the aim of Hi-Fi is to provide an application so it behaves as real one, which is mainly focused on user interface interaction. That means that the application does not have any real communication with back-end services. For Coffee Time there was prepared local JSON data source with randomly generated cafe names and their data. Besides that, a few, real one cafes were added to be less general and more known for potential testers. The Hi-Fi focused on all earlier described use cases. The cafe list screen and cafe's detail screen is shown in Figure 2.9. The prototype's source code can be found at `https://github.com/petrnymsa/coffee-time/releases/tag/prototype` [34].

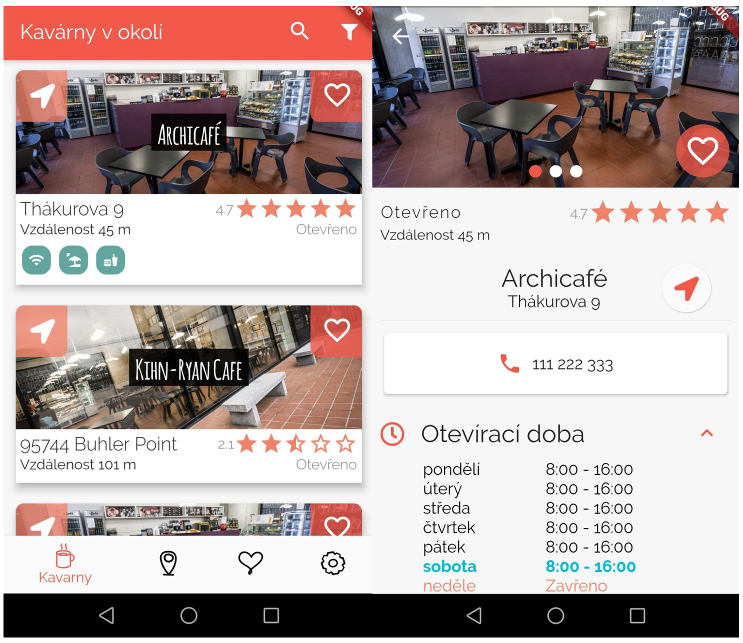


Figure 2.9: Hi-Fi Prototype. Cafe List (left) and Detail Screen (right).

2.4.2 Nielsen Heuristic

Nielsen Heuristic [35] is a usability engineering method for finding the usability problems in a user interface design so that they can be attended to as a part of an iterative design process. The heuristic evaluation involves a small set of rules and these rules should be judged by a small group of "evaluators".

Following lines describe one of each of ten rules. Each description is taken from article [35].

1. **Visibility of system status** – The system should always keep users informed about what is going on through appropriate feedback within a reasonable time.

2. **Match between system and the real world** – The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

3. **User control and freedom** – Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

4. **Consistency and standards** – Users should not have to wonder whether different words, situations, or actions mean the same thing.

5. **Error prevention** – Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

6. **Recognition rather than recall** – Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for the use of the system should be visible or easily retrievable whenever appropriate.

7. **Flexibility and efficiency of use** – Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. **Aesthetic and minimalist design** – Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

9. **Help users recognise, diagnose, and recover from errors** – Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

10. **Help and documentation** – Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

2.4.2.1 Coffee Time Evaluation

During *MI-NUR* lecture, the above-described heuristic was evaluated against Hi-Fi prototype. Each screen was taken individually and judged with all rules. Note that only found rule violations are described for each screen.

- Cafe list
  - Rule #2: Some tags icons are hard to understand, and their meaning can be easily misunderstood.
  - Rule #8: When filtering is on, it occupies nearly one-third of the screen.
  - Rule #9: There is no visible error handling for missing internet connection or location services.
- Favourites cafes
  - Rule #2: Same as Cafe list.
  - Rule #3: When a favourite cafe is set off, there is no indication that action is permanent without the ability to undoing it.
- Detail screen
  - Rule #2: Same as Cafe list.
  - Rule #6: It is not clear how to start navigation. This functionality is provided by the button to the right of the cafe's address. There should be a more visible indication that this action is possible.
- Map screen – No rules violation found.
- Filter screen – The only violation is the same as within Cafe list about icons.

Once the violations are found, they should be prioritised. The priority indicates how serious violation is. The more severe violation, the more unusable is it for users. The priorities are:

- 1 – negligible,
- 2 – visible issue,
- 3 – the usability can suffer,
- 4 – frustrating experience,
- 5 – User is not able to use application at all.

The most violated rule is number #2 with tag icons. With this issue suffer most of the screen, because icons are used very often. This issue has priority 3. The second issue with favourites cafes (priority 2) can be solved, for example, with small notification with the undo button. The prototype does not have implemented any kind of error handling regarding internet connection or location services (priority 2).

2.4.3 User Testing

The prototype was tested with seven testers. An important note is that four testers are people who have "IT knowledge" and are classmates from university. It was assumed that their understanding could vary against other testers.

The process of testing was as follows – application was launched on a real device. Each testing was recorded. The recording was essential to know what testers thought, how they used the application and behaved. Each user had the same set of use case scenarios, and the supervisor did not interact with tester either provide any advice to testers how to proceed. There was only one exception when the supervisor was allowed to advise, and that was when tester did not know how to proceed – he became "totally lost". This situation could indicate two different things. First, from the test case scenario, it was not particularly evident what tester should do. Secondly, the test case was understandable, but the application interface not. The second problem is more important for the testing and the results. The test case scenario can be rewritten or with the help of supervisor fully explained.

The description of test case scenarios, along with established issues, follows. Each test case has an introduction for users in which situation they are. Next contains test case goal, expected user behaviour and actual behaviour. Note that, the actual behaviour is not part of this summarization. Instead of, the found issues are summarised.

2.4.3.1 Test Case 1: Favourite Cafe

An user visits his favourite cafe and he would like to add this Cafe to his favourite list.

- **Goal** – Add Cafe to favourites.
- **Expected Behaviour**
    1. An user is located on the cafe list screen. The targeted cafe is displayed.
    2. An user clicks on the heart icon located in top right corner of the cafe's tile.

3. Alternative flow: An user is located on detail screen and taps heart icon located on the top of the screen.

- **Found Issues** – The testers went through a test case without any issues.

2.4.3.2 Test Case 2: Start Navigation to Selected Cafe

An user has selected cafe and wants to navigation to the cafe's location.

- **Goal** – Start navigation.
- **Expected Behaviour**
  1. An user is located on the cafe list screen. The targeted cafe is displayed.
  2. An user clicks on the navigation icon located in top left corner of the cafe's tile.
  3. Alternative flow: An user is located on the detail screen and taps navigation icon located to the right of address.
- **Found Issues**
  1. Three different icons used to start navigation. Icon is different on the cafe list from icon within detail screen.
  2. In the detail screen, it is not clear that icon can start navigation.
- **Proposed Solution**
  1. Use same icon everywhere.
  2. The same problem stated in the Nielsen heuristic evaluation. The icon should be highlighted. Can be added "elevation" [36] – shadow under the component to provide depth.

2.4.3.3 Test Case 3: Filter Cafes with Tags

An user is interested in cafes where they offer beside the coffee has also beer.

- **Goal** – Filter Cafes that has "beer" tag.
- **Expected Behaviour**
  1. On the cafe list screen, an user selects filter icon to open filter screen.
  2. Within filter screen, an user selects to filter by tags and choose "beer".
  3. User confirms the filter.

- **Found Issues**

  1. "Filter screen is confusing. Confirmation is useless."
  2. "Do not open filter screen, just open some modal window to selects tags".
  3. The filter icon is unclear.
  4. "If the user is located on the detail screen, clicking on the tag icon should set the filter."

- **Proposed Solution**

  1. The filter icon is standardised icon.
  2. Clicking on tag should set the filter.

2.4.3.4 Test Case 4: Suggest New Tag

An user visits regularly cafe which is friendly to dog owners. The user found out that this information is missing in the application.

- **Goal** – The user suggests tag "dog friendly" or similar for the cafe.

- **Expected Behaviour**

  1. An user selects cafe and opens detail screen.
  2. An user clicks on "suggest change".
  3. An user adds tag "dog friendly" and confirms suggestion.

- **Found Issues** – No issues found.

2.4.3.5 Test Case 5: Tag review

An user visits regularly cafe which has mistaken information about "available parking". He would like to inform about this wrong tag.

- **Goal** – The user should review "parking" with dislike.

- **Expected Behaviour**

  1. An user selects cafe and opens detail screen.
  2. An user clicks on "suggest change".
  3. An user reviews "parking" with dislikes and confirms suggestion.

- **Found Issues** – No issues found.

2.4.3.6 Test Case 6: Most popular nearby cafe

An user wants find the most popular cafe around him.

- **Goal** – The user change ordering from “by distance” to “by popularity”.
- **Expected Behaviour**
  1. An user opens filter screen.
  2. An user changes to ordering by popularity.
  3. An user confirms filter.
- **Found Issues** – No issues found.

From testers was obtained beneficial feedback which helps to tune user interface for better usability. Besides feedback from individual test cases, a few additional tips were given.

- Cafe’s name on tile can be harder to read with the contrasting background image. *Can be solved with making photo darker.*
- Landscape mode is not working. *During prototype it wasn’t considered.*
- The map marker could contain more information than cafe’s name.

All feedback was gathered and was taken into account during the next phase of implementation.

2.5 Back-end Analysis

In this section, the technical analysis of available technologies for back-end services is discussed. The focus is only on the back-end part. The technical detail of the mobile application is covered in the next Chapter 3.

One vital part of the whole architecture is Google Places API (GPA) [37]. This API offers access to Google Maps services, searching places, place details, place photos and more. The API has five services [37].

- Place search – a list of places based on the user’s location or search query.
- Place details – returns more detail information about a specific place.
- Place photos – provides access to place-related photos.
- Place autocomplete – automatically fills in the name or address of a place.
- Query autocomplete – provides a query prediction service for text-based geographic searches.

To access these services, the Google API Key must be obtained [38].

2.5.1 Google Places API

The Google Places API returns plenty of information about each place, but the application uses an only subset of available information. The application makes usage of four available services – Find a place, Nearby search, Place detail and Place photos. Some parameters are the same for each service, and the output of each service is similar. Each place has several fields such as name, address, location, rating and much more. However, not every field is used in the application, and it is not necessary to obtain these fields in the response. On top of that, each field is included in the pricing category. That means that some of the fields are more expensive to obtain than others [39]. It was important to analyse which fields are essential to get before the API model could be created. Following lines summarise each API usage, its parameters, fields and expected output.

2.5.1.1 Find Place

The purpose of this service is to find places based on the text-based query or custom location. Table 2.2 describes each used parameter. Table 2.3 describes each expected field in the response.

Parameter	**Usage**
key	API key
input	search query
inputtype	set to textquery
language	language code
locationbias	set to circle:radius@lat,lng.

Table 2.2: Find Place Parameters.

2.5.1.2 Nearby Search

Nearby Search allows finding nearby places around the given location. In contrast with *Find place*, the *Nearby Search* returns all fields available to place and can return up to 60 places. The result is paginated up to three pages with 20 places in each page. If the result has another page, the pagetoken is included in the response. As before Table 2.4 shows service parameters.

2.5.1.3 Place Details

Each place has additional information accessible through place details. Table 2.5 describes each parameter used, and Table 2.3 describes fields which should be returned in the response.

Field	**Description**
place_id	unique place identification
name	place's name
icon	place's icon
geometry	geolocated latitude,longitude
formatted_address	human-readable address
types	list of place categories such as "cafe"
photos	contains photo width, height and photo_reference
opening_hours	contains only open_now. For full information, detail request is required
price_level	value from 0 to 4
rating	value from 1.0 to 5.0

Table 2.3: Find Place Fields.

Parameter	**Usage**
key	API key
location	latitude, longtitude
radius	circular radius in meters, max 50 000 m
language	language code
opennow	place is currently open; if place does not have this field, it is omitted from results
type	place type, always set to *cafe*
pagetoken	token for next page

Table 2.4: Nearby Search Parameters.

Parameter	**Usage**
key	API key
place_id	place's id
language	language code

Table 2.5: Place Details Parameters.

Parameter	**Usage**
international_phone_number	phone number in international format
opening_hours	opening times and if is currently open
photos	additional up to ten photos
review	up to five user reviews
utc_offset	offset in minutes from UTC
website	place's website

Table 2.6: Place Details Fields.

Note that *opening_hours* contains localised *weekday_text*. If a place is always open, the *close* field within *opening_hours* is missing and *open* field has day equal to zero and time equal to “0000”.

2.5.1.4 The API Response Format

The GPA can respond with *JSON* or *XML* format. The *JSON* format was chosen as a more typically used and standard format for REST API and mobile communication [40].

```
{
  "html_attributions": [],
   "results": [
      {
         "business_status": "OPERATIONAL",
         "id": "0c19db184d6705e945686f3d83ae02a3fbf23068",
         "name": "JS Café",
         "opening_hours": {
            "open_now": true
         },
      }
   ],
   "status": "OK"
}
```

Listing 18: Nearby Search Example Output (Shortened).

The *Find Place*, *Nearby Search* and *Place Details* has similar response format where `<status>` indicates response code because GPA for each request always responds with HTTP status code 200 OK. The status field can have several values such as `OK` for successful request, `ZERO_RESULTS` for empty results or `INVALID_REQUEST` for general invalid request. Each service has different result field name – `candidates` for *Find Place*, `results` for *Nearby Search* and `result` for *Place Details*. An example output is shown in Listing 18.

2.5.1.5 Place Photos

Each place can have photos. The response from *Find Place* or *Nearby Search* contains `photo_reference` along with width and height of the photo. For obtaining actual photo itself, the *Place Photos* endpoint must be called. The endpoint accepts as always the `key` (API key) and `photo_reference`. The result can be altered with parameters for maximal width or height. Returns byte-encoded image if the request was successful otherwise HTTP status code *400 Bad Request* or *403 Forbidden* if the quota limit was reached.

Although Google Places API returns plenty of information about each place, one added benefit of the Coffee Time application is tags suggested bu the users. This requirement cannot be fulfilled only with Places API, so there is a requirement for another, custom, API.

2.5.2 Coffee Time API

The Coffee Time API (CTA) will be used for two main functionalities. First of all, it will provide access to Google Places API and secondly, it will be used for getting assigned tags for cafes. While it is possible to use GPA directly from the mobile application (client), it would lead to the need for another request to obtain the cafe's tags and another possible required custom information. Hence, if custom API is used as a proxy to GPA, with one request to this API, the response can be modified with additional data and returned at once. Figure 2.10 shows such a communication.

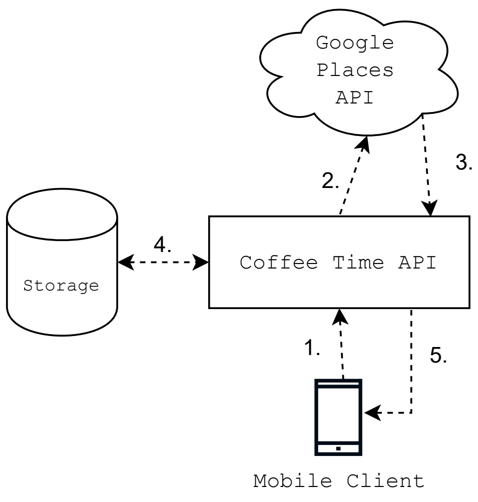


Figure 2.10: Client and API Communication Flow.

1. HTTP request made by client to Coffee Time API.
2. The request is processed and redirected to Google Places API.
3. The response from Google Places API is obtained.
4. If appropriate the query to storage is issued, and response is enhanced with obtained data.
5. The final result is returned to the client.

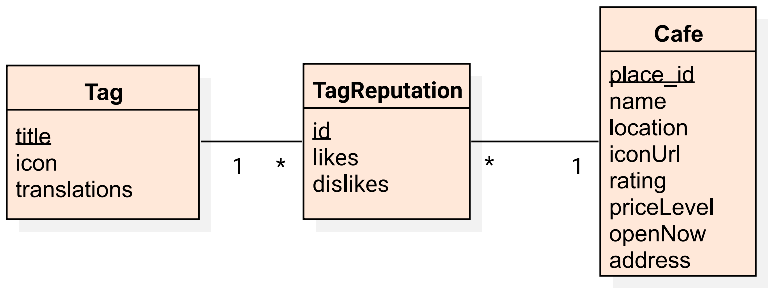


Figure 2.11: Cafe's Tag and Cafe Relation.

2.5.2.1 The Cafe Tags

The set of available tags is predefined, with the requirement to be able to change it any time without changes within the client or in the API. Each tag has a title and an icon and can be translated into another language. This definition and requirement for no changes within the client imply that used icon has to be part of tag's definition in the CTA. To sum up, each tag is an entity with a unique title, icon and translation.

As was said earlier in this chapter, each cafe can have assigned tags. Each assigned tag is reviewed by users with the concept of "likes" and "dislikes". This requirement implies that for each cafe is needed to be known assigned tags and their reviews. This relation is shown in Figure 2.11.

2.5.2.2 Domain Model

Within detail screen, additional information is displayed, which can be obtained from the Place Details call. This concludes that the application's cafe domain entity is split into two entities - Cafe and Cafe Detail entities. The domain entity graph is listed in Figure 2.12 showing all entities related to CTA response.

2.5.3 The Selection of the Right Technology

The technology which will be used to build the API should fulfil the following requirements:

- The access to API should be secured,
- The back-end service should be simply scaled up when needed,
- The maintenance cost should be kept low.

There are many technologies which can be used to achieve this requirements. One of the ways is implementing API from the ground with technologies such as *.NET Core* and its *ASP .NET Core API*. Then the API can

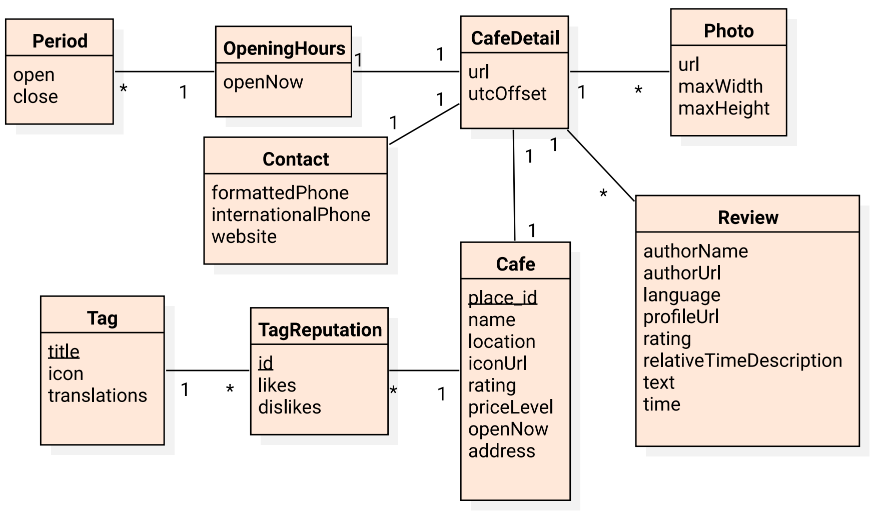


Figure 2.12: Coffee Time Domain Model.

be deployed to service hosting providers such as *Microsoft Azure* or through technologies such as *Docker* containers and *Kubernetes* [41] as container orchestration.

Another way is to use serverless services. The serverless is approach where “The cloud provider is responsible for executing a piece of code by dynamically allocating the resources. And only charging for the amount of resources used to run the code. The code is typically run inside stateless containers that can be triggered by a variety of events including http requests, database events, queuing services, monitoring alerts, file uploads, scheduled events (cron jobs), etc.” [42]. This approach is sometimes called as *Functions as a Service* (FaaS in short). The major providers are *Amazon Web Services* and its *Lambda Functions*, *Microsoft Azure Functions* or *Cloud Functions (Firebase Functions)* by *Google.*

The *Firebase* by *Google* is collection of services which helps to build mobile application faster, improve application quality and grow business. The *Firebase* offer services as real-time NoSQL database, application usage analytics, A/B user testing, cloud web hosting or *Cloud Functions* and much more. Within one service, the storage for data, the API as functions and security can be made.

2.5.3.1 Cloud Firestore

Cloud Firestore is a flexible, scalable database for mobile, web, and server development. It keeps data in sync across client apps through realtime listeners and offers offline support for mobile and web. The *Cloud Firestore* uses NoSQL data model. The data are stored in documents that contain fields mapping to values. These documents are stored in collections, which are containers for documents that are used to organize data. Documents support many different data types, from simple strings and numbers, to complex, nested objects. Documents can include subcollections and build hierarchical data structures that scale as database grows [43].

2.5.3.2 Cloud Functions

Cloud Functions for *Firebase* is a serverless framework that lets automatically run backend code in response to events triggered by *Firebase* and HTTPS requests. After function is deployed, Google's servers begin to manage the function immediately. The function can be used directly with an HTTP request. Each function runs in isolation, in its own environment with its own configuration and are scaled accordingly to usage load [44].

2.5.3.3 Firebase Authentication

Firebase Auth offers service to secure application with *OAuth 2.0* standard [45]. It automatically integrates with identity providers such as *Google*, *Microsoft*, *Facebook* and more. Besides that it offers standard authentication through user's email or phone [46] and anonymous authentication where for each user randomly generated identifier is used. The service guarantees that for each user, as long they use the same device, the identifier stays the same.

The *Cloud Functions* can be used to build Coffee Time API. The *Cloud Firestore* as a database (storage) and API can be secured through *Firebase Authentication*. On top of that, *Firebase* offers services for analysis, logging and user testing. The *Firebase* has "pay as you go" model and offers a free plan (Spark plan). The Spark plan offers up to 1 GiB data storage, 20 thousands of writes and 50 thousand of document reads per day. The *Cloud Functions* can be invoked 125 thousand times per month. This pricing perfectly suits the requirements and can be used freely to build Coffee Time application. If the application becomes popular and frequently used, the actual price of *Firebase* services is calculated by current usage and services can be scaled up accordingly. How the *Firebase* services are used in detail is more described in Chapter 3.

2.6 The Conclusion

In this chapter, the considered application Coffee Time was introduced.
The analysis of existing alternatives was made to obtain inspiration for how each application behaves. The prototype was created to propose user interface and tested it with users. In the end, the analysis of Coffee Time API was made along with analysis of chosen technology to build this API.

Chapter 3

Implementation

When the Coffee Time specification was created and the prototype was tested, the implementation itself could begin. In this chapter, the implementation details are covered. In the beginning, the proposed architecture of the application is introduced and explained. Next section describes how the Coffee Time API is designed and implemented. This section also covers which framework for creating REST API was chosen and how it can be connected to *Firebase* services. Detailed section of the mobile application implementation follows. In this section, used techniques and packages are described to achieve desired functionality.

3.1 Clean Architecture

In order to have easy to write, maintainable and testable code, some overall architecture should be considered. The architecture should follow these rules:

1. Independence of UI from application business logic.
2. The business logic has to be testable without any dependent requirements such as server connections.
3. Interchangeable UI. The UI part can be changed without affecting the rest of the system.
4. Independence of business logic from the infrastructure – the data source can be changed without affecting business logic.

There are several attempts and approaches on how to design such architecture [47]. The inspiration for chosen architecture comes from *Clean Architecture*, initially proposed by *Bob C. Martin* [48]. In the basic form the *Clean Architecture* (fig. 3.1) consists of four layers organised as circles. Architecture defines dependency rule where each layer should be dependent only on

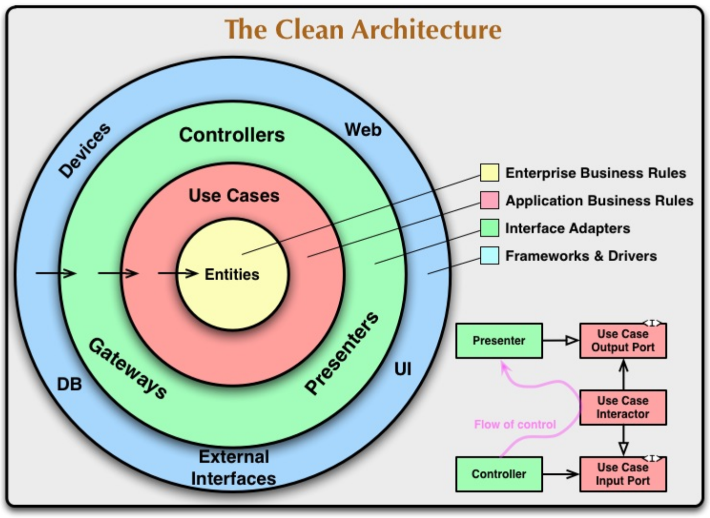


Figure 3.1: Clean Architecture Defined By *B.C. Martin* [47].

the inner circles. Accordingly to *Bob C. Martin* [47], layers can be described as:

Entities encapsulate Enterprise-wide business rules. These entities, which can be simple objects or hierarchical structure of objects are the least likely to change when something external changes.

Use Cases are application-specific business rules. They encapsulate all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities and direct those entities to use their enterprise-wide business rules to achieve the goals of the use case. Changes within the use case do not affect entities and similarly, changes of outer layers (such as interchanging database provider) do not affect the use cases.

Interface Adapters converts data from the format most convenient for the use cases and entities to the format most convenient for some external agency such as database or API. The application presentation (UI) should be here. Data models in this layer most likely consist only of data that are passed down to the use cases and then back from use cases to the view. This layer is also responsible for converting any data from external forms into an internal form used by the use cases.

Frameworks and Driver contain any external frameworks and tools. This layer typically exposes interfaces to glue the external framework wit the rest of the application.

This architecture was chosen for its simplicity of dependencies, easy testability and extendability.

3.2 Coffee Time API

The Coffee Time API is served through *Firebase Cloud Functions* services. Each function can run the backend code in response to events triggered by *Firebase* features and HTTPS requests. Functions can be written with *JavaScript* or *Typescript* language. For implementation, the *Typescript* was chosen due to the safer typing system, and due to the fact, that author has more experience with *Typescript* than with plain *JavaScript*.

Functions are written under *Node.js* [49], and every function that is exported is considered as "Cloud function" and can be accessed through HTTPS request. This can be used to build fully functional REST API. For the implementation, *Express.js* framework [50] was selected as popular and easy to use solution. Developers define routes, which HTTP method can be used and how the request should be processed. *Express.js* has a simple yet powerful mechanism of middlewares, where each request can be pre-processed before sending to the next processing. For example, it can be used to check if the user is authorised to make the request. With this setup, nothing prevented to create Coffee Time API. Furthermore, as *Express.js* allows to define REST API, only one cloud function can be exported – "the API" function. All requests are processed through this function. However, before implementation itself, it had to be defined what API will offer and how it can be consumed.

3.2.1 The API Endpoints

The API has endpoints divided into three categories – places, tags and photos. As a client is considered to be multilingual, the API is too. The Places API accepts, as was described earlier, language parameter to specify how the response should be localised. The API is designed with language parameter as a mandatory part of the URL for places related requests. The tags category has endpoints for obtaining and manipulating tags. Lastly, photo category gives access to downloading the place's photo. Every response coming from GPA which returns places are modified such that each "place" entry contains additional `tags` field which is an array of the `tag` model.

Table 3.1 describes each API endpoint and its usage. Mandatory parameters within the URL are highlighted as `<param>`. Each URL is shown without prefix `http[s]://domain/api` (the hosting URL and `/api` suffix).

Table 3.2 describes *nearby* parameter. Note that, if radius is omitted, the results are sorted by Google's ranking – "Ranking will favor prominent places within the specified area. Prominence can be affected by a place's ranking in Google's index, global popularity, and other factors" [51] otherwise the results are sorted by distance.

URL	**HTTP Method**	**Description**
`<language>/nearby?parameters`	GET	Get nearby cafes
`<language>/find?parameters`	GET	Find cafes within area based on text query
`<language>/detail/<place_id>`	GET	Place's details of given `place_id`
`<language>/basic/<place_id>`	GET	Returns basic place information

Table 3.1: Places Endpoints.

Parameter	**Required**	**Description**
`location=lat,lng`	Yes	Location to search, where `lat`=latitude, `lng`=longitude
`radius`	No	If omitted, the results is sorted by *prominence* (see below), otherwise by distance
`opennow`	No	If present returns only opened cafes
`pagetoken`	No	If present, next page returned

Table 3.2: Nearby Parameters.

Parameter	**Required**	**Description**
`input`	Yes	Search query to use
`location=lat,lng`	No	Search within circular area defined by radius
`radius`	No	Circular radius in meters

Table 3.3: Find Parameters.

Table 3.3 describes *find* parameters. The `location=lan,lng` and `radius` must be provided both if used.

Table 3.4 describes all endpoints related to *tags*. The POST request to `/tags/<place_id>` should have a body with array of *TagUpdate* model. The example is listed in listing 19 where `id` contains tag id and `change` contains value of `like` or `dislike` which corresponds with increasing (decreasing) tag's score.

3.2.2 Express.js Pipeline

Express.js framework has a powerful and configurable way how to process incoming request. Every request can be processed through functions, called

URL	HTTP Method	Description
`/tags`	GET	Get all defined tags
`/tags/<place_id>`	GET	Get all assigned tags to given `place_id`
`/tags/<place_id>`	POST	Updates tag's review for given `place_id`

Table 3.4: Tags Endpoints.

```
[
    {
        "id": "tag",
        "change": "like|dislike"
    }
]
```

Listing 19: Tag Update Content Example.

middlewares, before sent to a router. The router is a set of functions which maps part of URL with HTTP method to trigger response to the incoming request. Together it creates a flexible way of developing any form of a web-server or in case of Coffee Time API a REST API.

Middleware functions can execute any code, make changes to the request (or to the response), end the request-response cycle or call next middleware [52]. The function accepts three parameters – `req`, `res` and `next`. `Req` is a request object containing all information related to the incoming request. The `res` is a response object which can be used to alter the response, and `next` is the callback function to trigger next middleware. In case of the Coffee Time API, three middlewares are used to process every incoming request.

First middleware checks and parses request body (if any provided) in JSON format. JSON format was chosen as a primary format for communication with Google Places API, the CTA forces JSON format to every request which contains the body. Next middleware logs each incoming request to the cloud functions. Each log message contains issued HTTP method, requested URL and sent body if present. Last middleware authorises incoming request against Firebase Authentication Service.

3.2.2.1 Authorisation Middleware

Each request to the API has to be authorised against Firebase Authentication services. The authorisation process uses a JSON Web Token (JWT) open standard [53]. Each request has to include Authorisation header with "Bearer token" obtained earlier from *Firebase Authentication.*

If authorisation header is missing, the request is ended, and response with status code *HTTP 403 Forbidden* is sent back. In case the token is provided, it is checked against Firebase Authentication service. If the token is valid, the user information is obtained, and the request is sent to next middleware. If token validation failed, response with status code *HTTP 401 Unauthorized* is sent. The implementation of this middleware is shown at Listing 20.

```
app.use(async (req, res, next) => {
 if (!req.headers.authorization ||
  !req.headers.authorization.startsWith('Bearer ')) {
  res.status(403).send('Unauthorized - No token provided');
  return;
 }
 const idToken = req.headers.authorization.split('Bearer ')[1];
 try {
  const decodedIdToken = await admin.auth().verifyIdToken(idToken);
  req.user = decodedIdToken;
  next();
 } catch (e) {
  res.status(401).send('Unauthorized - Invalid token');
 }
});
```

Listing 20: Authorisation Middleware.

3.2.3 Routing

When every middleware processed a request, the routing takes its place. Routing defines which function should be called to a particular URL along with the HTTP method. For example

```
app.get('/', (req,res) => res.send('Hello world'));
```

will respond with "Hello World" to every GET request made to the homepage. The router mechanism supports dynamic parameters – part of URL can be dynamically matched to parameters. Moreover, each route can be divided to sub-routers. This helps for code readability.

The dynamic parts are used for `<language>` parameter within *places* endpoints and for `<id>` whenever tag id, place id or photo id (reference) is needed. CTA uses three sub-routers. One for *places* related requests, second for *tag* related requests and third for all *photo* related requests.

3.2.3.1 Routing Implementation

The `index.ts` file contains initialisation of API, the definition of all middlewares and routing setup (Listing 21). For example, the *places* router uses a dynamic parameter `<language>` which is parsed beforehand and added as part of the request object.

```
// initialize app
app = express();
// ... configuration

// middlewares - skipped

// add language parameter to request
app.param('language', (req, res, next, value) => {
  req.language = value;
  next();
});
// Find places routes
app.use('/:language', placesRoute(tagsRepository));
// Tags routes
app.use('/tags', tagsRoute(tagsRepository));
// Photo
app.use('/photo', photosRoute());
// export api as cloud function
export const api = firebase.region('europe-west1')
                           .https.onRequest(app);
```

Listing 21: API Definition.

3.2.4 Integration with Firestore

In order to have access to *Firestore* storage and to authorise tokens, a *Firebase Admin SDK* is used to communicate with *Firebase* services. To be able to use SDK, a `google-services.json` file has to be provided. This file contains all required information to have access to the *Firebase*. Note that, this file contains sensitive information and should not be committed to the version control. Figure 3.2 shows how the request is processed and the relation between *Express.js* modules with *Firebase Admin SDK*.

3.2.5 Functions Deployment

A developer can run functions locally and they can be served through localhost while he or she developing them. Once the code is ready, it has to be deployed

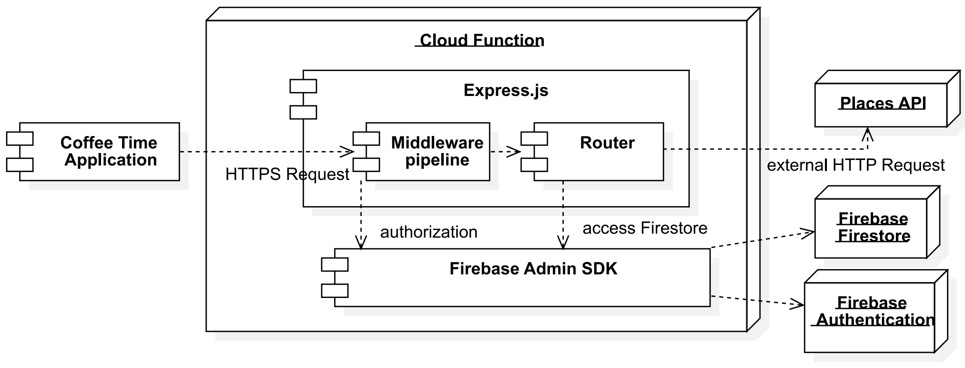


Figure 3.2: API Components.

to the *Firebase Cloud Functions* service. In order to do that, *Firebase CLI* tooling and its deploy command have to be used. If any version exists already in the *Firebase*, it is replaced by a new one.

Firebase Cloud Functions can be deployed to different physical areas such as *West Europe* or *US East*. While the Coffee Time application is mainly tested in the Czech Republic, *West Europe* was selected as the best option because of its closest availability. One downside is that each location is part of the API URL. This can add complexity to the client side if more locations are used in case of globally available application.

3.3 Coffee Time

For Coffee Time purposes, the architecture was slightly simplified and is made from three layers – domain, data and presentation. The domain layer encapsulates all domain entities and defines contracts for repositories. This layer corresponds with "Entity" layer from Clean Architecture. Next, the Data layer encapsulates all external communication such as API requests or location services. This layer also defines data models which are passed down to the repositories where data are processed and transformed to domain entities. Data layer corresponds with *Interface Adapters*. The presentation layer is responsible for describing the user interface and interacts with BLoC objects where the business logic is encapsulated.

This section describes each layer – its implementation details and how it is used with other layers.

3.3.1 Domain Layer

Domain Layer defines every entity used by the application, definition of repository contracts – repository interface definitions and application-wide exceptions. Repository implementation resides in Data layer.

One of important requirements of the Presentation layer and overall Flutter's philosophy is immutability. Hence, the domain entities are designed as immutable objects. Another requirement is object equality. By default, Dart compare objects by identity – two objects are same instance, they are considered as identical (equal). In order to have fully functional application, it is necessary to override default equality behaviour to comparison by object properties.

3.3.1.1 Equatable Package

If the default equality behaviour has to be changed, the `==` operator and `hashCode` getter have to be overridden [54]. Even with small classes, this work becomes quickly tedious and can leads to unnecessary bugs for example when some property is added and developer forgot to update these overrides. Therefore in case of complex classes the amount of bugs only increases.

Equatable package [55] (developed by *F. Angelov*) was created to simplify creating immutable classes with proper equality implementation. Class which needs to implement equality has to extend `Equatable` base class provided by the package. Then it is needed to override `props` getter where all class properties are listed. With that setup, equatable package internally provides proper equality comparison. This package is used to for all entities and classes which needed proper equality implementation. An example of its usage is shown in Listing 22, which is `OpeningHours` entity implementation.

3.3.1.2 Repositories Contracts

Domain Layer also defines repository contracts. The repository is used to as connection between presentation layer and data layer. Coffee Time application defines two repositories – `CafeRepository` and `TagRepository`. Within domain layer, repositories are defined as abstract classes which define required interface. The implementation itself, as was said earlier, is in the data layer. The `CafeRepository` (Figure 3.3) has methods related to cafe places such as finding nearby cafes or user's favorited cafes.

`TagRepository` (Figure 3.4) has methods related to tags such as getting all available tags or tags assigned to specific cafe.

In order to have convenient approach of handling exceptions raised within repository (or from layers above) in repository layer, each method of repository returns type `Either<L,R>`. Either is functional concept [56] of returning two distinct values from one method. It is used to return desired return type in case of success or error in case of exception. This approach helps to avoid `try - catch` clauses in presentation layer and the code is more readable in functional way.

```
class OpeningHours extends Equatable {
  final bool openNow;
  final List<Period> periods;
  final List<String> weekdayText;

  OpeningHours({this.openNow,
              this.periods, this.weekdayText});

  OpeningHours copyWith(
      {bool openNow,
       List<Period> periods,
       List<String> weekdayText}) {
         return OpeningHours(
          openNow: openNow ?? this.openNow,
          periods: periods ?? this.periods,
          weekdayText: weekdayText ?? this.weekdayText);
  }
  @override
  List<Object> get props => [openNow, periods, weekdayText];
}
```

Listing 22: OpeningHour Entity with Equality.

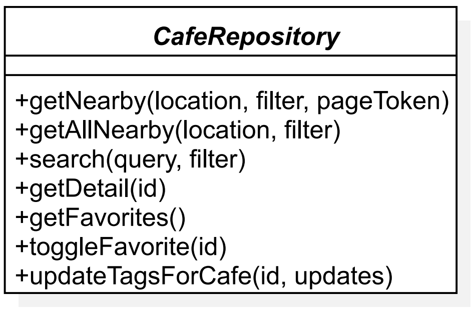

CafeRepository
+getNearby(location, filter, pageToken)
+getAllNearby(location, filter)
+search(query, filter)
+getDetail(id)
+getFavorites()
+toggleFavorite(id)
+updateTagsForCafe(id, updates)

Figure 3.3: Cafe Repository Interface.

The `Either<L,R>` is generic class, where `L` is left type and `R` is right type. The returned value from such method can be used with `when()` or `map()` methods to access actual returned value. Internally the `Either<L,R>` is implemented with *freezed* package [57]. This package and its usage is more precisely described later, in the presentation layer. In this way, Either is strongly typed. Similar approach could be use for example `Tuple<T1,T2>` type, but losing strongly typed solution.

Listing 23 shows how `Either<L,R>` is used within `TagRepository`. In case

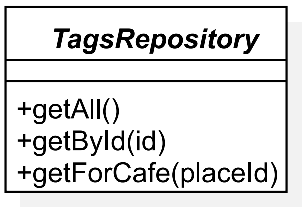


Figure 3.4: Tag Repository Interface.

```
@override
Future<Either<List<Tag>, Failure>> getAll() async {
    try {
      final tags = await _getAllTags();
      return Left(tags); // Return Left type - List<Tag>
    } on ApiException catch (e) {
      // Return Right type - Failure
      return Right(ServiceFailure(errorMsg, inner: e));
    } catch (e) {
      // Return Right type - Failure
      return Right(CommonFailure(e));
    }
}
```

Listing 23: Returning Value as Either within Tag Repository.

of successful operation, `Left` type is returned, in this case the list of tags. If any exception occurs, the `Right` type – Failure is returned instead. The usage of such returned value is shown in Listing 24, where either a `List<Tag>` is returned or a `Failure` object instead.

```
// ... call to TagRepository
final result = await repository.getAll();
final allTags = result.when(
      // successful call - get tags
      left: (tags) => tags,
      // failure - return empty array in this case
      // ... and for example log the failure to logger
      right: (failure) => <Tag>[],
);
```

Listing 24: Usage of Either Returned Value.

3.3.2 Data Layer

Purpose of layer service is to implement repositories, add services which communicates with the Coffee Time API and define a models which are mapped to the responses from API. As with the repositories, the services are implemented against interfaces. This approach helps to write testable code. Repositories which use these services depend on interfaces and not on the concrete implementations. Figure 3.5 shows association between repositories and those services.

Each service communicates with a concrete part of the Coffee Time API. The *CafeService* has methods related to obtaining nearby cafes or getting cafe detail, *FavoriteService* has methods to obtain favorited cafes, *PhotoService* gives access to downloading cafe's photos and *TagService* serves for manipulation with tags.

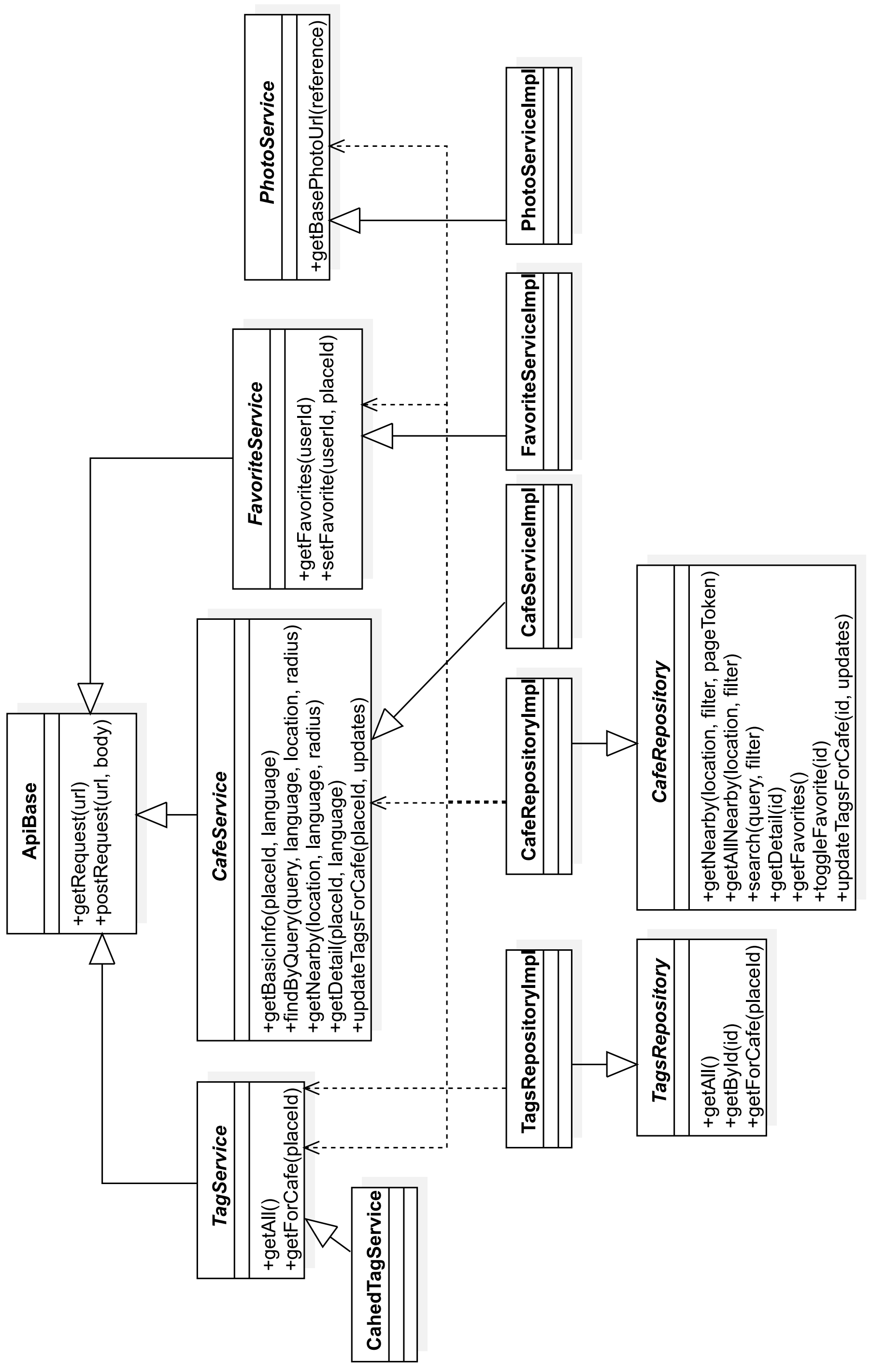


Figure 3.5: Repositories and Services Association.

3.3.2.1 Favorite Local Service

Current application version uses for saving user's favorite cafes only local storage. Thanks to implementation against interface, in the future, the synchronization of favorited cafes over API can be added without breaking other parts of the application.

3.3.2.2 Cached Tag Service

To prevent unnecessary amount of calls to API when obtaining all tags, a caching mechanism was implemented for *Tag Service*. When all tags are returned from the API, they are cached for future calls. As a whole implementation is focused on clean code implementation, maintainability and testability, a generic `CachedValue<T>` class was implemented as wrapper around cachable values.

```
typedef ExpirationCallBack<T> = Future<T> Function();

class CachedValue<T> {
  T _value;
  DateTime _timeStored;
  final Duration durability;
  final ExpirationCallBack<T> onExpire;
  final TimeProvider timeProvider;
  // ... constructor ommitted
  Future<T> get() async {
    if (_value == null ||
        timeProvider.now()
                    .difference(_timeStored)
                    .inMilliseconds
            > durability.inMilliseconds) {
      _value = await onExpire();
      _timeStored = timeProvider.now();
    }
    return _value;
  }
}
```

Listing 25: Cached Value.

The `CachedValue` accepts `TimeProvider` which provides current time, duration for how long value should be cached and `ExpirationCallback<T>` which is called whenever cached value has to be re-assigned. `TimeProvider` class is abstraction of `DateTime.now()`. This abstraction was necessary for unit testing.

3.3.2.3 Entity Models

Every response from API which contains data is mapped to *model*. Model is a immutable class, similar to *entity* classes except that model includes parsing from (and to) JSON format. These models are returned by services to repositories, where models are transformed into entities and returned further. Subset of models correspond with entities in one to one relationship, some other models have logic for mapping its values to the corresponding entity.

3.3.3 Representation Layer

The responsibility of representation layer is to define user interface, connect it with a BLoC objects which communicate with repositories. How to properly organize and architect user interface within Flutter, mainly how to separate responsibilities between BLoC classes is still very subjective opinion and there is no "best practice" yet. However, commonly used approach is divide BLoC responsibility per "big enough feature". That means, identify feature of the application and separate BLoC responsibilities appropriately. Another approach is "BLoC per screen". This approach is well suited for smaller application. As the Coffee Time is still relatively small application, second approach was chosen for its simplicity. Each screen has its own, separated BLoC class and each BLoC define states for given screen and set of events which can accept.

```
│ core ........................................ shared BLoCs
└─ screens
   └─ cafe_list ............................... Cafe List Screen
      └─ bloc ................................. Cafe List's BLoC
         └─ bloc.dart ......................... BLoC implementation
         └─ state.dart ........................ BLoC's states
         └─ event.dart ........................ BLoC's events
      └─ widgets .............................. Specific screen widgets
      └─ screen.dart .......................... Main screen widget
   └─ detail .................................. Next screen
   └─ ..
└─ shared ..................................... Shared widgets and theming
└─ app.dart ................................... Main application widget
```

Figure 3.6: Presentation Layer Screens Organization.

Figure 3.6 shows overall presentation code organisation. Each screen has defined entry file `screen.dart` where screen's widget is defined. Widgets folder contains all screen specific widgets. Bloc folder contains BLoC implementation along with states and events. Shared folder contains shared widget and application theming related settings. This organization helps to keep organized code with strictly defined where particular BLoC can be found for given screen.

3.3.3.1 Flutter_bloc Package

As was pointed out in the first chapter, *flutter_bloc* is a package developed to help implement a BLoC pattern with a simplified manner. Each BLoC class has to extend `BLoC<TEvent,TState>` class. Its simple usage was shown in Listing 13 in the first chapter. Each BLoC class has to override `mapEventToState(TEvent)` method. In this method, BLoC should decide which state to yield based on the event. When state is yielded, it is compared to current BLoC's state and if they are different, widgets listening for changes are rebuilt. This implies important requirement for state class – it has to have overridden equality comparison.

To connect BLoC with other widgets, *flutter_bloc* offer several widgets to help that:

- **BlocProvider** – Provides BLoC instance down to the tree. Internally it is using *Provider* package.
- **BlocBuilder<Bloc, State>** – Listen for changes from given *Bloc* and its state, which has to be provided somewhere in the parent widgets. It has `builder` callback, where a new widget based on obtained state is returned.
- **BlocListener<Bloc, State>** – In order to do side effects, such as display notification, *BlocListener* can be used. As *BlocBuilder* it is listening for changes from Bloc and its new state.
- **BlocConsumer<Bloc,State>** – Combines functionality of *BlocBuilder* and *BlocListener* together.

3.3.3.2 State and Events Implementation

Each BLoC has typically more than one state and event. To do that, the simplest approach is to define abstract class for state (event) and every specific state (event) extends this abstract class. This works well, however in the `mapEventToState()` method with this approach has to be distinguished between specific types. Only one possible option is to use `is` operator. Such example is shown in Listing 26.

This approach has several issues, mainly:

- Repetitive code with “event is” and,
- no compile time control. When new event is added, there is no way to force add new “else if” branch. Options such as default else with throwing exception is not viable solution, as it does not add any benefit.

```
Stream<CafeListState> mapEventToState(CafeListEvent event) async* {
    if (event is LoadNext) {
      yield* _mapLoadNext(event);
    } else if (event is Refresh) {
      yield* _mapRefresh(event);
    } else if (event is SetFavorite) {
      yield* _mapSetFavorite(event);
    } else if (event is UpdateTags) {
      yield* _mapUpdateTags(event);
}
```

Listing 26: Abstract Class Approach – State Mapping to Events.

Same issue is with mapping states within widgets. There is better solution with “union types” and functional way of “map” method.

3.3.3.3 Union Types to the Rescue

Union is an object containing value of different types, but allows manipulating that value with type-safety and compile time check. In fact, earlier introduced `Either<L,R>` is a such union type. Package *Freezed* [57], by the same author of the *Provider*, helps to do such implementation.

One popular functionality within *Dart* ecosystem is code-generation and *Freezed* package is no difference. In order to implement union type, developer has to define abstract class with all possible sub-values and package will generate rest of the code. The generated class is immutable and has:

- overridden equality operators,
- `copyWith()` method to copy object with altered properties,
- `map()` method to map from one possible sub-class,
- `when()` method to map from one possible value. Difference with `map()` is that `when()` returns values of the sub-class, but `map()` returns sub-class instance,
- and has `maybeMap()` and `maybeWhen()` which do not force include all possible values.

Union type forces compile-time check in case of `map()` usage. This eliminate issue with `is` operator. Listing 27 shows *CafeListEvent* definition as a union type. In Listing 28 is shown same events mapping to state but with union approach. Whenever new event is added, the compile-time error will occur as the `map()` forces it.

Similarly state mapping within CafeList Screen (Listing 29) is simplified and compile-time safe.

```
@freezed
abstract class CafeListEvent with _$CafeListEvent {
  const factory CafeListEvent.loadNext(
      {String pageToken,
       @Default(Filter()) Filter filter}) = LoadNext;
  const factory CafeListEvent.refresh(
          {@Default(Filter()) Filter filter}) = Refresh;
  const factory CafeListEvent.setFavorite(
      {@required String cafeId,
       @required bool isFavorite}) = SetFavorite;
  const factory CafeListEvent.updateTags(
      {@required String cafeId,
      @required List<TagReputation> tags}) = UpdateTags;
}
```

Listing 27: Union Class Approach – CafeList Event Definition.

```
@override
Stream<CafeListState> mapEventToState(CafeListEvent event) async* {
    yield* event.map(
      loadNext: _mapLoadNext,
      refresh: _mapRefresh,
      setFavorite: _mapSetFavorite,
      updateTags: _mapUpdateTags,
    );
}
```

Listing 28: Union Class Approach – Events Mapping To State.

3.3.3.4 BLoC to BLoC Communication

Each screen has its own assigned BLoC, however some communication between BLoCs is also needed. For example in case of updating favorite cafe, the *FavoriteBloc* is updated. The *CafeListBloc* and *DetailBloc* listen to such changes and updates appropriately. Figure 3.7 shows associations between screens, blocs and, bloc to bloc communication.

3.4 SOLID and Dependency Injection

One important part of maintainable and testable code are SOLID principles, originally designed by *Bob C. Martin* [58]. Each letter stands for one principle:

1. *Single Responsibility Principle* – a class should only have on responsibility. That class should have one reason to change.

```
return BlocBuilder<CafeListBloc, CafeListState>(
  builder: (context, state) {
    return state.map(
      loading: (_) => const CircularLoader(),
      loaded: (loaded) {
        if (loaded.cafes.length == 0) {
          return const NoData();
        }
        return CafeList(state: loaded);
      },
      failure: (failure) => FailureContainer(
        message: failure.message,
        onRefresh: () => context
            .bloc<CafeListBloc>()
            .add(Refresh(filter: failure.filter)),
      ),
      // shortened for brevity
    );
  },
);
```

Listing 29: CafeListState Mapping within CafeList Screen Build Method.

2. O*pen Closed Principle* – a class should be open for extension, but closed for modification.
3. *Liskov Substitution Principle* – if class A is a subtype of class B, then it should be possible to replace B with A without disrupting the behaviour of the program.
4. *Interface Segregation* – many specific interfaces are better than one general-purpose interface. In other words, split larger interfaces to smaller ones.
5. *Dependency Inversion* – high-level modules should not depend on low-level modules. Both should depend on abstractions.

Dependency Injection mechanism is used to provide concrete implementations of each required abstraction to classes which depends on it.

Dart (and Flutter) does not provide such Dependency Injection mechanism out of the box. However, *get_it* is a package, which helps with that. In fact, *get_it* works more as Service Locator. Every concrete implementation has to be registered and paired with its abstraction counterpart within *GetIt* container. When those implementation are registered, container can be accessed through singleton instance and obtain concrete implementation.

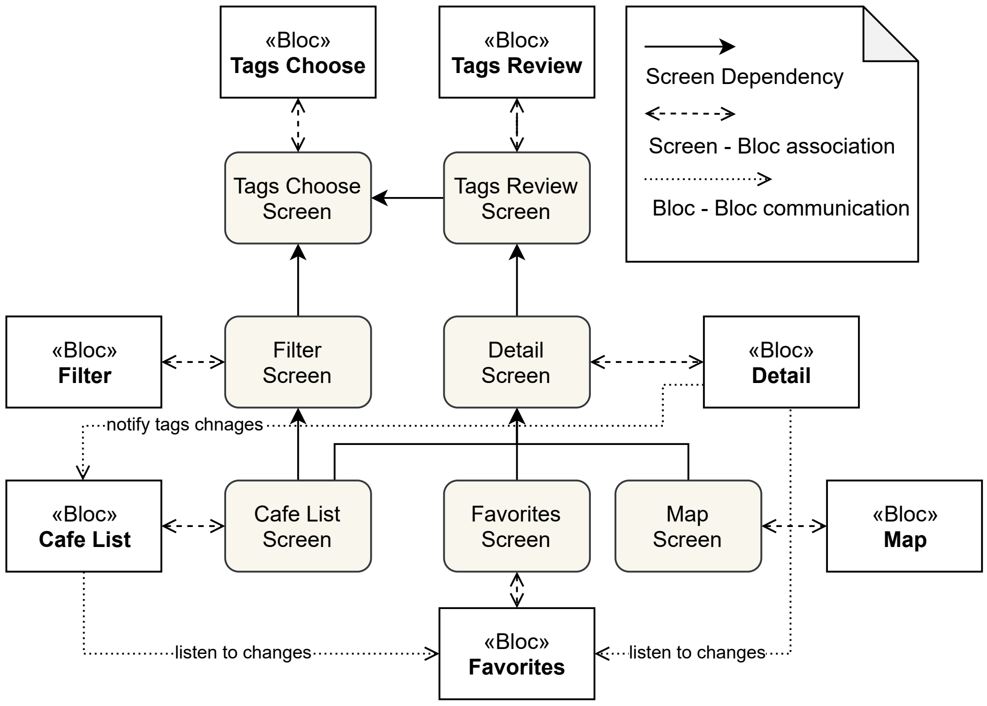


Figure 3.7: Screens and Blocs Association.

However, such Service Locator is usually considered as anti-pattern if used careless. If such container is accessed everywhere, it can hides dependencies and it is highly coupled to every module. On the other hand, if each module lists its dependencies within constructor, such modules can be constructed at one place. In this place, usually called "composition root", this issue can be avoided. Thus, in the Coffee Time implementation, every dependency is registered within *GetIt* container (see `di_container.dart` file). Then concrete implementations are created through *GetIt* locator mainly in `App.dart` or when doing navigation between screens.

3.5 Unit Testing

Properly written project should have tested code. Coffee Time application has written unit tests for each layer. Domain entities has unit tests for their logic, service has unit tests for proper handling of API communication, repositories has tests for business logic and, BLoCs are tested that emits proper state based on received events.

In order to have independent, isolated unit tests, some class dependencies are mocked to provides desired behaviour. To create mocks, a *mockito* package [59] is used. Package offers flexible of mocking parts of the class. Developers can define what mocked method for given parameters should respond. Another package, *bloc_test* [60] (same author of *Equatable* and *flutter_bloc* packages) is used to test implementation of BLoCs.

An example of unit test is shown in Listing 30. Unit tests use both mentioned packages. `FilterBloc` has dependency on `TagRepository` which is mocked in order to return faked values. Method `blocTest()` comes from second packages and offer convenient way for BLoC testing.

3.6 Conclusion

This chapter described important parts of the implementation. In the beginning, the design and the used framework for Coffee Time API implementation along with the deployment to Cloud Functions were represented. Furthermore the architecture of the application, each architecture's layer and used techniques were discussed. In the following chapter, a development process with continuous integration is described along with internal testing and final user testing before release.

```
// creates mocked TagRepository
class MockTagRepository extends Mock
                        implements TagRepository {}

void main() {
  MockTagRepository tagRepository;
  final allTags = [
    Tag(id: '1', icon: Icons.filter, title: '1'),
    Tag(id: '2', icon: Icons.filter, title: '2'),
  ];

  setUp(() {
    tagRepository = MockTagRepository();
    // when getAll called on repository
    // then answer with defined allTags array
    when(tagRepository.getAll())
         .thenAnswer((_) async => Left(allTags));
  });

  FilterBloc createBloc(Filter filter) =>
      FilterBloc(tagRepository: tagRepository,
                 initialFilter: filter);

  blocTest(
    'Remove tag and should be empty',
    build: () async => createBloc(Filter(tagIds: ['1'])),
    act: (bloc) async {
      bloc.add(Init());
      bloc.add(RemoveTag(tagId: '1'));
    },
    expect: [
      isA<FilterBlocState>(),
      FilterBlocState(
          filter: Filter(tagIds: []),
          addedTags: [],
          notAddedTags: allTags
       ),
    ],
  );
   // ...
}
```

Listing 30: FilterBloc Unit Tests.

Chapter **4**

Application Release

Before application can be released to production, it should be properly tested. In this last chapter, the development process, automated unit testing and pull-request approach of development even as a single developer are outlined. Later on, the process of incrementally tested application though internal a beta testing channels on the Android devices is explained. In the end, released version of the application is shown along with link to *Google Play Store*.

4.1 Development Workflow

Coffee Time was developed as fully open-sourced (MIT licensed) application available at GitHub. Even as a single developer, author wanted to have full control over development process. The development workflow was setup accordingly:

- *Master* branch is production code. Code within *master* has to be fully tested, working code. Each commit within *master* branch has to have own tag – version number.
- *Dev* branch is for active development.
- Each feature or bug has assigned GitHub *issue*.
- Each feature is developed within its own *feature* branch, branched from *dev* branch. The *feature* branch has to be named as `feature/X` where X is the issue number.
- Each bug has its own *fix* branch and is named as `fix/X` where X is an issue number.
- Every feature and bug fix has to be merged to *dev* branch through GitHub *Pull Request*.

- Each *Pull Request* can be completed only if automated build with unit tests is successful.
- Each commit contains reference to an issue number.

Moreover, created issues has assigned *labels*, such as architecture, design or bug, for higher clarity and each issue is assigned to the milestone. This helped to organize remaining work and stick to the plan.

```
name: 'Mobile - Build and test'
on:
   push:
      paths: mobile/**
      branches: [feature/**, fix/**, dev]
jobs:
   build-and-test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
    - uses: actions/setup-java@v1
      with:
        java-version: '12.x'
    - uses: subosito/flutter-action@v1
      with:
         channel: 'stable'
    # Get flutter packages
    - name: 'pub get'
      working-directory: mobile
      run: flutter pub get
   # Build runner
    - name: 'pub runner'
      working-directory: mobile
      run: flutter pub run build_runner \
           build --delete-conflicting-outputs
    # Build
    - name: 'build'
      working-directory: mobile
      run: flutter build aot
    - name: 'test'
      if: always() && !cancelled()
      working-directory: mobile
      run: flutter test
```

Listing 31: Configured GitHub Action.

4.1.1 Continuous Integration

GitHub has a feature called *GitHub Actions* – automation of the development workflow. Whenever pull request is created or new commits are pushed to created pull requests or *dev* branch, an automated action is run. This action checkouts current code, build it with Flutter tools and run unit tests. If any of those action fails, build is marked as failed and in case of a pull request, pull request can not be closed.

Configuration of *GitHub Actions* are done through configuration files within *YAML* format. Coffee Time build action configuration is shown in Listing 31. Action is triggered on push to *feature*, *fix* or *dev* branch and only if change set contains any file within *mobile* folder. Java and Flutter tools are installed in order to build Flutter application. Before building application, Dart packages has to be downloaded. Finally after the application is built, unit tests are started. If any of tests failed, whole action is marked as failed.

4.2 Internal Testing

In the moment, when application had minimal working feature, internal testing channel on *Google Play Services* was created in order to have early feedback from internal testers. Internal Channel provides access to applications in early stages to selected users. In case of Coffee Time application, internal testers were author's closest persons and family members. A number of testers were up to ten people and during development, they gave valuable feedback. An example of such feedback was not working navigation action on some Android versions.

During development, several versions were published for internal testing:

- **v0.0.1** – First internal test flight. Implemented nearby search without filter functionality. Other features were not available.
- **v0.1.0** – Added filter functionality, multi language support, fixed several bugs. Design improvement.
- **v0.2.0** – Added map functionality. Fixed several issues with API communication. Filtering changes and bug fixes.
- **v0.2.1** – Hotfix version. Fixed bug when application crashed in case of GPS was turned off.
- **v0.3.0** – Performance and stability fixes.

After internal testing, application was opened up for beta testers within *Beta Channel* with version *v0.3.0.* With that version more users tried application and gave more feedback. Based on the feedback, more bugs were fixed and last beta version *v0.3.1* were deployed for test.

All versions were merged to *master* branch and properly tagged, so each version can be found on GitHub project.

4.3 Crashlytics

When the application was ready to release another *Firebase* service was added to the application – *Crashlytics* [61]. *Crashlytics* is a realtime crash reporter that helps track, prioritize and fix stability issues. The official plugin for *Crashlytics* was used to enable this service. In the application only a two lines of code had to be added to enable reporting (Listing 32).

```
// ... within main() method
Crashlytics.instance.enableInDevMode = kDebugMode;
// pass all uncaught errors from the framework to Crashlytics.
FlutterError.onError = Crashlytics.instance.recordFlutterError;
```

Listing 32: Enable Crashlytics.

4.4 User Testing Re-Evaluation

When the release candidate version was prepared, same user testing as was done with the prototype version was evaluated. Five users were involved in this tests. User profiles varied from their overall mobile application skills to their personal preferences over cafes.

Test was evaluated in the same manner as before, with same test cases to know, if found issues within prototype were fixed or not. To remind, found issues were:

1. **Filter screen is confusing. Confirmation is useless.**

   Confirmation within filter screen was removed. When an user press a back button, selected filters are applied.

2. **Do not open filter screen, just open some modal window to selects tags.**

   Filter screen remained the same as a more feasible solution by author opinion.

3. **The filter icon is unclear.**

   Icon remained the same as it is standard icon used across different applications. However, two new testers confirmed once again this confusion.

4. **If an user is located on the detail screen, clicking on a tag icon should set filter.**

For now not implemented.

5. **Three different icons are used to start navigation. Icon is different on the cafe list from icon within detail screen.**

   Icons were consolidated into one icon.

6. **Cafe's name on tile can be hard to read with the contrasting background image.**

   Solved with darker photo overlay.

7. **Landscape mode is not working.**

   Landscape mode was disabled.

8. **The map marker could contain more information than the cafe's name only.**

   Currently it is not doable due to a limitation of the maps plugin.

New testers encountered old issue only with "unclear filter icon". The more serious issue was found within the tag review screen. On some screen resolutions, a confirm button to submit review can be "hidden" and the user has to scroll down a little bit to see a button. One user was confused as it looked like no button is not visible and pressed back button to confirm review. Such action did not submit review and test case failed. A solution to prevent this issue could be to use `FloatingActionButton` which is still visible on the screen. Same approach is used for "favorite" button within detail screen.

As the test results did not find any serious issues which could prevent release the application, final steps were taken to deploy the application to the production.

4.5 Release

Coffee Time was released as version *1.0.0* to the *Google Play Store* for Android devices. In order to publish a release, a proper name and description in all available languages had to be filled before the application could be accepted. Moreover, the store displays application photo, logo and a cover image. These fields were necessary to fill as well. After the release, two minor versions *1.0.1* and *1.0.2* were released as hotfixes to prevent application crashes on some devices.

Figure 4.1 shows the released version. In contrast with prototype version (Figure 2.9), there are no significant graphical changes other than

- changed font typeface,
- moved favourite button from header to FloatingActionButton,

- slightly different color pallete.

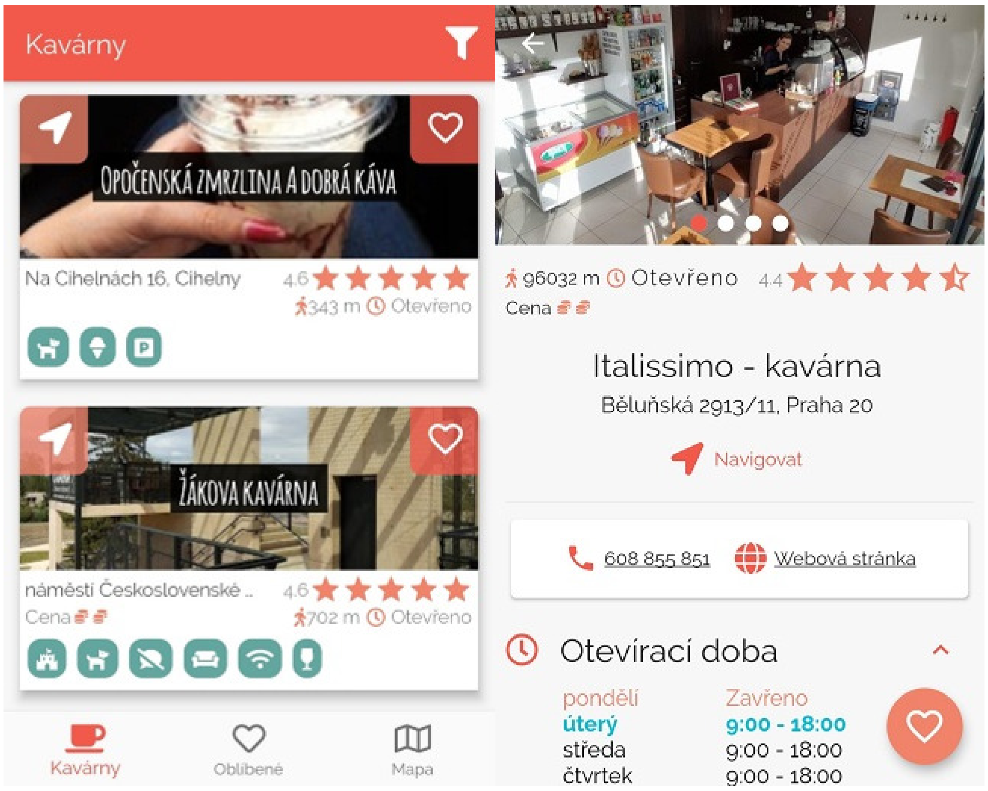


Figure 4.1: Released Version – Cafe List and Detail Screens.

Figure 4.2: Coffee Time QR Code for Downloading the Application.

4.5.1 Missing Feature from Prototype

One feature from prototype did not make it to first released version and that is custom search location. In the current version, custom location can be set through map view but not from cafe list. The implementation with this feature is partially done, as the API back-end already partially support it and in the application, the data layer has already methods for that feature. However, during implementation was acknowledged that fully functional text search should include autocomplete services from Google Places API. Unfortunately, the current design of API and application did not count with such requirements. Feedback from users includes also a requirement for such features. Thus this feature is considered as the most wanted feature in future development.

4.5.2 Conclusion

Coffee Time was successful implemented and released to production for *Android* devices. A *QR* code for downloading is displayed in Figure 4.2. As Flutter is multi-platform, *iOS* could be deployed too. However there were a few "obstacles" that prevented to do that:

- Application design was inspired from Material Design and this design do not correspond with Apple Deisgn,
- *iOS* development requires *MacOS* device in order to build *iOS* application,
- and finally, goal was to provide fully functional application at least for *Android*. In the future development, the *iOS* version is planned as well.

Conclusion

In the first chapter, Flutter framework as a new promising cross-platform framework was introduced. How the framework works, its philosophy "everything is a widget" was explained and different approaches of state management were covered.

Next chapter was focused on introducing proposed application, its designing process, prototyping and user testing. Low-Fidelity and High-Fidelity prototypes were created to verify proposed design. Both prototypes are also included along with final implementation in the appendix.

In the implementation chapter, more details were covered and also several popular packages and solutions within Flutter community were introduced. In the end, the *Android* version of the Coffee Time application was successfully tested and it was released for download.

Next Steps

Coffee Time is for sure not "feature-complete" and there is plenty of space for improvements. Ongoing future development is planned to obtain more experience with Flutter framework and bring an even better, feature "rich", application.

Next possible features and steps are:

- Add missing feature "searching cafes in custom location".
- To synchronize favourite cafes.
- A better map view with more information in it.
- Optimise application performance, responsivity and adaptability to different form factors (mobile phones, tablets, ...), screen sizes and resolutions.

- Add full *iOS* support. This includes redesign of the application to more modern cross-platform look.
- As an experiment, build application also for web and desktop platform.

Personal Author's Note in the End

I really believe that Flutter is a promising framework which increasingly gains on popularity and in the closest future there will be more opportunities as the framework will get adopted by larger companies. In the contrast with Xamarin, my subjective opinion is that the Flutter development workflow is faster, more convenient and easier to use. Of course, as with every new popular framework, there is no guarantee that Flutter will be truly the right solution in the future. But I am convinced that, at this moment, Flutter is stable to use as a production-ready framework. If the application does not use very specific platform features, Flutter can be used without any hesitation.

Bibliography

[1] Facebook Inc. React Native. [online] [accessed: 10. 4. 2020]. Available from: https://reactnative.dev

[2] Microsoft. What is Xamarin Forms. [online] [accessed: 10. 4. 2020]. Available from: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms

[3] Ionic. Ionic framework. [online] [accessed: 10. 4. 2020]. Available from: https://ionicframework.com

[4] Stack Overflow. Stack Overflow Flutter trends statistics. [online] [accessed: 28. 4. 2020]. Available from: https://insights.stackoverflow.com/trends?tags=flutter%2Cxamarin%2Creact-native%2Cionic

[5] Google LLC. Flutter framework. [online] [accessed: 10. 4. 2020]. Available from: https://flutter.dev

[6] Stackoverflow. Stack Overflow Developer Survey 2019. [online] [accessed: 28. 4. 2020]. Available from: https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-other-frameworks-libraries-and-tools

[7] Nymsa, P. *Mobile Enterprise Architecture Process Analytic Tool Based on the DEMO Methodology*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, May 2018.

[8] Google LLC. Technical overview - Flutter. [online] [accessed: 28. 4. 2020]. Available from: https://flutter.dev/docs/resources/technical-overview

[9] Google LLC. *Dart Programming Language Specification*. Fifth edition, 2019, version 2.2. Available from: https://dart.dev/guides/language/specifications/DartLangSpec-v2.2.pdf

[10] Google LLC. The Dart type system. [online] [accessed: 28. 4. 2020]. Available from: `https://dart.dev/guides/language/sound-dart`

[11] Google LLC. Dart – Platforms. [online] [accessed: 28. 4. 2020]. Available from: `https://dart.dev/platforms`

[12] Microsoft. Xamarin.Forms Data Binding. [online] [accessed: 29. 4. 2020]. Available from: `https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/data-binding/`

[13] Boelens, D. Reactive Programming Streams Bloc. [online] [accessed: 29. 4. 2020]. Available from: `https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc`

[14] Escoffier, C. 5 Things to Know About Reactive Programming. [online] [accessed: 29. 4. 2020]. Available from: `https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming`

[15] Google LLC. Introcution to widgets. [online] [accessed: 29. 4. 2020]. Available from: `https://flutter.dev/docs/development/ui/widgets-intro`

[16] Google LLC. Layouts in Flutter. [online] [accessed: 29. 4. 2020]. Available from: `https://flutter.dev/docs/development/ui/layout`

[17] Google LLC. Start thinking declaratively. [online] [accessed: 30. 4. 2020]. Available from: `https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative`

[18] Boelens, D. Flutter–Widget–State–Context. [online] [accessed: 29. 4. 2020]. Available from: `https://www.didierboelens.com/2018/06/widget-state-context-inheritedwidget`

[19] Google LLC. Differentiate between ephemeral state and app state. [online] [accessed: 30. 4. 2020]. Available from: `https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app`

[20] Google LLC. Simple app state management. [online] [accessed: 3. 5. 2020]. Available from: `https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple`

[21] Google LLC. InheritedWidget. [online] [accessed: 3. 5. 2020]. Available from: `https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html`

[22] Rousselet, R. InheritedWidget, but simple. [online] [accessed: 3. 5. 2020]. Available from: `https://github.com/rrousselGit/provider`

[23] Angelov, F. Bloc. [online] [accessed: 3. 5. 2020]. Available from: https://bloclibrary.dev

[24] Soares, P. Flutter / AngularDart — Code sharing, better together (DartConf 2018). [online] [accessed: 29. 4. 2020]. Available from: https://youtu.be/PLHln7wHgPE?t=1326

[25] Boelens, D. Flutter internals. [online] [accessed: 5. 5. 2020]. Available from: https://www.didierboelens.com/2019/09/flutter-internals/

[26] Futured. Gastromapa Lukáše Hejlíka. [online] [accessed: 12. 4. 2020]. Available from: https://play.google.com/store/apps/details?id=com.thefuntasty.gmlh

[27] Zeman, L. Pivní deníček. [online] [accessed: 12. 4. 2020]. Available from: https://play.google.com/store/apps/details?id=cz.proteus.pivnidenicek

[28] Restu devs. Restu. [online] [accessed: 12. 4. 2020]. Available from: https://play.google.com/store/apps/details?id=com.thefuntasty.restu

[29] Zomato. Zomato. [online] [accessed: 12. 4. 2020]. Available from: https://play.google.com/store/apps/details?id=com.application.zomato

[30] Google LLC. Google Maps. [online] [accessed: 12. 4. 2020]. Available from: https://play.google.com/store/apps/details?id=com.google.android.apps.maps

[31] Babich, N. Prototyping 101: The Difference between Low-Fidelity and High-Fidelity Prototypes and When to Use Each. [online] [accessed: 12. 4. 2020]. Available from: https://theblog.adobe.com/prototyping-difference-low-fidelity-high-fidelity-prototypes-use

[32] Google LLC. Material Design. [online] [accessed: 12. 4. 2020]. Available from: https://material.io/design

[33] Balsamiq Studios, LLC. Balsamiq, version 3.5.17. [online] [accessed: 1. 10. 2019]. Available from: https://balsamiq.com

[34] Nymsa, P. Hi-Fi prototype. Available from: https://github.com/petrnymsa/coffee-time/releases/tag/prototype

[35] Nielsen Norman Group. 10 Usability Heuristics for User Interface Design. [online] [accessed: 12. 4. 2020]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[36] Google LLC. Elevation – Material Design. [online] [accessed: 12. 4. 2020]. Available from: https://material.io/design/environment/elevation.html#elevation-in-material-design

[37] Google LLC. Places API Overview. [online] [accessed: 14. 4. 2020]. Available from: https://developers.google.com/places/web-service/intro

[38] Google LLC. Get an API Key. [online] [accessed: 14. 4. 2020]. Available from: https://developers.google.com/places/web-service/get-api-key

[39] Google LLC. Places API Usage and Billing. [online] [accessed: 14. 4. 2020]. Available from: https://developers.google.com/places/web-service/usage-and-billing

[40] Strassner, T. XML vs JSON. [online] [accessed: 14. 4. 2020]. Available from: https://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html

[41] Kubernetes. Production-Grade Container Orchestration. [online] [accessed: 14. 4. 2020]. Available from: https://kubernetes.io

[42] Anomaly Innovations, Serverless Stack. What is serverless. [online] [accessed: 14. 4. 2020]. Available from: https://serverless-stack.com/chapters/what-is-serverless.html

[43] Google LLC. Cloud Firestore — Firebase. [online] [accessed: 14. 4. 2020]. Available from: https://firebase.google.com/docs/firestore

[44] Google LLC. Cloud Functions for Firebase. [online] [accessed: 14. 4. 2020]. Available from: https://firebase.google.com/docs/functions

[45] Okta. OAuth 2.0. [online] [accessed: 14. 4. 2020]. Available from: https://www.oauth.com

[46] Google LLC. Firebase Authentication. [online] [accessed: 14. 4. 2020]. Available from: https://firebase.google.com/docs/auth

[47] Martin, R. Clean Architecture. [online] [accessed: 5. 5. 2020]. Available from: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[48] Martin, R. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* London, England: Prentice Hall, 2018, ISBN 978-0134494166.

[49] OpenJS Foundation. Node.js. [online] [accessed: 10. 5. 2020]. Available from: https://nodejs.org/en

[50] Node.js Foundation. Express – Node.js web application framework. [online] [accessed: 10. 5. 2020]. Available from: https://expressjs.com

[51] Google LLC. Place Search. [online] [accessed: 14. 4. 2020]. Available from: https://developers.google.com/places/web-service/search#PlaceSearchRequests

[52] Node.js Foundation. Writing middleware for use in Express.js. [online] [accessed: 10. 5. 2020]. Available from: https://expressjs.com/en/guide/writing-middleware.html

[53] Auth0. JSON Web Token Introduction. [online] [accessed: 10. 5. 2020]. Available from: https://jwt.io/introduction

[54] Google LLC. Effective Dart: Design – Equality. [online] [accessed: 17. 5. 2020]. Available from: https://dart.dev/guides/language/effective-dart/design#equality

[55] Angelov, F. Equatable. [online] [accessed: 17. 5. 2020]. Available from: https://pub.dev/packages/equatable

[56] Neal Ford, I. Functional error handling with Either and Option. [online] [accessed: 17. 5. 2020]. Available from: https://www.ibm.com/developerworks/library/j-ft13/index.html

[57] Rousselet, R. Freezed. [online] [accessed: 17. 5. 2020]. Available from: https://pub.dev/packages/freezed

[58] Robert, M. Design Principles and Design Patterns. [online] [accessed: 17. 5. 2020]. Available from: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[59] Dart Lang team. Mockito. [online] [accessed: 17. 5. 2020]. Available from: https://pub.dev/packages/mockito

[60] Angelov, F. bloc_test. [online] [accessed: 17. 5. 2020]. Available from: https://pub.dev/packages/bloc_test

[61] Google LLC. Firebase Crashlytics. [online] [accessed: 17. 5. 2020]. Available from: https://firebase.google.com/docs/crashlytics

Appendix A
Acronyms

AOT	Ahead-of-Time.
BLoC	Business Logic Component.
CTA	Coffee Time API.
GPA	Google Places API.
Hi-Fi	High Fidelity.
JIT	Just-in-Time.
JWT	JSON Web Token.
Lo-Fi	Low Fidelity.
UI	User Interface.

Appendix **B**

Content of Enclosed CD

```
README.txt ........................ Brief content description
src
  app ............................. Coffee Time source code
  examples ........................ Examples source code
  thesis .......................... Thesis text LaTeX source code
text
  thesis.pdf ...................... Thesis text in PDF format
prototype
  lofi.pdf ........................ Low Fidelity clickable prototype
```