Dmitrii Slepnev

STATE MANAGEMENT APPROACHES IN FLUTTER

Bachelor's thesis

Information Technology

Bachelor of Engineering

2020

South-Eastern Finland
University of Applied Sciences

XAMK
South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Dmitrii Slepnev	Bachelor of Engineering	December 2020

Thesis title	
State management approaches in Flutter	98 pages 0 pages of appendices

Commissioned by

Supervisor

Timo Hynninen

Abstract

Flutter has tens of different ways to solve the issue of state management, which creates an ambiguity in the selection of the approach. Flutter is a promising technology which can possibly reduce the costs of cross-platform application development. The main objective of the thesis was to categorize the state management approaches and find a way to select the most suitable one for the most common use case scenarios. Instead of focusing on the particular use cases, a set of decision-making criteria was determined, and the approaches were analyzed and compared based on these criteria.

Quantitative research methods were used in the theoretical part in order to introduce the mobile development market, compare Flutter to the competing technologies, prove that the problem of state management approach selection ambiguity indeed exists, and determine which approaches would be worth studying by measuring their popularity among Flutter developers. These methods included, for instance, analyzing and comparing the statistical data concerning the mobile development market, number of GitHub stars, various pub.dev scoring metrics and so on. Qualitative methods (e.g. critical analysis of existing literature) were used to categorize the approaches, determine the comparison criteria, analyze and compare the selected approaches and make a conclusion.

The study resulted in several new contributions to the existing literature. First, all of the at least a little bit popular state management approaches were collected in one place. Second, these approaches were categorized by the common attributes defined in the thesis. Third, the most popular and widely used representative of each group was studied in detail by implementing a real application with shared preferences, local persistence and remote API requests. These included setState, InheritedWidget, Provider, GetX, BLoC, MobX, Redux. Finally, the approaches were analyzed based on the criteria defined in the thesis, and the comparison table was created. This table allowed finding the most suitable approach for the needs of each particular project or developer.

The resulting comparison table and the implementation part explaining how to use each state management approach give an answer to the question: how to select the most suitable state management approach? It also gives the starting point beginning to use the selected approach. The goals that were set for this thesis were fully achieved; in several places the thesis provides even more detailed answers than was initially expected.

Keywords
Flutter, state management, mobile development, BLoC, Redux, MobX, ChangeNotifier, Provider, MVVM

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The world of mobile platforms nowadays is dominated by two players – Android and iOS. They compete with each other in almost everything. Each platform has its own tools which are used to develop mobile applications.

If a company wants to create a mobile application, the classic approach is to create two different applications which look and act similarly – one for Android built with Kotlin or Java, and another one for iOS built with Swift or Objective-C. In terms of costs, it is two times more expensive than if they could just write one application and compile it for each platform. This obvious idea has made many people work on finding the solution to the problem of double effort and double costs associated with the development of mobile applications. Their work has resulted in the creation of such platforms and frameworks as Xamarin, React Native, Ionic, and finally, one of the most recent ones – Flutter.

Flutter is a software development kit (SDK) for creating cross-platform applications with a single codebase that will be discussed in this thesis work. More specifically, I focus on the various state management approaches in Flutter. State management is very important; it is one of the key topics with which every Flutter developer has to work on a daily basis. The definition of state management and the reasons why it is so important in Flutter will be given in the theoretical part of this thesis.

There are tens of approaches on how to manage state in Flutter, and there is no answer to the question which one is the best. Obviously, there can be no objective answer to such question. Therefore, instead of trying to determine which approach is the best, this thesis aims to find out how to choose the right approach based on the project requirements. I was motivated to select this thesis topic after I had worked for five months as a Flutter Developer trainee and had faced this problem in real working life. Whenever a new project is started, it is very difficult to choose the most suitable state management approach. To sum up, the main objective of this thesis is to answer the question: How to choose state management approach?

The first theoretical part of the thesis studies the most popular approaches, categorizes them by the technology or logic lying behind, since some approaches have the same roots with the other ones, finds out the differences and similarities, benefits and drawbacks and determines what are the most suitable use cases for each. Of course, before starting working on state management, I introduce what Flutter is, why it exists, and when it is supposed to be used. The main objective of this part is to categorize the existing state management approaches.

Since Flutter is a new and fresh technology in the world of mobile development, there are not so many materials on this topic, and very few studies have been made on that. This is why my study is important and contributes to the existing literature. It is possible to find some blog posts and articles on the Internet related to the topic, but no one studied it yet. After reading the theoretical part, a person with no prior knowledge of cross-platform technologies should understand what the role of Flutter is and why it exists in the modern mobile development market, have an idea of how it works and what the key building blocks needed to create apps are, and understand the issue of state management selection ambiguity.

In the second, practical part I create some example applications to illustrate the main approaches. This part is about coding the actual apps, and each approach will also be explained in a more detailed way. The contribution of the practical part is to create the sample implementations of the most popular state management approaches in one place, so that a reader can not only identify the most suitable approach for his or her application, but also see how to implement it in practice. Moreover, the differences between approaches in the practical implementation will be analyzed, because theory alone cannot always provide the complete picture. It is important to implement either the same, or equally complex apps in order to make the analysis more objective. The main takeaway is that the person should be able to identify the most suitable state management approach for various use cases. Again, due to the novelty of Flutter, there is no literature where one could find this information in one place presented in a consistent manner. Therefore, this thesis will fill in this gap.

The resources used for composing this thesis are primarily scientific: books, articles, conference notes, reports and journals. I found these using Google Scholar together with the Xamk library services, when needed. The relevance of the source is assessed on a case-by-case basis. Flutter was first announced in 2017. Since the first stable release was in December 2018, the scientific resources on the topic dated 2019–2020 can be considered relevant when referring to the information that directly involves Flutter. In cases where the existence of Flutter does not affect the information referred from the source, the previously mentioned constraints do not apply. Since Flutter is so new, it is sometimes impossible to find the required information in scientific sources. In these cases, the preference is given to the official Flutter documentation, Medium posts in the official Flutter community, articles written by the recognized members of Flutter community and video recordings of the official Google presentations related to Flutter. In general, the sources directly connected to the Flutter team are automatically considered to be relevant. For statistics and numbers, the public data from various research companies like Statista and Newzoo can be used.

2 THEORETICAL PART

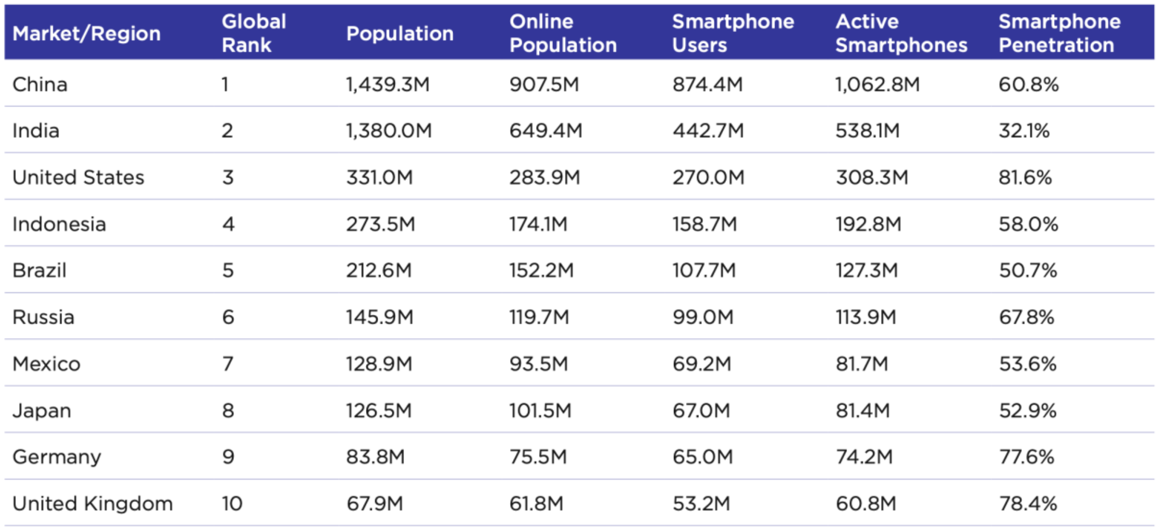
The first objective in the theoretical part is to introduce the reader to the context of the mobile development market, which will help with understanding why Flutter exists and why this technology is important. In order to do so, the situation in the mobile development market is described. It should help understanding where the place of this technology in the mobile world currently is. The second objective is to outline the main concepts of Flutter and to define the main terms which are crucial to understand when speaking about this technology. Following this, a theoretical study on the state management approaches in Flutter is conducted.

2.1 Current situation in mobile development

This section illustrates the current situation in the mobile market and determines the place of cross-platform solutions in the modern mobile development world. This is needed to help the reader understand the context and the importance of my research which is conducted further. It also gives a well-grounded answer to the question: What is the role of Flutter in mobile development?

2.1.1 Mobile market

There are 3.5 billion smartphone users in the world nowadays. This number is predicted to increase by 300 million in 2021 (Statista 2019).
According to Yale University (2020), the population of Earth in 2020 is about 7.8 billion. This means that almost half of the world's population already has smartphones. However, in the developed countries the percentage of smartphone users is even higher. The absolute number of smartphone users in ten countries is compared in Figure 1.



Market/Region	Global Rank	Population	Online Population	Smartphone Users	Active Smartphones	Smartphone Penetration
China	1	1,439.3M	907.5M	874.4M	1,062.8M	60.8%
India	2	1,380.0M	649.4M	442.7M	538.1M	32.1%
United States	3	331.0M	283.9M	270.0M	308.3M	81.6%
Indonesia	4	273.5M	174.1M	158.7M	192.8M	58.0%
Brazil	5	212.6M	152.2M	107.7M	127.3M	50.7%
Russia	6	145.9M	119.7M	99.0M	113.9M	67.8%
Mexico	7	128.9M	93.5M	69.2M	81.7M	53.6%
Japan	8	126.5M	101.5M	67.0M	81.4M	52.9%
Germany	9	83.8M	75.5M	65.0M	74.2M	77.6%
United Kingdom	10	67.9M	61.8M	53.2M	60.8M	78.4%

Figure 1. Top 10 Markets by absolute number of smartphone users (Global Mobile Market Report 2020)

Figure 1 shows, for example, that in the United States 81.6% of the population (270 million people) are smartphone users. This is only 13.9 million less than the overall "online population" – people who use the Internet.

These large figures indicate that the mobile market is enormously huge nowadays, and it keeps growing year by year. Mobile applications and games are two markets which especially benefit from this. We can draw a simple logical conclusion from this – more users mean more customers, more customers potentially mean more money in the industry, more money and customers mean higher demand for various kinds of mobile applications. And according to

Chamley (2014), demand creates its own supply. The mobile application market is valued at USD 170.52 billion in 2020 and is expected to surpass more than a double of its current value and reach USD 366.34 billion by 2027 (Grand View Research 2020).

All these numbers are just a confirmation of the main fact which I want to underline in this chapter – there is a fierce competition in the market of mobile applications. This means that for a big number of small companies, as well as for the players which will enter the market in the following years, it won’t be easy. They do not have such budgets as the giants like Google or Facebook. The market is already established, and in order to be competitive, the young players have to be efficient. “These small app vendors cannot afford to make mistakes and need to react in an agile way to market opportunities” (Nayebi et al. 2016).

Let us make a short conclusion for the situation in the mobile apps market for today and the near future. The market is already quite big and established, but it is predicted to grow even bigger. However, at the same time, it is and will be extremely competitive, which especially affects the smaller players who have to adapt and become more efficient. Flutter can possibly play a key role in reducing the costs, increasing the efficiency and surviving the fierce market competition. This will be explained in more detail in Section 2.1.4.

2.1.2 Mobile operating systems

Apple iOS and Google Android together possess almost 99 percent of the global market share in the field of mobile operating systems. Android has almost 75% of the market share, while iOS has approximately 25% (Statcounter 2020a). This may seem like it should be enough for most of the small to medium sized developers to create their applications for only one platform – Android – and have 75% of the users for 50% of the effort. This is not true due to the fact that in countries with higher purchasing power the share of iOS devices is predominant. For example, in the USA, iOS has 60% of the market (Statcounter 2020b). This results in the fact that iOS apps in theory can earn more money per user than Android apps.

2.1.3 Modern mobile development options (Native vs Cross-platform)

This section introduces the options that developers currently have when they start creating a new mobile application. Many developers want to make their apps available on both mobile platforms in order to get the maximum coverage of the audience. But each platform has its own tools for app development which are not compatible with each other. According to Hu et al. (2019) this brings developers to a choice between two major options:

1. Build the application *natively* for each platform and maintain at least two versions of the same application.
2. Make a *cross-platform* app that shares the single codebase across different platforms. Write the code once, then compile it for each platform.

The first approach means that the programmers must develop and maintain two different projects which are created using different technologies, with the only thing shared in common – the end result on the device. Based on common sense, this means that the resources needed for one app have to be doubled to reach the same outcome on both platforms. And the things become even more complicated when it gets to the bug fixes, updates and platform-specific issues. However, the result will be perfect on both platforms. The result will be an optimized application which takes into account platform-specific guidelines and constraints and communicates with the operating system natively. Nevertheless, developers should remember that the amount of the resources required to build such an app is huge.

Sometimes spending the resources on creating two native applications can be justified. Serious projects with high budgets for which the security is vital, which need to utilize native operating systems' APIs and have the best possible performance and granular control over literally everything that happens with their app, should definitely consider using the native approach. A banking application can be a good example of such a project.

At the same time, not all projects need all of this, and many projects simply cannot afford spending that much money. This is when a cross-platform solution should be applied. It will help the developers to create the app with one codebase for several platforms, and only this code has to be maintained later on to introduce the updates to the application. In situations where there is no need for deep integration with the OS APIs, this solution can generally cover all the needs and bring a similar result to what would have been expected from the native approach, but faster and cheaper.

“A small to medium-sized cross-platform development project costs $35,000 to $70,000 using native development kits, but only $20,000 to $40,000 using hybrid (cross-platform) development tools” (Hu et al. 2019). This proves that there is a significant difference between native and cross-platform apps in terms of costs. And all previously mentioned facts correlate with the conclusion made in Section 2.1.1 – cross-platform development is one of the instruments for smaller players to adapt to the market conditions, become more efficient and survive in the fierce competition.

2.1.4 Cross-platform solutions for mobile development

Figure 2 shows a graph which compares the popularity of different cross-platform development solutions in 2019 and 2020. It somehow helps us not only to understand the current situation, but also to predict where the trend goes in the near future.

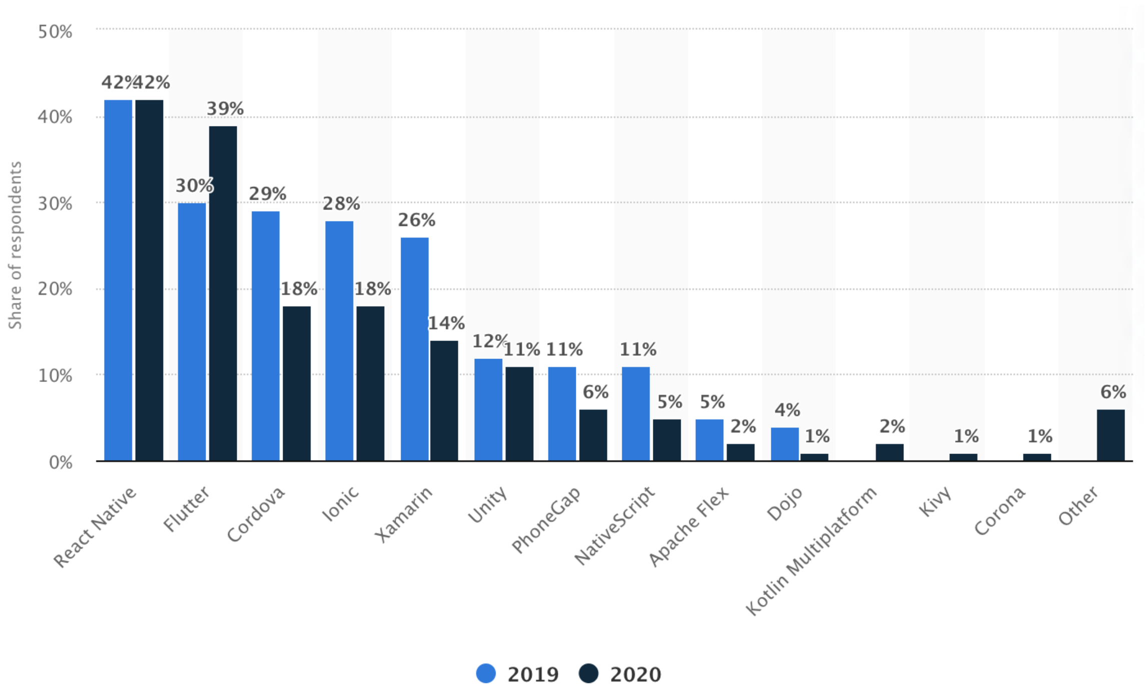


Figure 2. Cross-platform frameworks used by the developers worldwide (Statista 2020)

The most popular solution for cross-platform development as of June 2020 is **React Native**, which is a JavaScript framework for writing mobile applications for iOS and Android that are rendered natively. It is based on React, Facebook’s JavaScript library for building user interfaces. React Native applications are written using a mixture of JavaScript and XML-like markup, known as JSX. Then the React Native “bridge” invokes the native rendering APIs in Objective-C/Swift (for iOS) or Java/Kotlin (for Android). Thus, the resulting application is rendered using real mobile UI components and looks like any other mobile application. React Native also exposes JavaScript interfaces for platform APIs, so the apps are able to access platform features like the phone camera, microphone, geolocation and so on. (Eisenman 2015, 1–2.)

Figure 2 clearly shows that the popularity of React Native in 2020 remained at the same level as in 2019 (42%). It can be explained by the fact that it already has a big ecosystem. It means that there is a lot of libraries, tutorials, and also a big community of developers. Also, many cross-platform projects which have been already started earlier and selected React Native as the best alternative at that time, have to be supported, and obviously rewriting the whole project from scratch using another technology quite often cannot be economically justified. However, it is highly possible that React Native has reached its peak.

The second most popular cross-platform solution in Figure 2 is Flutter. **Flutter** is an open-source user interface software development kit created by Google. It is used to develop applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia, and the web – all from a single codebase. Flutter apps are written in the Dart language and make use of many of the language's advanced features (Wikipedia 2020a). Like React Native, Flutter uses reactive views. However, while React Native transpiles to native widgets, Flutter compiles all the way to native code. Flutter controls each pixel on the screen, which avoids performance problems caused by the need for a JavaScript bridge that can be observed in the previously mentioned technology. (Flutter Website 2020a.)

In 2020 Flutter gained 9% and reached 39% popularity, which is quite close to React Native. Statista (2020) provides data for June 2020. However, in October 2020, when this section was written, the situation already might have changed. GitHub stars is a measure of popularity of repositories. People can use the stars to bookmark the repositories they are interested in or to show that they like the repository. I have compared the number of GitHub stars for the repositories of Flutter and React Native and figured out that flutter/flutter has 103k stars, while facebook/react-native has only 90.6k stars. This can indicate that Flutter has actually either already taken over React Native, or it will happen in the nearest future. Additionally, I have used a web tool to build a graph which compares the number of stars of the two repositories over time (Figure 3).

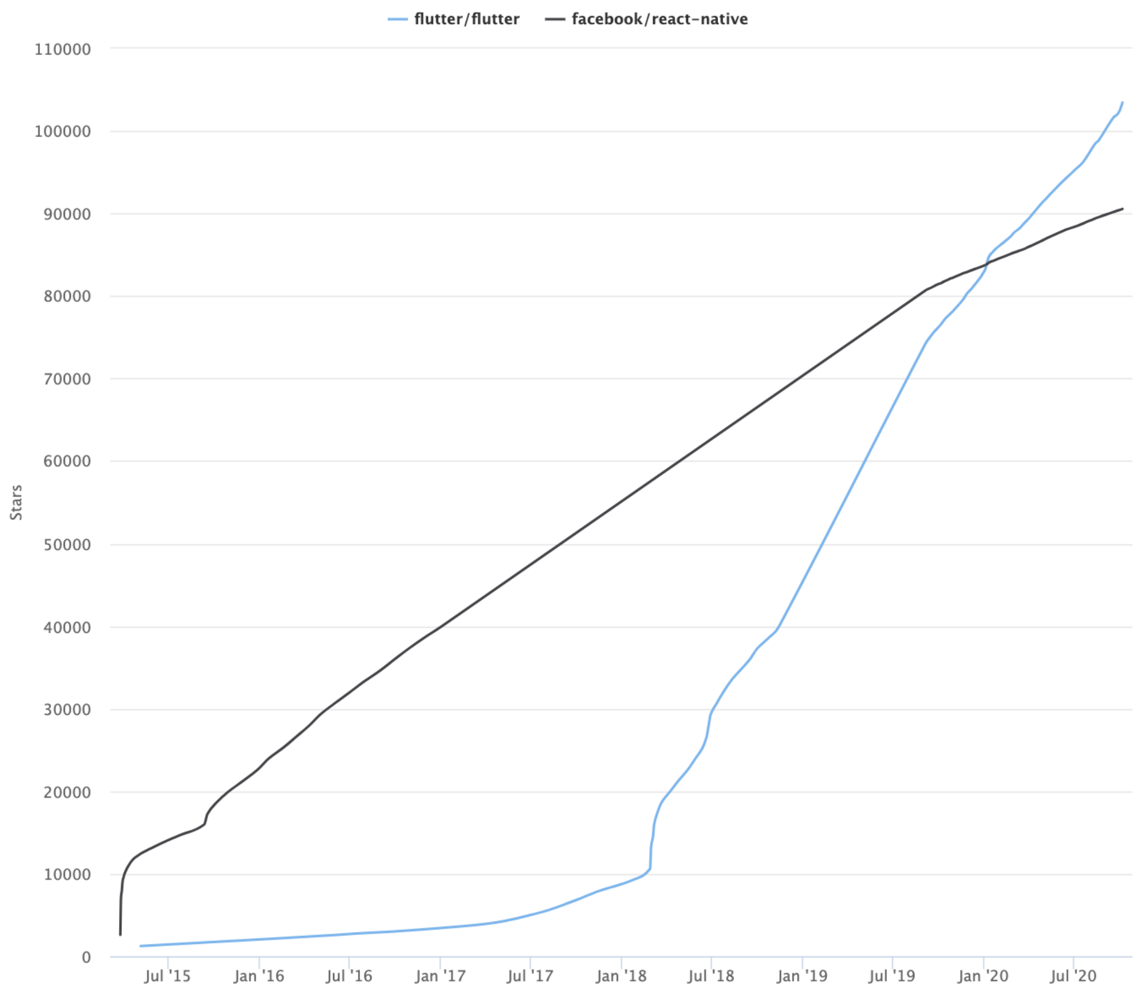


Figure 3. Flutter versus React Native popularity on GitHub

This graph shows us that from the beginning of 2020 till October 2020 React Native gained only about 7,000 new stars, while Flutter gained over 20,000. It clearly shows that Flutter is now gaining popularity significantly faster than React Native. Figure 2 also shows that other technologies like Cordova, Ionic, Xamarin etc. are losing their popularity extremely fast. Since the popularity of React Native remained at the same level, and popularity of Flutter has increased, we can draw a conclusion that people mostly start switching from the above-mentioned technologies to Flutter. In other words, these technologies are either in the end of their lifecycle, or experience difficult times – in both cases, they are not worth discussing in this thesis. The technologies like Kotlin Multiplatform or Kivy, on the other hand, are too young and immature.

To sum up the previous four sections, the mobile applications market is large and continues to grow every year, which increases the competition and makes the players to find more efficient ways to develop their applications, which is especially true for small companies that just enter the market. One of the key issues in the modern mobile development is that there are two equally important operating systems, which have completely different software development tools. It doubles the work that has to be done to develop an application for both platforms, since actually two applications have to be developed. Cross-platform solutions mitigate this issue by allowing to compile apps for several platforms from the single codebase. In 2020, Flutter is the most promising and fastest-growing cross-platform solution. It exists not to replace the native development, but to help the companies with fewer resources to be competitive. And this is exactly the answer to the question why Flutter exists, and what its role is.

2.2 Flutter

This section introduces the basic technological concepts of Flutter. The main goal is to help the reader understand how it works and what the key building blocks needed to create Flutter applications are. Flutter is an SDK developed by Google for cross-platform application development. Its goal is to provide the developers with the ability to create beautiful apps for mobile, web and desktop from a single codebase. (Flutter Website 2020b.)

According to Flutter Website 2020b, Flutter provides the following benefits:

1. Fast development due to stateful hot reload. This means that a developer can make a change to the code, save it, and see the result immediately, without a need to rebuild the app which is usually the case for the native development.
2. Expressive and flexible UI. The user interface is promised to be rendered extremely fast due to the high performance of the framework by design. The declarative layouts in Flutter provide the developer with the granular control over what actually is displayed on the screen. Flutter is also capable to mimic the native UI design of the operating system.

3. Native performance. The SDK takes into account all critical platform differences (scrolling, navigation, fonts and icons), and the code is compiled into native ARM machine code (i.e., there is no "JavaScript bridge" like in React Native).

The very first alpha release of Flutter took place in May 2017. The first stable release was on 4 December 2018, after which it started to gain popularity in the field of cross-platform mobile development solutions. (Wikipedia 2020a.)

2.2.1 Dart

All Flutter apps are written in the Dart programming language. Dart was initially Google's attempt to create an alternative to JavaScript by including the Dart VM into Google Chrome, which would allow this web browser to interpret and execute the Dart code. However, Google has failed to do it, because it still needed to support interoperability with JavaScript which did not allow creating the language they wanted. Nevertheless, Dart has found its place as a language for Flutter, and its primary use nowadays is Flutter. The main page of Dart demonstrates its use mostly with Flutter rather than alone. (Dart Website 2020.)

According to the Dart Programming Language Specification (2019), it is a class-based, single-inheritance, pure object-oriented programming language. Dart is also optionally typed. It supports reified generics.

2.2.2 How it works

After introducing Flutter and Dart in general, it is time to explain how it works. This section bases on the article of Omotunde (2019) and Flutter Website (2020d). Usually, when the app is developed using native instruments, the resulting application communicates with the OS and requests it to draw the so-called OEM widgets (buttons, text fields etc. provided by the operating system). With the cross-platform technologies, the situation is different. Because JavaScript is not capable to contact OEM widgets directly, in the frameworks like React Native, developers write JS code which is then interpreted by the framework and the framework requests the OEM widgets from the OS. This creates a JavaScript bridge which is a significant overhead affecting the performance in an extremely negative way.

Flutter is different. It does not use the OEM widgets at all. In other words, whenever a developer creates a button in Flutter, the framework does not ask the operating system to draw it (Omotunde 2019). Instead, it uses its own renderer, and draws the button itself, pixel-by-pixel. The high-performance renderer in Flutter utilizes Skia Graphics Engine which is also a product of Google. It is used for rendering in such famous products as Google Chrome, Chrome OS, Mozilla Firefox, Android, LibreOffice and others. (Wikipedia 2020b.)

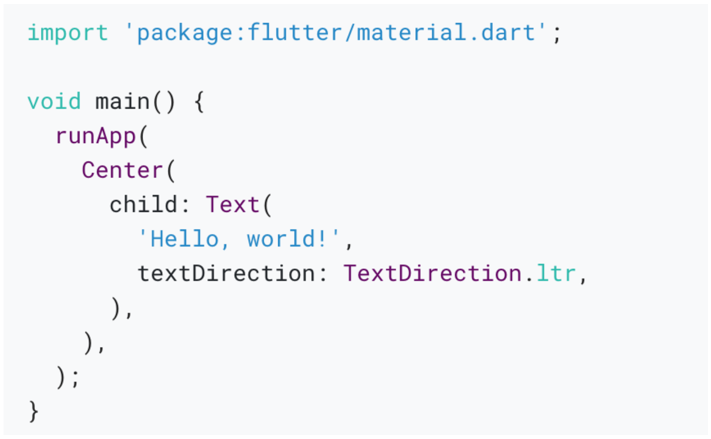
In sum, the previously mentioned facts mean that Flutter actually acts more like a game engine rather than a traditional cross-platform development solution. Unity or Unreal Engine, for instance, can be compared to Flutter to some extent. Flutter draws user interfaces instead of sprites and objects in games. The apps made with Flutter can look exactly the same way as native apps, but in fact they don’t have that much in common with native, because instead of requesting the OS to provide an OEM widget, Flutter draws everything itself. This means that if a developer wants, he or she can make the user interface look exactly the same on all iOS and Android versions supported by Flutter. Or, as another example, Flutter can bring iOS 14 look and feel to iOS 10.

2.2.3 Widgets

Widgets are the main UI building blocks of Flutter applications. An extensively advertised phrase about Flutter is “Everything is a widget”. In fact, this is absolutely true. However, it needs to be clarified: every user interface element is a widget. (Flutter Website 2020c.)

Flutter emphasizes widgets as a unit of composition. Each widget is an **immutable** declaration of a part of the user interface. Widgets form a hierarchy that bases on composition. Each widget is nested inside its parent and can receive context from the parent. And this structure goes up to the topmost root widget which is a container holding the whole Flutter application. (Flutter Website 2020c.)

Figure 4 shows the code that is needed to display the “Hello, world!” text in the center of the screen.



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

void main() {
  runApp(
    Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Figure 4. Hello World app in Flutter

It is built by combining two widgets together: the root widget in the widget tree here is Center. All it does is to ensure that its child is centered on the screen. Text widget is then nested inside the Center widget, and the positional argument with the text alongside with the named textDirection argument is provided.

Again, once declared and rendered on the screen, the widget is immutable. It is impossible to access the UI object representing the widget on the screen and start performing any mutations on it. How to change the widget when it is already rendered, though? To answer this question, we need to understand the differences between the declarative and imperative UI and introduce the concept of state.

2.2.4 State and the declarative UI

The first question is What is state? The definition can be found in Flutter API docs (2020): “**State** is information that can be read synchronously when the widget is built and might change during the lifetime of the widget.” Flutter Website (2020f) provides a definition of state in simple words: whatever data you need in order to rebuild your UI at any moment in time.

Basically, state represents that dynamic data that affects what is shown on the screen. As mentioned in Section 2.2.4, the widgets are immutable. But they can have the state object tied to the widget, and whenever the state changes, it is possible to trigger the widget rebuild. This means that instead of changing the properties of the existing widget object, a new widget with the new properties is built and rendered on the screen.

Another useful explanation is provided on the Flutter Website (2020e):
“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.”

The change of state in one widget usually triggers only updates to this one widget, without affecting other widgets (ancestors, predecessors), which makes Flutter very efficient in terms of performance. The ideas of having immutable UI widgets with mutable state objects come from the **declarative** nature of the framework. Another style of UI programming is called **imperative**. It is used in Android SDK or iOS UIKit. In short, to build a UI using the imperative style, a programmer has to manually construct a full-functioned UI entity, and later mutate it using methods and setters when the UI changes. (Flutter Website 2020g.)

As already mentioned, Flutter is different. It uses declarative UI programming style where the programmer only has to describe the current state of the user interface, and let the framework manage the actual transitions. With the declarative approach, it is also absolutely fine to simply rebuild the parts of the UI instead of modifying it. Figure 5 illustrates how the layout on the screen is built.

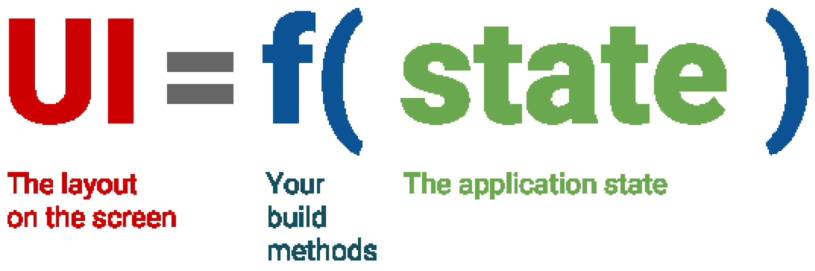


Figure 5. How Flutter renders the user interface (Flutter Website 2020h)

Basically, the application state, which contains the state objects of all widgets, is passed through the build methods of the widgets. The build methods return the actual static layout which is then shown on the screen. By this time, the reader should have an understanding of what state is in Flutter, and how it is used by the framework to render the user interface.

2.2.5 State management

We have already discovered that each dynamic UI element in Flutter must have the corresponding state, and it is possible to change this element only by changing the state and somehow notifying the framework that a change might have taken place there.

User interfaces usually contain much more than one dynamic element. One screen can have tens or even hundreds of dynamic widgets, the state of which can change based on the state of other widgets. If we have a simple switch and a text label which says “On” or “Off”, based on whether the switch is turned on or off, we can say that the state of the label depends on the state of the switch (Figure 6).

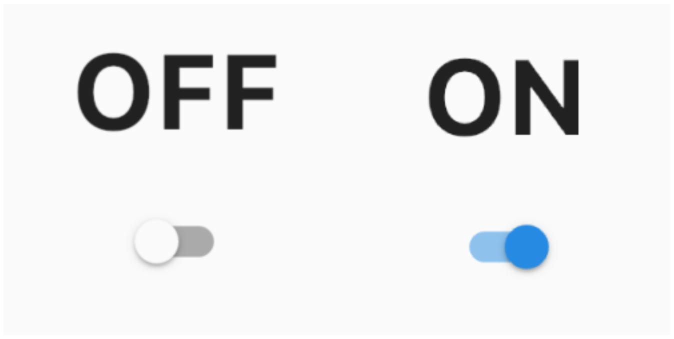


Figure 6. Illustration of how the state of one widget can depend on the state of another

In the case of the switch described above, it is quite easy to keep track of the state with the built-in mechanisms of the Flutter framework. As the app becomes more complex and such functionality as OS API calls, local storage access, HTTP requests etc. is added, it becomes significantly more difficult to keep track of the states of all widgets and to update the states correctly and efficiently, while keeping the code readable, testable and maintainable. This is when the developers realize that some more advanced approaches to state management are required.

Turning to the definition of **state management**, it is the management of the state of one or more user interface controls such as text fields, OK buttons, radio buttons, etc. in a graphical user interface. In this user interface programming technique, the state of one UI control depends on the state of other UI controls. (Wikipedia 2020a.) The example that is given in the aforementioned source describes a situation when a state managed button will be in the enabled state only when the input fields have valid values, and otherwise it will be in the disabled state. It is also said in the same source that as the apps grow, chances are that state management can end up being one of the most complex problems in the UI development. That’s exactly the reason why the topic of this thesis is very important. The only way to solve the problem of state management is to find the most suitable approach out of the tens of existing ones.

2.3 Approaches to state management

This section reports a theoretical study in the existing approaches to state management starting with official Flutter team's recommendations, based on Flutter Website (2020i). The Flutter team acknowledges that state management is indeed a complex topic. Therefore, they provide the developers with some recommendations. I have combined all their recommendations into Table 1 below.

Table 1. Flutter Team state management recommendations

State management approach	Description from the Flutter team
Provider	A recommended approach.
setState	The low-level approach for ephemeral widget state.
InheritedWidget & InheritedModel	The low-level approach used to communicate between ancestors and children in the widget tree. It is used in many other approaches under the hood.
Redux	A state container approach brought to Flutter from the web development world.
BLoC / Rx	A family of stream/observable based patterns.
MobX	A popular library based on observables and reactions.
GetX	A simplified reactive state management solution.

The team behind the Flutter project has already collected a list of the most popular approaches in order to provide the developers with the starting point. Because they also describe Provider as the recommended approach, the reader may ask why the answer to the main question of this thesis would not be to use Provider. This is a valid point. However, the Flutter team also gives at least six other approaches alongside with Provider, and based on this fact, we can make a conclusion that Provider is not an ideal solution for all situations.

The Flutter team which works on this project every day and is the most trustworthy source of information about Flutter, has collected seven approaches which they have found to be the most widely used. This means that later in the practical part exactly these approaches have to be studied in detail.

2.3.1 A bigger selection

Nevertheless, there are much more options for managing state in Flutter applications. It is only possible to find the most popular ones on the Flutter website. It is also possible that the Flutter team has provided the most popular approach out of a family of similar approaches.

The real number of state management approaches is very difficult to determine, because every developer can create his own approach and publish it. It may be used only by this developer and perhaps several other developers. Obviously, such solutions are too small and unpopular to be considered in this thesis, as the objective that was set in the very beginning is to study only the most popular solutions. However, in order to reach the objective of explaining the state management selection ambiguity, the smaller approaches at least need to be mentioned. This means that we need to find the full list of approaches somehow. As already mentioned before, the official sources of the Flutter team only provide a list consisting of seven approaches, which does not feature all approaches.

Therefore, the next step is to find as many approaches as possible. I tried to search with the following queries in Google:

1. “List of Flutter state management approaches”
2. “A full list of state management approaches in Flutter”
3. “Flutter state management”
4. “Flutter state management options”

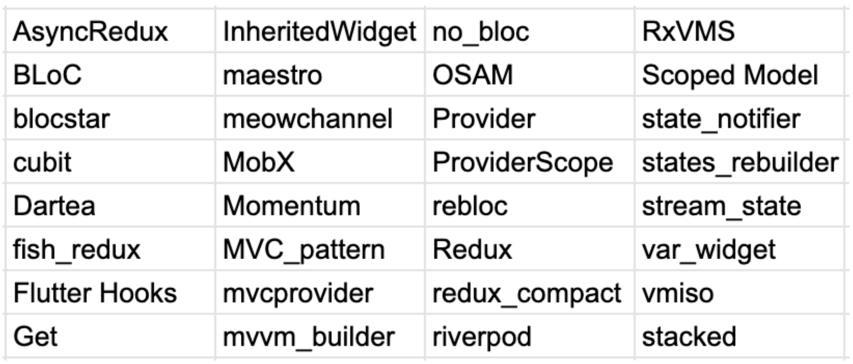
None of these brought me to the place where I could find a full list of approaches. These queries returned the Flutter documentation featuring the recommendations covered in Section 2.3. Other search results were mostly articles describing one or more of the approaches mentioned before. These results are not enough for the study that is conducted here, so I had to come up with other options.

Flutter has a fast-growing community of developers using it. I had to find the place with the highest activity of the community members and try to search in it. Flutter Website (2020j) again can help with this as they have a separate page for listing the places where the community is the most active at, including Stack Overflow, Twitter, Medium, Slack, Discord, Reddit and some other platforms. The platforms with the highest chances of finding the information I was looking for were Medium which features a lot of blog articles and Reddit which has the posts of all kinds, from small to large.

I used the same queries for Medium as the previously mentioned queries for Google, and the result was the same – I didn’t manage to find anything. However, Reddit turned out to be much more useful. It has a large community of Flutter developers with more than 50,000 users. This is the place where beginners can ask their questions and get answers, and experts can share the results of their work, so the community is very diverse in terms of the proficiency of its members. It was founded in 2013, which is when the general public had no clue that Google is working on Flutter. Nowadays it is a very active community with several tens of posts per day. All this makes it a perfect place for a developer who comes up with a new approach for state management to share it with other developers and see, if it succeeds.

I tried searching with several queries in the r/FlutterDev subreddit, looked through an extremely huge number of posts, and finally found what I was looking for. The query that worked for me was “Flutter State Management library” as I was already thinking about simply trying to collect as many approaches as possible myself manually. However, I was lucky to find a comment for one of the posts where a developer was sharing his approach to state management (Reddit 2020). This comment included about 30 existing state management approaches/libraries. Later I found out that some developers were keeping track of the new state management solutions and adding them to the list which was migrating from one post to another. It was extremely difficult to find, but it is exactly what was needed for the present study.

After I had obtained the full list, removed some approaches that were not in use anymore and added some approaches which I found myself in the Reddit community, I ended up with a list of 32 state management approaches (i.e., libraries). This list is considered full for this study, although it is possible that some unpopular libraries are missing. The list is presented in Figure 7.



AsyncRedux	InheritedWidget	no_bloc	RxVMS
BLoC	maestro	OSAM	Scoped Model
blocstar	meowchannel	Provider	state_notifier
cubit	MobX	ProviderScope	states_rebuilder
Dartea	Momentum	rebloc	stream_state
fish_redux	MVC_pattern	Redux	var_widget
Flutter Hooks	mvcprovider	redux_compact	vmiso
Get	mvvm_builder	riverpod	stacked

Figure 7. The full list of 32 state management approaches in Flutter

To sum up, the result of the study reported in this section is that we now have a list of 32 state management approaches. The fact that there are so many different libraries and approaches to solve only this one problem proves that there indeed exists an ambiguity in the selection of the state management approach.

2.3.2 Determining approaches worth studying

It is very good that more than 30 approaches are listed in this thesis. However, it is not possible to study in detail each of them, because it would take too much time and the result would be too large for a thesis work. It means that I need to narrow down the amount of the approaches that will be considered in the further research, and to remove the approaches which are not worth studying.

A good indicator of whether an approach should be considered is its popularity. Google has created a website “pub.dev” which is used for the publication of Dart packages (Pub.dev 2020a). In Flutter/Dart world, it is currently the only reliable source of public packages, and at the same time the largest collection of them.

The team behind this website has developed a three-dimensional scoring model which is displayed for each package in order to give the developers an insight of the current state of the package mostly in terms of its perception by the community. Let me briefly introduce the criteria used on this website, based on the Pub.dev (2020b):

1. **Likes** – a measure of how many developers have liked the package. It provides a raw measure of the overall sentiment of a package from peer developers.
2. **Pub score** – a measure of quality. It includes several dimensions of quality such as code style, platform support, maintainability.
3. **Popularity** – a measure of how many developers use a package, providing insight into what other developers are using. It measures the number of apps that depend on a package over the past 60 days. The result is shown on a normalized scale from 100% (the most used package) to 0% (the least used package).

It was important to introduce these dimensions now, but it is not yet possible to make any decisions on which approaches should be eliminated as unpopular. So, the first thing to do should be collecting the raw data on each approach. Table 2 presents the raw data that I have collected by manually checking the pub.dev page of each package and copying the values into the table.

Table 2. Raw data of three-dimensional scores for each approach (27 September 2020)

Name	Likes	PUB	Popularity
AsyncRedux	59	100	89%
BLoC	1163	110	99%
blocstar	1	100	0%
cubit	564	110	99%
Dartea	2	90	75%
fish_redux	32	110	91%
Flutter Hooks	294	110	96%
Get	1418	100	98%
InheritedWidget	-	-	-
maestro	6	110	24%
meowchannel	6	40	21%
MobX	409	110	98%
Momentum	63	110	69%
MVC_pattern	47	100	97%
mvcprovider	4	90	40%
mvvm_builder	9	100	54%
no_bloc	6	110	90%
OSAM	13	90	60%
Provider	2307	110	100%
ProviderScope	0	90	48%
rebloc	10	110	73%
Redux	153	105	97%
redux_compact	8	100	39%
riverpod	188	110	94%
RxVMS	30	105	87%
Scoped Model	89	100	97%
state_notifier	87	100	95%
states_rebuilder	238	100	96%
stream_state	3	110	63%
var_widget	0	90	0%
vmiso	0	80	31%
stacked	409	110	96%

It is possible to see the overall situation from Table 2, and the process of finding out which criteria we can use in order to eliminate the unpopular libraries from the research can be started. I wanted to be very careful with this elimination process since I don’t want to accidentally remove an approach which is actually valid, and this is exactly why after experimenting with the criteria and manually checking the results of the filtered table, I came to a conclusion that such a simple condition as “*a package must have 10 or more likes*” removes all the “dead” packages from the list, while keeping the not very popular, but “alive” packages there, which is exactly what I wanted to reach. Table 3 was obtained using this rule.

Table 3. Filtered and sorted list of approaches to be considered in this thesis

Name	Likes	PUB	Popularity	Link
InheritedWidget	-	-	-	https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html
Provider	2307	110	100%	https://pub.dev/packages/provider
Get	1418	100	98%	https://pub.dev/packages/get
BLoC	1163	110	99%	https://pub.dev/packages/flutter_bloc
cubit	564	110	99%	https://pub.dev/packages/bloc
MobX	409	110	98%	https://pub.dev/packages/mobx
stacked	409	110	96%	https://pub.dev/packages/stacked
Flutter Hooks	294	110	96%	https://pub.dev/packages/flutter_hooks
states_rebuilder	238	100	96%	https://pub.dev/packages/states_rebuilder
riverpod	188	110	94%	https://pub.dev/packages/riverpod
Redux	153	105	97%	https://pub.dev/packages/redux
Scoped Model	89	100	97%	https://pub.dev/packages/scoped_model
state_notifier	87	100	95%	https://pub.dev/packages/state_notifier
Momentum	63	110	69%	https://pub.dev/packages/momentum
AsyncRedux	59	100	89%	https://pub.dev/packages/async_redux
MVC_pattern	47	100	97%	https://pub.dev/packages/mvc_pattern
fish_redux	32	110	91%	https://pub.dev/packages/fish_redux
RxVMS	30	105	87%	https://pub.dev/packages/rx_command
OSAM	13	90	60%	https://pub.dev/packages/osam
rebloc	10	110	73%	https://pub.dev/packages/rebloc

After applying the filter with the condition defined previously and sorting the table in descending order by the number of likes, I have obtained the resulting Table 3. This table now contains twenty most popular approaches to state management in Flutter with the three-dimensional Pub.dev scores and the links to the corresponding packages.

2.3.3 Grouping the approaches

The next goal of the research is to categorize the approaches into groups based on the technological idea lying behind each approach. The only reliable way which I have found to do it was to manually read the description, usage instructions, and check the examples for each approach from Table 3. Flutter Website (2020i) was also partially used as a basis for finding out the existing families of approaches already listed by the Flutter team.

Firstly, we have to understand the difference between the ephemeral and app state. According to Flutter Website (2020k), ephemeral state (which is also sometimes referred to as UI state or local state) is the state that you can neatly contain in a single widget. Other widgets seldom need to access this kind of state. For example, it can be a current page in a PageView, or current progress of a complex animation. Application state, on the other hand, is state that is not ephemeral, that you want to share across many parts of the app, and that you want to keep between user sessions. Figure 8 shows an easy way to differentiate between app state and ephemeral state.

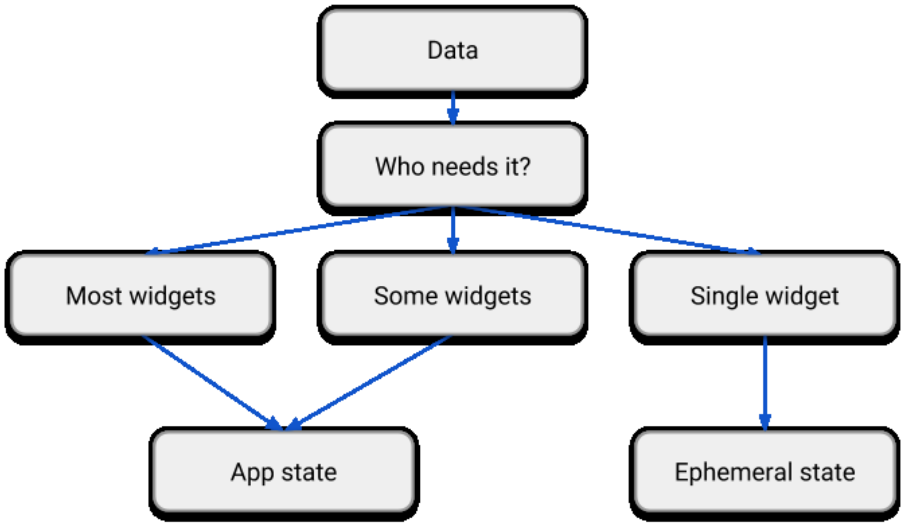


Figure 8. App state vs ephemeral state

Ephemeral state management is generally not an issue in Flutter since it can be easily done using the low-level approach built into the framework called **setState**.

So, this chapter is mainly focused on grouping the app state management approaches.

According to Flutter Website (2020i), **InheritedWidget** is a low-level approach used to communicate between ancestors and children in the widget tree, and it is used in many other approaches under the hood. We can also add ChangeNotifier which is also a part of Flutter SDK as InheritedWidget is usually used for providing the dependencies, while ChangeNotifier is more flexible for actually notifying about the state changes. It means that some of the approaches are just wrappers around the original InheritedWidget or ChangeNotifier, and possibly this is one of the groups of state management approaches. We can call this group "**Wrappers around Flutter SDK built-in classes**". I have tried to figure out which of the approaches from Table 3 could be added to this group by reading the description of each package, looking at the code and usage examples. I was mainly looking for something that would explicitly say that this package is somehow connected to the InheritedWidget or ChangeNotifier. I have managed to find out that in addition to the InheritedWidget itself, there are 5 more packages that are either rewritten implementations of it, or some improved versions using this approach under the hood: *provider, riverpod, state_notifier, osam, Flutter Hooks*. Then there is a group of MV* patterns implementations which specifically focus on making the package in such a way that it is easier for the programmers to implement the MV* (e.g. MVC, MVVM) patterns. This group includes *stacked*, *momentum* and *mvc_pattern.*

Another big family of approaches is **Redux** and its various implementations for Flutter. It was influenced by the concepts of **functional** programming significantly. More about it in the following chapters. This group includes, for instance, the following approaches: *Redux, AsyncRedux, fish_redux*.

Following this, there is one more large family– **reactive** approaches, which are based on the reactive programming paradigm, as the name implies. It means that in these approaches events trigger the execution of some code (reaction). These approaches are usually based on streams/observables. Various **BLoC** pattern implementations combined are one group that belongs to the reactive approaches. It will be discussed in detail later. For now, we only need to know that such packages as *bloc*, *cubit* and *rebloc* belong to the group of BLoC reactive approaches. There also are other reactive approaches which cannot be grouped based on some common pattern they tend to implement: *MobX, GetX, RxVMS, states_rebuilder, scoped_model*.

So, as a result, we have 3 big groups:

1. Wrappers around Flutter SDK built-in classes.
2. Functional programming based – Redux implementations.
3. Reactive programming based – BLoC family, MobX, GetX, other.

The result of the conducted research can be presented as a graph which can be seen in Figure 9.

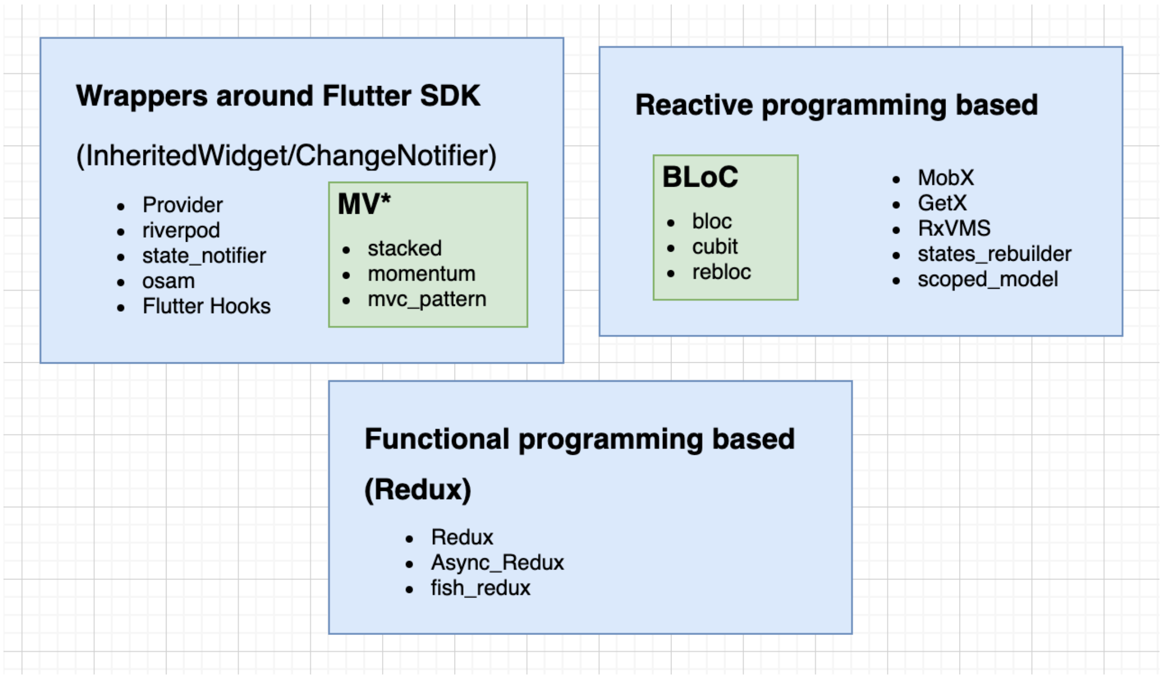


Figure 9. State management approaches categorized

The results do not contradict with the information on the existing state management approaches which can be found on Flutter Website (2020i), which means that it is valid. The objective of categorizing/grouping the approaches is reached.

2.4 Summary

We introduced the context of the mobile development market and explained why Flutter exists and why it is important based on the quantitative research of software development economics lying behind. After reading Section 2.1 even a reader who is not familiar with Flutter should understand why it is an important and promising technology, which explains why the study conducted in this thesis is relevant and important.

Following this, the main concepts of Flutter were introduced and explained in Section 2.2. These concepts include Dart, widgets, state, declarative UI and state management. Without having an idea about these, it is not possible to understand the further study on the main topic.

Finally, a theoretical study on the state management approaches in Flutter was conducted in Section 2.3. This study produced an important result that is used later in the practical part – a grouped list of the most popular state management approaches (Figure 9). This result didn’t exist before, therefore it is considered as a contribution to the existing information about state management approaches in Flutter.

3 PRACTICAL PART

The goal of this part is to study in more detail each of the popular state management approaches and create the example implementations for each of them in order to be able to compare them, determine the benefits and drawbacks of each approach, and finally find out the situations in which each particular approach can perform the best. In the end we will have a comparison table based on the criteria defined in Section 3.2, and a repository with the demo apps for each approach.

3.1 Determining which approaches will be studied and compared

Table 3 lists twenty approaches to state management. This is too much to study in one thesis work without making it too long or missing some important details.

Therefore, I think that the approaches listed on Flutter Website (2020i) should definitely be studied as they were collected by the Flutter team.

These approaches include *setState* (ephemeral state), *InheritedWidget* (Flutter API), *Provider* (wrapper around Flutter API), *GetX* (reactive), *BLoC* (reactive - BLoC), *MobX* (reactive) and *Redux* (functional). This covers all groups from Figure 2. We don’t have any approach representing the small MV* subgroup. This subgroup doesn’t need to be covered separately, because after studying the source code of these libraries I figured out that they are just wrappers around setState/InheritedWidget/ChangeNotifier/Consumer with different names. When developers write the code with these libraries, it turns out to have only minor differences from Provider+ChangeNotifier implemented according to the MV* architecture, for instance. I have written the app using stacked, and the source code was almost 100% similar to my implementation of the same app using Provider. This means that this group can be safely left out as its concepts will be described in the first three approaches. In sum, seven approaches are studied in the next chapters, and these approaches cover all existing groups from Section 2.3.3, and at the same time are the most popular approaches according to Table 3.

3.2 Defining the comparison criteria

As we are studying different approaches to do the same thing, we also want to compare these approaches in order to better understand when each approach can be better. The criteria have to be clearly defined before starting to study each approach in more detail.

After watching the video recordings of Google I/O 2019 conference (YouTube 2019) and DartConf 2018 (YouTube 2018) where state management in Flutter was discussed, I came up with the criteria which can be seen in the list below. I have selected these criteria because they were used by the speakers in these conferences who are directly involved in the development of Flutter SDK and therefore are considered a trustworthy source of information. Some of these criteria were used by them directly, while others were only mentioned.

1. *Complexity* – this criterion represents how difficult is the state management approach to understand, read and maintain. This is a very subjective quality. I make an assessment of the perception of the other developers based on the materials that I find online, but this is still a somewhat subjective assessment. The scale goes from *difficult* to *easy*.
2. *Amount of boilerplate code* – this criterion shows how much extra code has to be written in order to achieve the same goal compared to other approaches. This parameter is important only when several approaches are compared, since it is relative. We assess this for each approach in relation to other approaches, and the scale includes *a little*, *average* and *a lot*.
3. *Code generation* – this criterion tells whether the approach involves any build-time code generation. Developers have different opinions regarding the code generation. Some find it good, while others try to avoid it. We are not going to find which opinion is better since it is out of scope of this work. However, this criterion may be helpful for the developers choosing the approach. The assessment is a simple *yes* or *no*.
4. *"Time travel" support* – this criterion shows whether a feature that allows easily going through the previous states and then come back is supported by the approach. The assessment is a simple *yes* or *no*.
5. *Scalability* – this criterion shows how good the approach saves its characteristics (for instance, complexity or amount of boilerplate code) once the app gets bigger and has more different modules. It is hard to quantify this parameter, so we have to look for the experiences of other people on that and assess this on a qualitative basis from *bad* to *the best*, having *average* and *good* in between.
6. *Testability* – this criterion is often left out, especially by less experienced developers. Nevertheless, it is still very important, especially for the big projects where testing is an extremely important stage of software development. This criterion is mostly related to how difficult it is to conduct unit testing, if possible at all. It is also assessed on scale *bad*/*average*/*good* based on the experiences of other developers.

Also, the benefits and drawbacks of each approach should be determined. All this information has to be enough for us to draw a conclusion on when it is the best to use each approach.

3.3 The app for the practical implementations

After thinking about what kind of a practical implementation for each state management approach I should do, I ended up with a conclusion that it should be the same app for each approach. It has to do absolutely the same thing, but state management has to be implemented differently. This is the only way to see the differences between the approaches while also keeping the things simple.

I have set the following criteria for the app:

1. It has to have several pages with bottom navigation, with one of the pages being the Settings which will allow us to see the immediate effect of a state change on one page affecting other pages.
2. It has to have local persistence since this feature is used very often in almost all of the apps.
3. It has to have remote API calls, just GET requests are enough for the demo.

At the same time, the app doesn't have to solve any real-world problem, since we only need to understand how the things work from the technical side.

3.3.1 UI design

The first step which I do before creating any kind of application is creating UI design wireframes. It speeds up the development of the UI later because you have something to refer to. I have created the wireframes (Figure 10) based on the criteria set in Section 3.3.

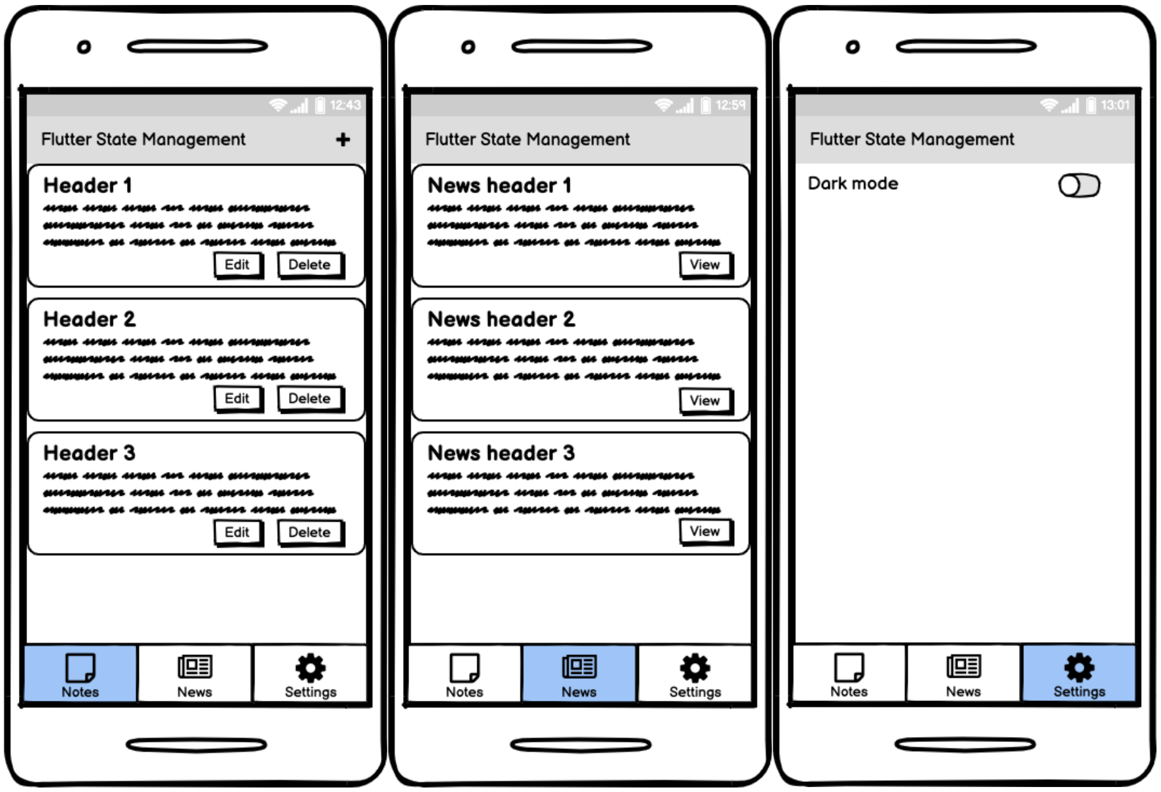
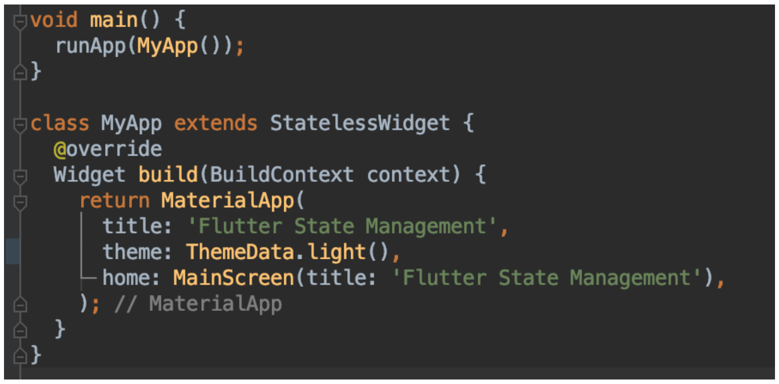


Figure 10. Wireframes of the UI design

The good thing is that the UI can be created once for all apps, as well as the classes from the data access layer (local persistence, remote API calls). The only thing that will change is the state management approach.

3.3.2 UI implementation in Flutter

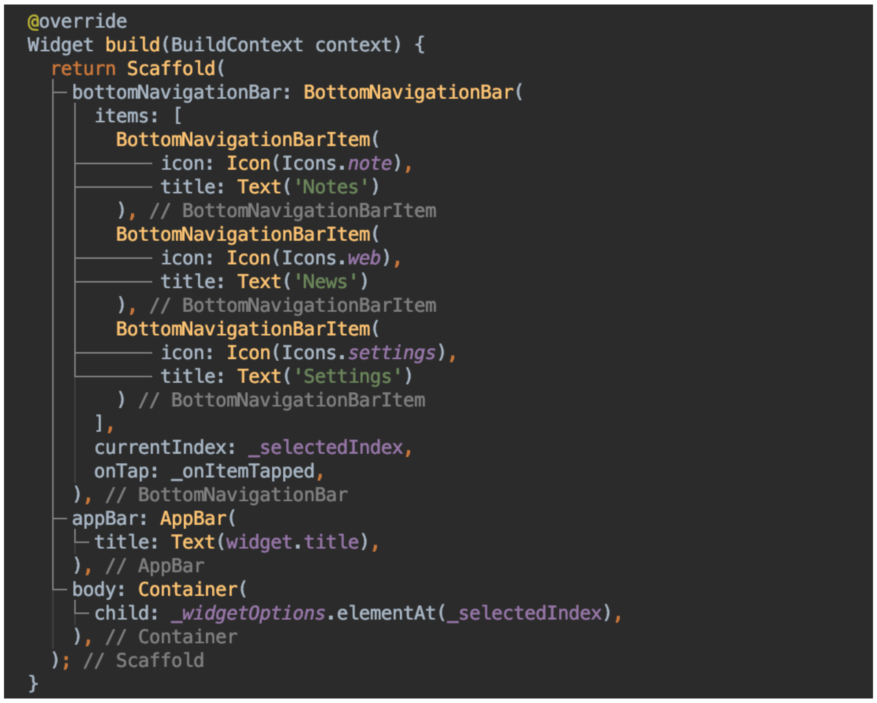
Since the UI is one of the things that will be common for all implementations of the approaches, it is obvious that it can be implemented first. Figure 11 shows how the main file looks like.



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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter State Management',
      theme: ThemeData.light(),
      home: MainScreen(title: 'Flutter State Management'),
    ); // MaterialApp
  }
}
```

Figure 11. main.dart

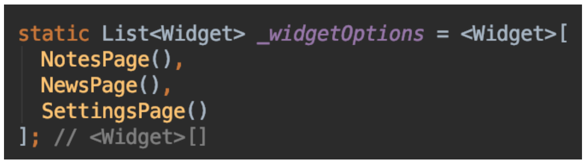
It simply returns the MaterialApp with the title, theme and home properties. MainScreen is the actual entry point of the UI code of the application. Figure 12, in turn, shows that MainScreen itself contains the code for the bottom navigation bar.



```
@override
Widget build(BuildContext context) {
  return Scaffold(
    bottomNavigationBar: BottomNavigationBar(
      items: [
        BottomNavigationBarItem(
            icon: Icon(Icons.note),
            title: Text('Notes')
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
            icon: Icon(Icons.web),
            title: Text('News')
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
            icon: Icon(Icons.settings),
            title: Text('Settings')
        ) // BottomNavigationBarItem
      ],
      currentIndex: _selectedIndex,
      onTap: _onItemTapped,
    ), // BottomNavigationBar
    appBar: AppBar(
      title: Text(widget.title),
    ), // AppBar
    body: Container(
      child: _widgetOptions.elementAt(_selectedIndex),
    ), // Container
  ); // Scaffold
}
```

Figure 12. BottomNavigationBar implementation

Based on the selected item, one of the widgets is returned in the body, as shown in Figure 13.



```
static List<Widget> _widgetOptions = <Widget>[
  NotesPage(),
  NewsPage(),
  SettingsPage()
]; // <Widget>[]
```

Figure 13. Widget options for BottomNavigationBar

Each of these widgets represents one of the actual pages with the corresponding content. I am not going to talk about the UI implementation more, because the goal of this thesis is to study the state management approaches, and not to learn the basics of Flutter. Figure 14 shows how each page of bottom navigation bar looks like on a real device.

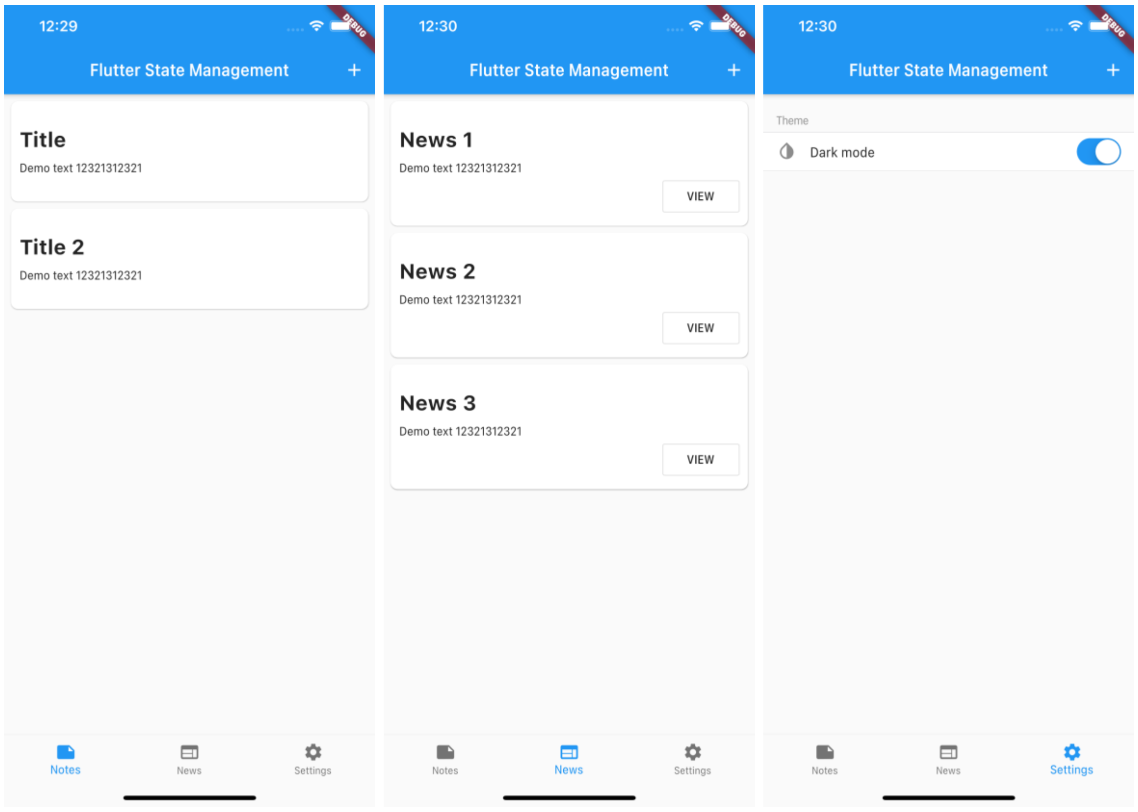


Figure 14. User interface of the app

The first two screens are just scrollable lists with material cards, while the last one is a generic settings screen. Figure 15 shows “add/edit note” and “view news item” screens. I have tried to keep the layout as simple as possible in order to focus on state management instead.

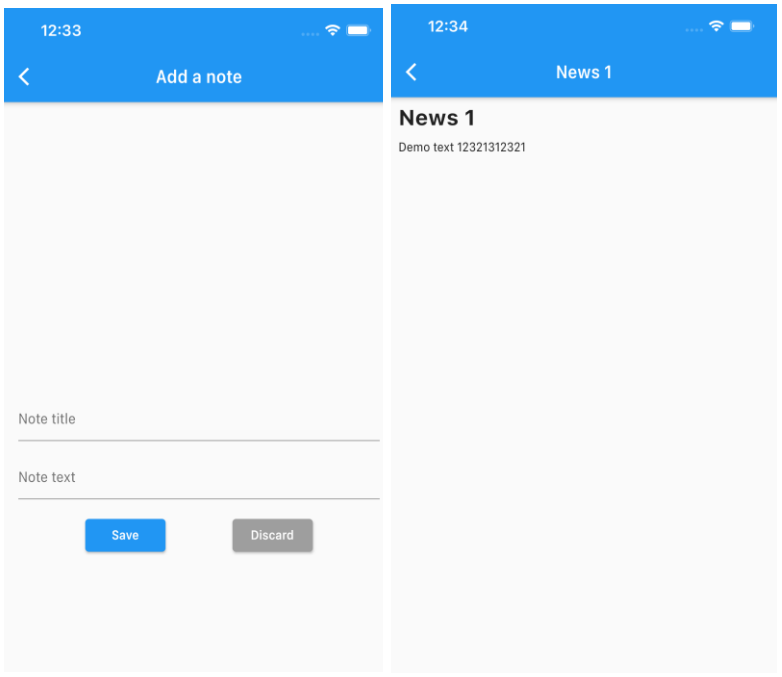


Figure 15. "Add/edit a note" and "View news item" screens

As Figures 14 and 15 showed, the application has five screens in total:

1. Notes list
2. News list
3. Settings
4. Add/edit note
5. View news item

The user interface implementation finishes here, and the next step is to implement the data access layer.

3.3.3 Data access implementation in Flutter

The application has three main features which involve some kind of data access:

1. Notes – local persistence with CRUD features.
2. News – accessing remote API.
3. Settings – also requires some kind of local persistence if we want the settings to be saved.

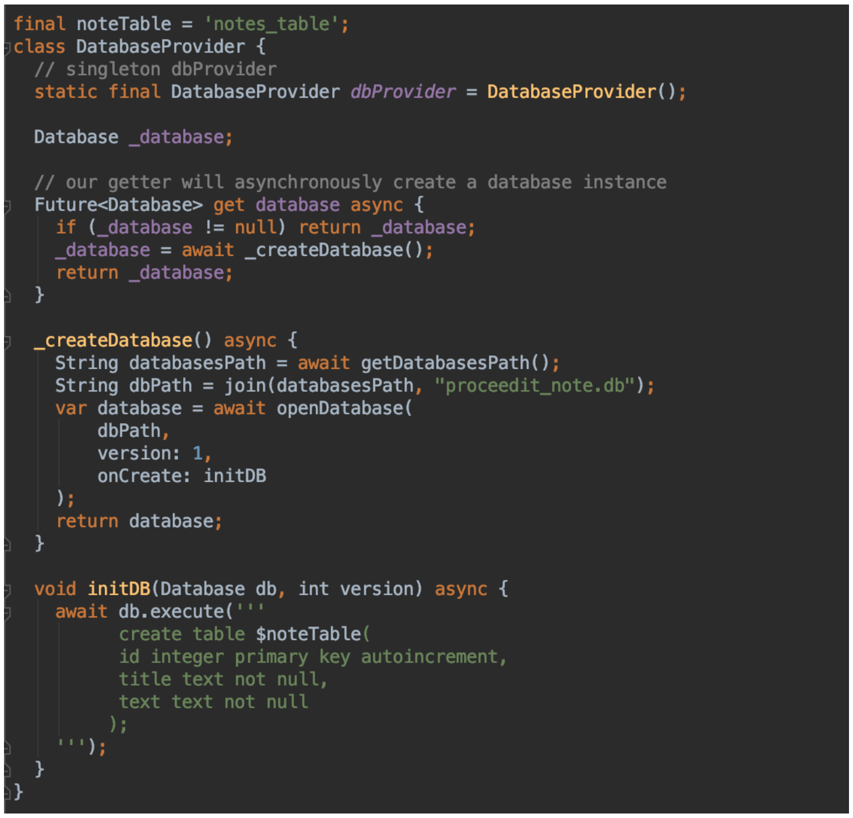
The first feature will be implemented with the help of *sqflite* package, which is a plugin for SQLite, a self-contained, high reliability, embedded, SQL database engine (Pub.dev 2020c). The second feature will be implemented using *retrofit* package which automatically generates the required *dio* client, which is a powerful HTTP client for Dart, which supports Interceptors, Global configuration, FormData, Request Cancellation, File downloading, Timeout etc. (Pub.dev 2020d) The third feature will be implemented with the help of *shared_preferences*, which was created by the Flutter team as an official wrapper for platform-specific storage for simple data (NSUserDefaults on iOS, SharedPreferences on Android).

I will use the repository pattern in order to implement all the data access related features, because it significantly improves the quality of the app architecture, testability, reusability of the code, and embraces the separation of concerns principles. **Repository** modules handle data operations. They provide a clean API so that the rest of the app can retrieve this data easily. They know where to get the data from and what API calls to make when data is updated. You can consider repositories to be mediators between different data sources, such as persistent models, web services, and caches. (Android developers 2020.)

Notes implementation with sqflite

The first thing to do is, obviously, to create a data model class, sometimes called DTO (data transfer object). It has to describe the note – what fields it has and may also include some features to convert the object of this class to JSON or to construct it from JSON, since sqflite uses JSON to store the data. I have defined only 3 fields – id, title and text, as that's all we need for a note.

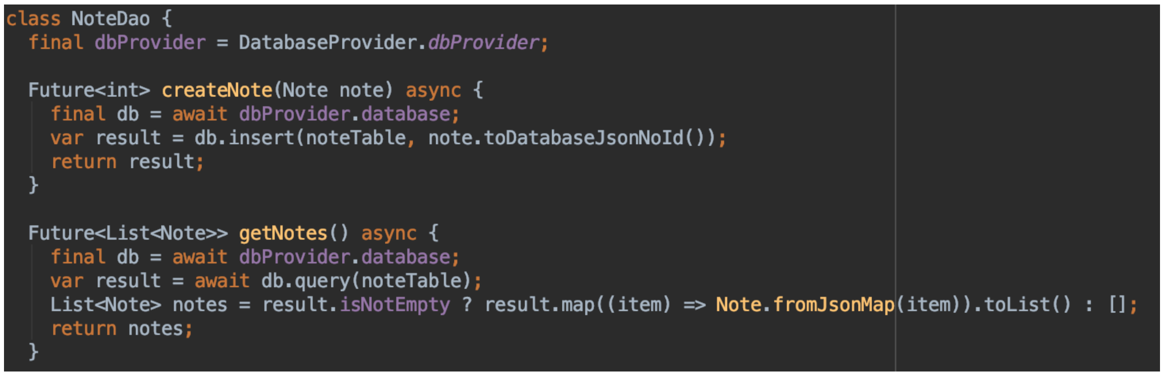
After this, I have created the DatabaseProvider class which uses the singleton pattern in Dart. It means that whenever an instance of this class is accessed anywhere in the code, the same object is always returned. If the object doesn't exist, it is automatically instantiated when accessed first time. The code can be seen in Figure 16.



```
final noteTable = 'notes_table';
class DatabaseProvider {
  // singleton dbProvider
  static final DatabaseProvider dbProvider = DatabaseProvider();

  Database _database;

  // our getter will asynchronously create a database instance
  Future<Database> get database async {
    if (_database != null) return _database;
    _database = await _createDatabase();
    return _database;
  }

  _createDatabase() async {
    String databasesPath = await getDatabasesPath();
    String dbPath = join(databasesPath, "proceedit_note.db");
    var database = await openDatabase(
        dbPath,
        version: 1,
        onCreate: initDB
    );
    return database;
  }

  void initDB(Database db, int version) async {
    await db.execute('''
          create table $noteTable(
          id integer primary key autoincrement,
          title text not null,
          text text not null
         );
    ''');
  }
}
```

Figure 16. DatabaseProvider code

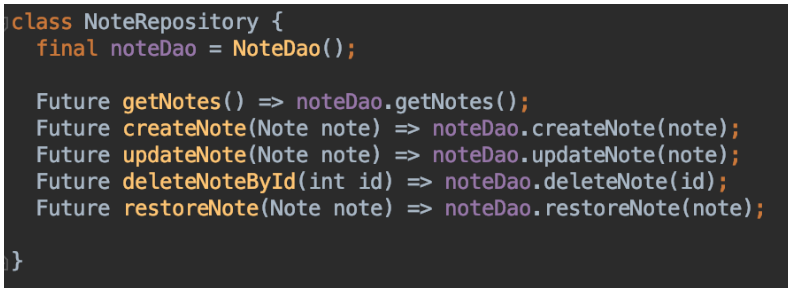
Following this, I have created a DAO (Data Access Object), which represents a pattern that provides an abstract interface to some type of database or other persistence mechanism (Wikipedia 2020d). This class requests an instance of DatabaseProvider, and then implements the CRUD functionality. An example of creating a note and getting a list of all notes can be seen in Figure 17.



```
class NoteDao {
  final dbProvider = DatabaseProvider.dbProvider;

  Future<int> createNote(Note note) async {
    final db = await dbProvider.database;
    var result = db.insert(noteTable, note.toDatabaseJsonNoId());
    return result;
  }

  Future<List<Note>> getNotes() async {
    final db = await dbProvider.database;
    var result = await db.query(noteTable);
    List<Note> notes = result.isNotEmpty ? result.map((item) => Note.fromJsonMap(item)).toList() : [];
    return notes;
  }
```

Figure 17. DAO code

Finally, a repository can be created. It is very simple in this case, since there is only one data source (SQLite database), but it is still a good practice in case of any possible changes in the future. Figure 18 shows the code of the repository.



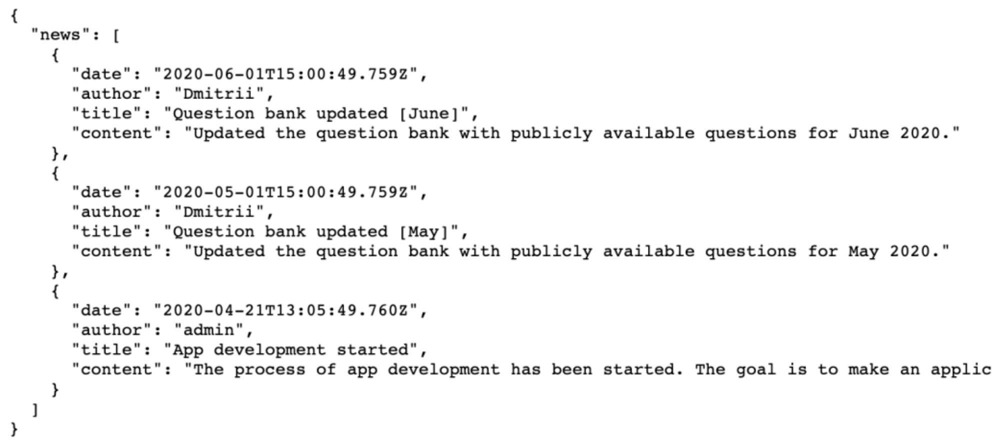
```
class NoteRepository {
  final noteDao = NoteDao();

  Future getNotes() => noteDao.getNotes();
  Future createNote(Note note) => noteDao.createNote(note);
  Future updateNote(Note note) => noteDao.updateNote(note);
  Future deleteNoteById(int id) => noteDao.deleteNote(id);
  Future restoreNote(Note note) => noteDao.restoreNote(note);

}
```

Figure 18. NoteRepository code

In our case, the repository simply has an instance of DAO and mappings of repository methods to the DAO methods.

News implementation with retrofit

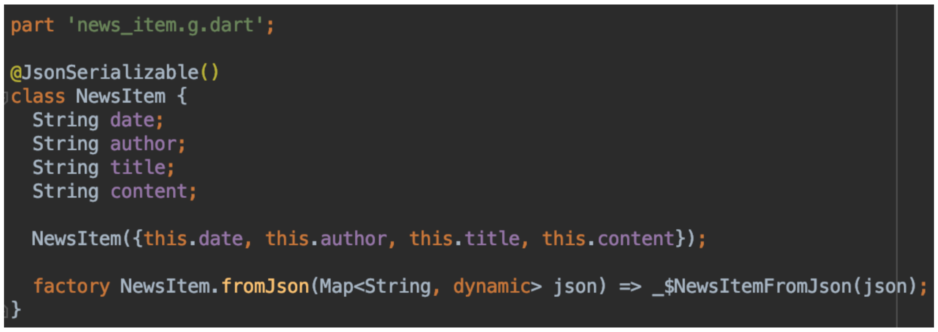
The same procedure as in the previous section should be used here. Firstly, a DTO has to be created. I am using the same API that I have already created for one of my apps, and Figure 19 demonstrates its structure.



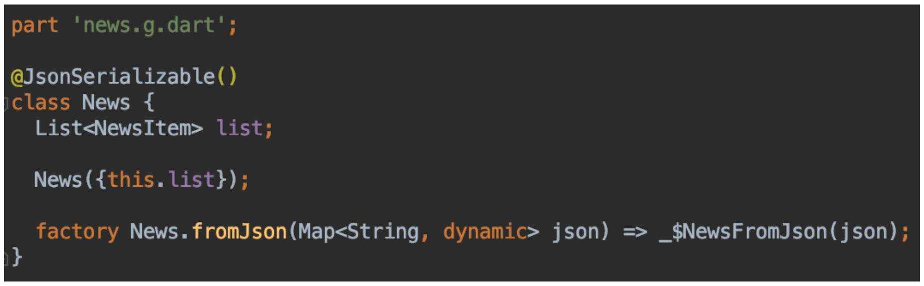
```
{
  "news": [
    {
      "date": "2020-06-01T15:00:49.759Z",
      "author": "Dmitrii",
      "title": "Question bank updated [June]",
      "content": "Updated the question bank with publicly available questions for June 2020."
    },
    {
      "date": "2020-05-01T15:00:49.759Z",
      "author": "Dmitrii",
      "title": "Question bank updated [May]",
      "content": "Updated the question bank with publicly available questions for May 2020."
    },
    {
      "date": "2020-04-21T13:05:49.760Z",
      "author": "admin",
      "title": "App development started",
      "content": "The process of app development has been started. The goal is to make an applic
    }
  ]
}
```

Figure 19. News API

Here we can see that there is a JSON array called “news” which contains the objects that have “date”, “author”, “title” and “content” fields. This means that to implement the things properly, there should be two DTOs – one for each particular object, and another one for the array. These two models can be seen in Figure 20. I am using json_serializable package recommended by the Flutter team in order to handle the JSON serialization.



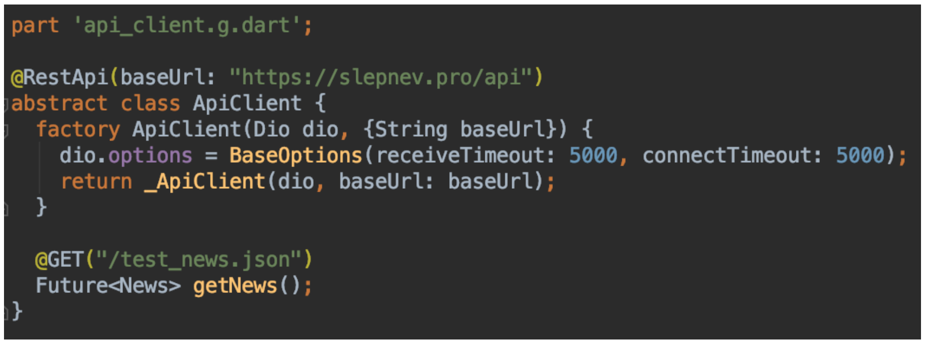
```
part 'news_item.g.dart';

@JsonSerializable()
class NewsItem {
  String date;
  String author;
  String title;
  String content;

  NewsItem({this.date, this.author, this.title, this.content});

  factory NewsItem.fromJson(Map<String, dynamic> json) => _$NewsItemFromJson(json);
}
```



```
part 'news.g.dart';

@JsonSerializable()
class News {
  List<NewsItem> list;

  News({this.list});

  factory News.fromJson(Map<String, dynamic> json) => _$NewsFromJson(json);
}
```

Figure 20. Two DTOs for retrieving the data from the API

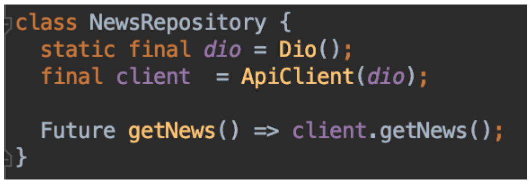
The next step is to create the Retrofit API client class. The code can be seen in Figure 21. Most of the things are generated automatically, I simply had to provide the base URL of the API, some timeout options, and then the actual endpoints and methods with the corresponding return type. We only have one endpoint, and the return type of the data is News (the ‘bigger’ DTO defined in Figure 20).



```dart
part 'api_client.g.dart';

@RestApi(baseUrl: "https://slepnev.pro/api")
abstract class ApiClient {
  factory ApiClient(Dio dio, {String baseUrl}) {
    dio.options = BaseOptions(receiveTimeout: 5000, connectTimeout: 5000);
    return _ApiClient(dio, baseUrl: baseUrl);
  }

  @GET("/test_news.json")
  Future<News> getNews();
}
```

Figure 21. Retrofit API client

Finally, a repository can be created just like in the previous section. The code can be seen in Figure 22.



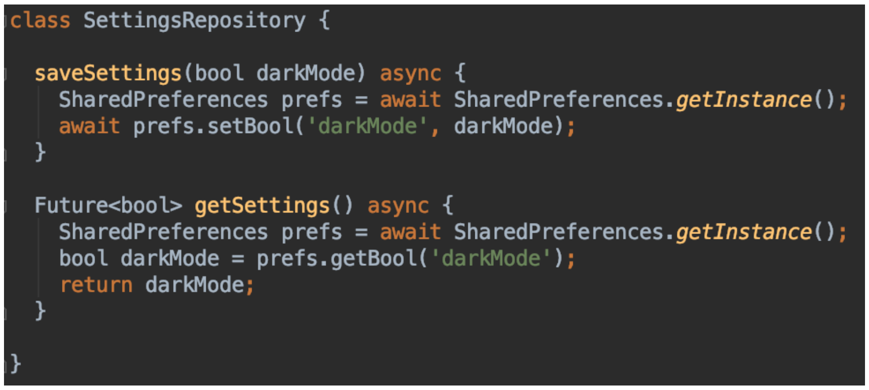
```dart
class NewsRepository {
  static final dio = Dio();
  final client  = ApiClient(dio);

  Future getNews() => client.getNews();
}
```

Figure 22. NewsRepository code

In the repository, a Dio client is instantiated as a static object, after that an ApiClient from Figure 21 is instantiated with the Dio client created before. Finally, there is a getNews method which simply calls the corresponding method in the Retrofit client.

Settings implementation with shared_preferences

Due to the high level of abstraction of shared_preferences library, and to the fact that we only have one property in Settings which is a Boolean value controlling whether the dark or light theme should be used, there is no need to create DTOs or any classes other than the repository itself. The code can be seen in Figure 23.



```
class SettingsRepository {

  saveSettings(bool darkMode) async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    await prefs.setBool('darkMode', darkMode);
  }

  Future<bool> getSettings() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    bool darkMode = prefs.getBool('darkMode');
    return darkMode;
  }

}
```

Figure 23. SettingsRepository code

The SettingsRepository only contains two methods: saveSettings() for saving the Boolean value and getSettings() for obtaining it. The SharedPreferences.getInstance() method always returns the same instance of shared preferences, and therefore there is no need to implement the singleton pattern as it is already implemented under the hood. Because all the code that is shared across the different implementations of the app is now written (UI and data access layer), only the business logic part is left. This is exactly about state management, so we can finally move on to studying the approaches.

3.4 Studying the approaches

This part studies and compares the approaches based on the criteria defined before. The criteria can be found in Section 3.2. The approaches that are studied further were selected in Section 3.1 in such a way that all groups of state management approaches are covered.

3.4.1 setState()

setState() is the low-level approach to use for widget-specific, ephemeral state (Flutter Website 2020i). Ephemeral state is defined and explained in Section 2.3.3. The main focus of this thesis is managing the app state and not the ephemeral state since ephemeral state is usually managed simply using setState. However, we cannot continue studying the app state management approaches without talking about setState, since it is one of the core Flutter features, and due to the flexibility of the framework it can even be used for managing the app state.

Flutter has two types of widgets: stateless and stateful. Stateless widgets are static, their ephemeral state does not change during the execution of the program. Stateful widgets, on the other hand, have a corresponding state object which is dynamic and can be changed.

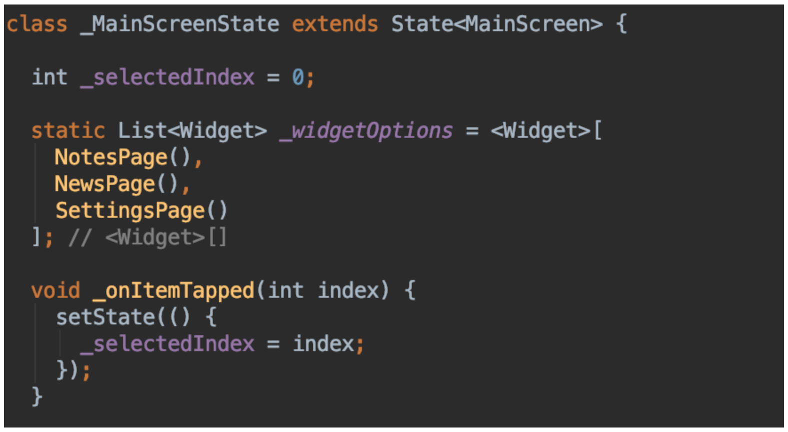
According to Flutter Website (2020l), a stateful widget is implemented by two classes: a subclass of StatefulWidget and a subclass of State. The State class contains the widget’s mutable state and the widget’s build() method. When the widget’s state changes, the state object calls setState(), telling the framework to redraw the widget based on the new state.

Practical implementation

Firstly, let’s see how to work with the ephemeral state using setState, because that’s what is should be used for. A good and slightly more advanced than the regular “counter” Hello Worlds of Flutter is the bottom navigation which is already implemented in the starter app.

If we take a look at Figure 12 again, we can see that the currentIndex of bottom navigation is defined by some _selectedIndex variable, and the same variable is also used to select which screen to show in the body. Another thing is that there is some _onItemTapped function which is provided to the onTap property. I didn’t mention what are these things in the UI part, but now it is suitable time to do so.

_selectedIndex is a variable that represents state, it is dynamic, and the framework keeps track of it, so whenever it is changed using setState, Flutter checks whether there is a need to rebuild some widgets which rely on the value of this variable. Figure 24 shows that it is simply a variable defined in the State part of the StatefulWidget. It is equal to 0 by default, and whenever onItemTapped callback is called, setState updates the _selectedIndex with the index of the tapped item in the bottom navigation bar.



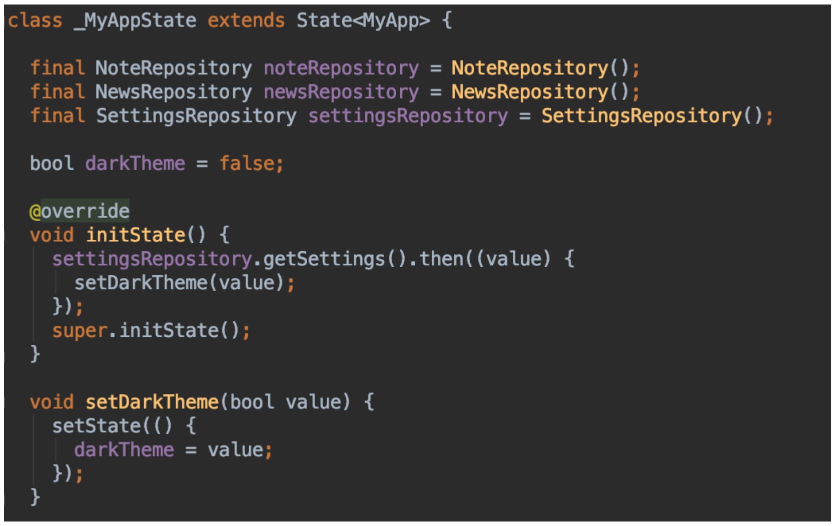
```dart
class _MainScreenState extends State<MainScreen> {

  int _selectedIndex = 0;

  static List<Widget> _widgetOptions = <Widget>[
    NotesPage(),
    NewsPage(),
    SettingsPage()
  ]; // <Widget>[]

  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
    });
  }
```

Figure 24. Ephemeral state management with setState()

For instance, the user is by default seeing the notes screen, and the index of the "Notes" tab in bottom navigation is 0. If he or she clicks on the "News" tab (which has an index of 1), the _onItemTapped(1) is called, which triggers the setState on _selectedIndex and its value is updated from 0 to 1. The framework then checks where this state variable is used. In this case, it is used to highlight the currently active tab in the bottom navigation view and to return the corresponding page from _widgetOptions. The framework then rebuilds the bottom navigation widget with the new state, so that the "News" tab becomes highlighted and the NewsPage widget is returned in the body. And that's how the ephemeral state of a single widget (MainScreen) is supposed to be managed. Refer to Figure 14 for better visual understanding.

It should be clear with the ephemeral state now, but what about the app state management with setState? Actually, it is possible, but generally not recommended. Implementing it should show on practice why exactly it is not the best idea to do so.

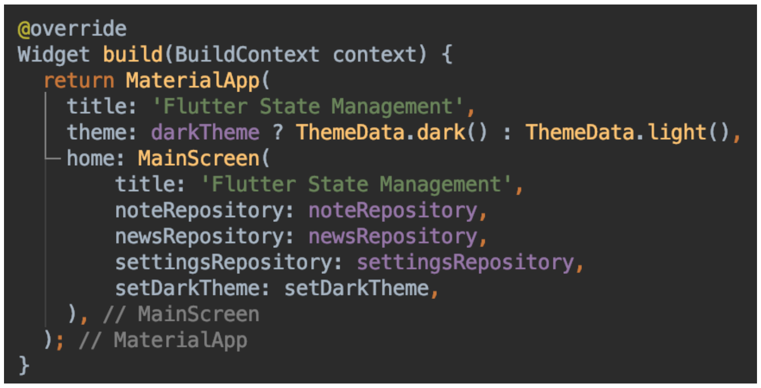
First of all, we need to instantiate the repositories somewhere. The topmost MyApp widget is the right place to do it, since we are then able to pass the repositories down the widget tree to anywhere. Also, the theme (light or dark) is set in this widget, so we have to make it stateful and add a state parameter that controls the theme – I have added a Boolean darkTheme value. I have also created a setDarkTheme function which will be a good example of how to update state of parent widget from the child widget. All these things can be seen in Figure 25.



```dart
class _MyAppState extends State<MyApp> {

  final NoteRepository noteRepository = NoteRepository();
  final NewsRepository newsRepository = NewsRepository();
  final SettingsRepository settingsRepository = SettingsRepository();

  bool darkTheme = false;

  @override
  void initState() {
    settingsRepository.getSettings().then((value) {
      setDarkTheme(value);
    });
    super.initState();
  }

  void setDarkTheme(bool value) {
    setState(() {
      darkTheme = value;
    });
  }
```

Figure 25. MyApp state code

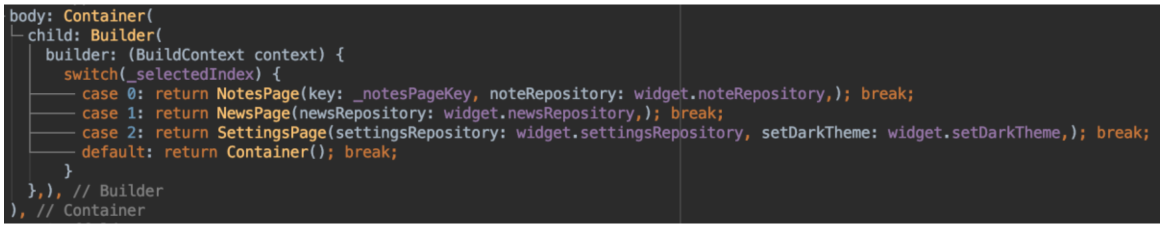
After this, the repositories have to be passed as parameters to the MainScreen which is responsible for the bottom navigation. Also, I am passing setDarkTheme function as it will be needed later (Figure 26). The theme selection depends on the darkTheme state variable value.



```dart
@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Flutter State Management',
    theme: darkTheme ? ThemeData.dark() : ThemeData.light(),
    home: MainScreen(
        title: 'Flutter State Management',
        noteRepository: noteRepository,
        newsRepository: newsRepository,
        settingsRepository: settingsRepository,
        setDarkTheme: setDarkTheme,
    ), // MainScreen
  ); // MaterialApp
}
```

Figure 26. build() method of MyApp

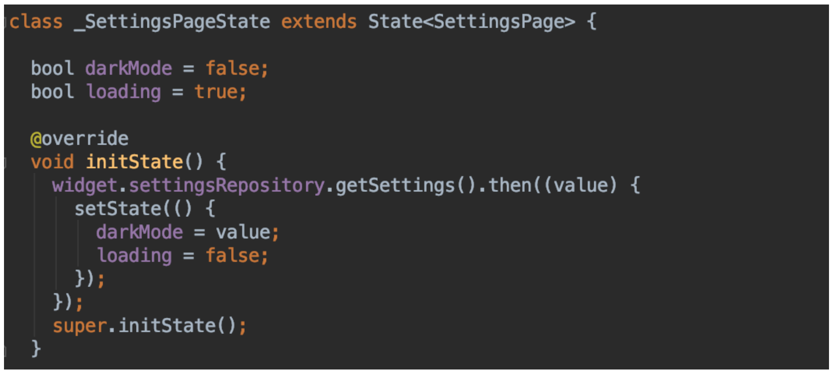
Following this, I had to pass each repository from the MainScreen to the corresponding page, as well as the setDarkTheme function. The code can be seen in Figure 27.



```
body: Container(
  child: Builder(
    builder: (BuildContext context) {
      switch(_selectedIndex) {
        case 0: return NotesPage(key: _notesPageKey, noteRepository: widget.noteRepository,); break;
        case 1: return NewsPage(newsRepository: widget.newsRepository,); break;
        case 2: return SettingsPage(settingsRepository: widget.settingsRepository, setDarkTheme: widget.setDarkTheme,); break;
        default: return Container(); break;
      }
  },), // Builder
), // Container
```

Figure 27. Passing the repositories down the tree from MainScreen

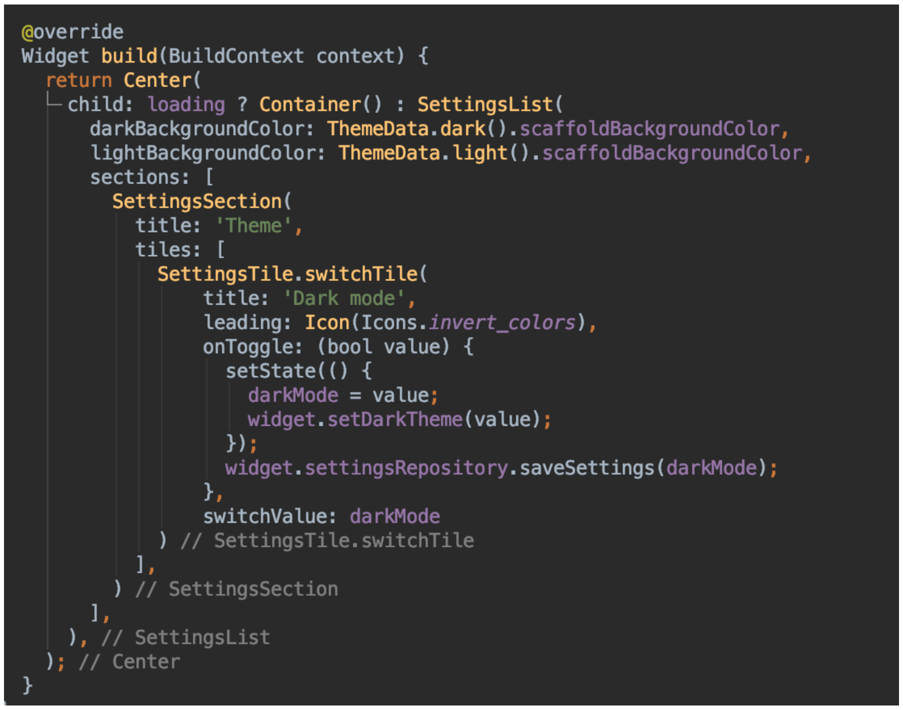
After this, for instance, in the settings page, the loading and darkMode state variables have to be defined. The code is shown in Figure 28. In initState() function, which is called every time the widget is built first, we can get the settings from the repository. The repository in this case is just an abstraction on top of SharedPreferences.



```
class _SettingsPageState extends State<SettingsPage> {

  bool darkMode = false;
  bool loading = true;

  @override
  void initState() {
    widget.settingsRepository.getSettings().then((value) {
      setState(() {
        darkMode = value;
        loading = false;
      });
    });
    super.initState();
  }
```

Figure 28. SettingsPage initState()

When the widget is built, the list of settings is only displayed when loading variable is false, and loading variable becomes false only after the saved settings are obtained from the shared preferences. The callback which is tied to the switch button toggle calls setState which updates the darkMode local state variable to the corresponding value, and also calls the setDarkTheme function passing the new value as a parameter. This function comes to the SettingsPage from MyApp through the MainScreen. It is possible because functions in Dart can be treated as objects of type Function. Finally, the settings are saved by calling the saveSettings function from the settings repository (Figure 29).

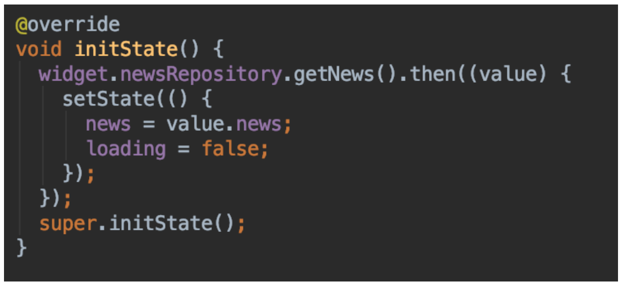


```
@override
Widget build(BuildContext context) {
  return Center(
    child: loading ? Container() : SettingsList(
      darkBackgroundColor: ThemeData.dark().scaffoldBackgroundColor,
      lightBackgroundColor: ThemeData.light().scaffoldBackgroundColor,
      sections: [
        SettingsSection(
          title: 'Theme',
          tiles: [
            SettingsTile.switchTile(
                title: 'Dark mode',
                leading: Icon(Icons.invert_colors),
                onToggle: (bool value) {
                  setState(() {
                    darkMode = value;
                    widget.setDarkTheme(value);
                  });
                  widget.settingsRepository.saveSettings(darkMode);
                },
                switchValue: darkMode
            ) // SettingsTile.switchTile
          ],
        ) // SettingsSection
      ],
    ), // SettingsList
  ); // Center
}
```

Figure 29. SettingsPage build() method with setState() state management

This all results in the fact that whenever a Dark Theme switch in the Settings page is tapped, the app theme changes immediately, and this setting is saved to the persistent storage.

For the news and notes pages the initial set up demonstrated in Figure 28 is almost the same. The difference is only in the fact that instead of Boolean darkMode setting, a List<NewsItem> or a List<Note> is defined. In the news page, the only place where the data has to be loaded is the initState() method (Figure 30).

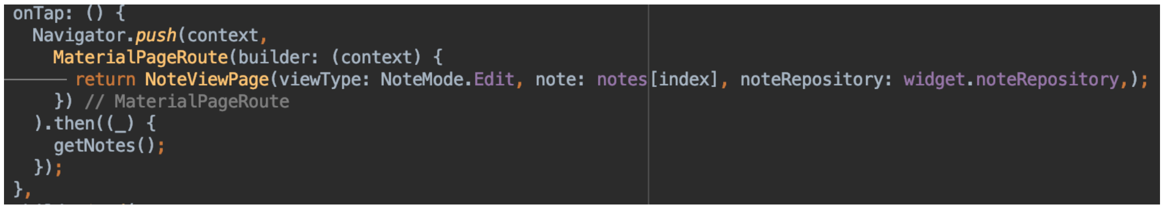


```
@override
void initState() {
  widget.newsRepository.getNews().then((value) {
    setState(() {
      news = value.news;
      loading = false;
    });
  });
  super.initState();
}
```

Figure 30. initState() of NewsPage

When state is initialized, we are requesting the repository to get the news, and then waiting for the result of the Future, and updating the news list with the loaded news and loading variable with false value. It is pretty straightforward how everything is rendered then in the build() method because we are then simply working with the variable of type List<NewsItem>.

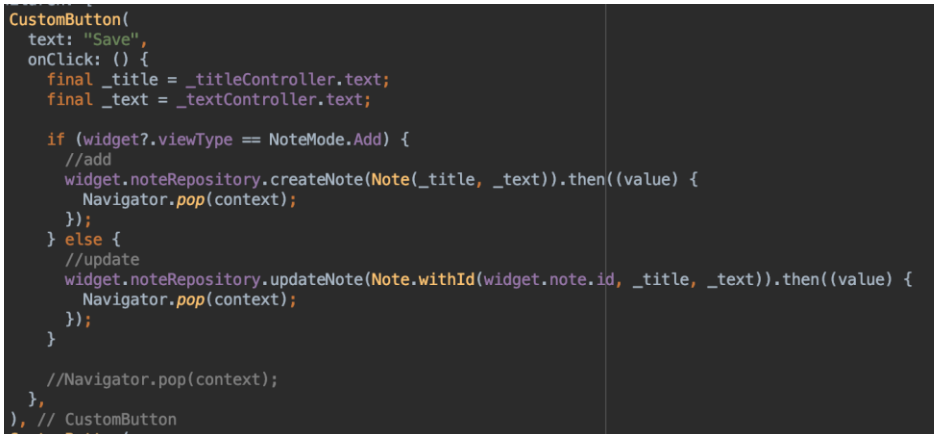
The difference for the notes part is that we need to implement full CRUD functionality. The “read” part is done the same way as with settings and news, and for “create” part, a NoteViewPage in NoteMode.Add is opened. The noteRepository is passed to this page as well, and then the entered note is simply passed to the createNote function of the repository, and the notes are refreshed when the navigation Future resolves. The same thing is done for “update” and “delete” parts, but the note itself is passed to the NoteViewPage, and the viewType is set to NoteMode.Edit, as shown in Figure 31.



```
onTap: () {
  Navigator.push(context,
    MaterialPageRoute(builder: (context) {
      return NoteViewPage(viewType: NoteMode.Edit, note: notes[index], noteRepository: widget.noteRepository,);
    }) // MaterialPageRoute
  ).then((_) {
    getNotes();
  });
},
```

Figure 31. Opening a screen to edit/delete the note

The “Save” button code from NoteViewPage can be seen in Figure 32, and it illustrates how the notes are created or updated.



```
CustomButton(
  text: "Save",
  onClick: () {
    final _title = _titleController.text;
    final _text = _textController.text;

    if (widget?.viewType == NoteMode.Add) {
      //add
      widget.noteRepository.createNote(Note(_title, _text)).then((value) {
        Navigator.pop(context);
      });
    } else {
      //update
      widget.noteRepository.updateNote(Note.withId(widget.note.id, _title, _text)).then((value) {
        Navigator.pop(context);
      });
    }

    //Navigator.pop(context);
  },
), // CustomButton
```

Figure 32. Creating and updating a note

Because the “Delete” button works similarly, there is no need to paste it here. Let’s see what is good and bad about this approach to state management.

Benefits and drawbacks

What is good about this approach based on the experience that we had in this section?

1. It is quite easy to understand if you simply learn how the Flutter framework works and what actually happens when setState() is called.
2. There is no need to use any external libraries, since it is a built-in Flutter method.

And that is basically it, this approach doesn't provide other significant benefits. At the same time, it has very important drawbacks which make it a bad solution for the app state management:

1. Developers have to manually provide any dependencies to the children from the ancestors. For instance, to reach the edit screen, the note repository had to go through the following path as a constructor parameter: MyApp -> MainScreen -> NotesPage -> NoteViewPage. The more complex the app becomes, the more layers of widgets it gets, and the more dependencies start to appear. This all makes the code difficult to read and maintain and adds a lot of extra work.
2. There is no separation of views and business logic – the repositories are called directly from the view classes.
3. It has difficult error handling (not implemented in the demo).

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 4.

Table 4. setState() assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
easy	a lot	no	no	bad	bad

This approach is easy because the only thing it uses is the built-in Flutter SDK method. However, the amount of boilerplate code grows as the app scales, which also results in bad scalability. It becomes more difficult to develop and maintain the app as it gets bigger. The absence of separation of UI code and business logic makes it difficult to test the app with automatic unit tests.

When to use

To conclude the results regarding the setState() state management approach, it seems to be perfect for ephemeral state management, but almost not suitable for any kinds of app state management. It should only be used for very simple projects / demos / prototypes, or in cases when there is no complex hierarchical structure of widgets (only one or two layers). In other cases, the scalability problems are too significant.

3.4.2 InheritedWidget

InheritedWidget is the low-level approach used to communicate between ancestors and children in the widget tree. This is what provider and many other approaches use under the hood. (Flutter Website 2020i.)

InheritedWidget class is a base class for widgets that is able to efficiently propagate information down the tree (Flutter API Docs 2020). Whenever the inherited widget is changed (since it is immutable – it cannot be changed, so it is better to say “whenever it is replaced with a new instance”), the consumer of the data that was affected is rebuilt.

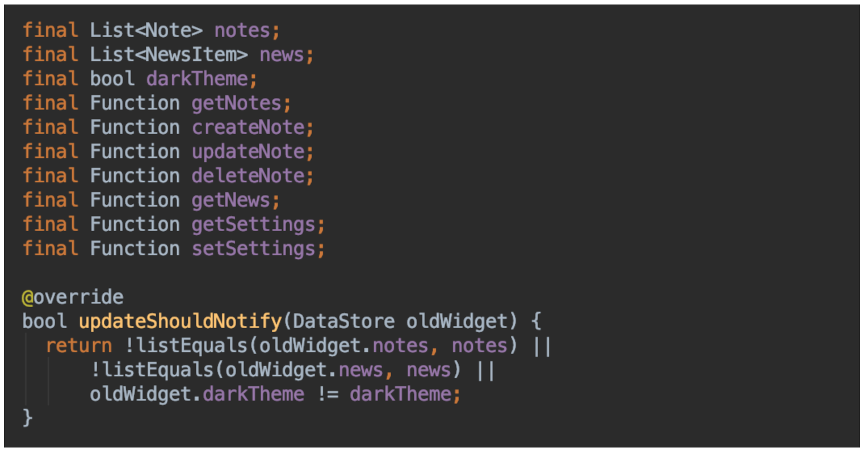
So, based on this description we can already make a conclusion that InheritedWidget is more of a dependency injection rather than a state management mechanism. It helps to get rid of the need to pass the objects through all constructors if we want to access some data from the higher layers of the widget tree in the lower ones. But it doesn’t offer a standalone state management solution (due to the fact that InheritedWidget is immutable), which means that we still have to wrap it in a stateful widget and use setState to mutate the InheritedWidget (basically, to replace it with a new one).

Practical implementation

The most obvious thing to do is just to use InheritedWidget for injecting the repositories where they are needed and use setState on lower level exactly like was done in Section 3.4.1. This is a valid approach, however, it doesn't allow us to see all features of InheritedWidget, so I had to take some time and think how it can be used differently without having almost the same implementation as with setState in the end.

The idea behind this implementation is that we can wrap an inherited widget in a stateful widget which has no UI elements but contains the business logic. I have called the InheritedWidget "DataStore" since its main function is to store the data accessible from all children below this widget. The name which I have chosen for the StatefulWidget wrapper is "ViewModel" as that is the part which contains business logic and connects Views to Models in MVVM architectural pattern. However, the state management approach, about which we are talking has nothing to do with MVVM, and this name is only used as an analogy.

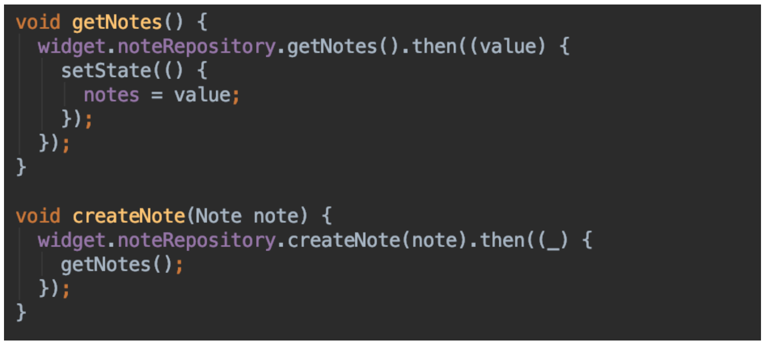
The DataStore class, which extends InheritedWidget, contains the lists of notes and news, darkTheme Boolean value, and references to the ViewModel's functions which can be used to call the corresponding repository operations. It can be seen in Figure 33. There is also an updateShouldNotify method which is used to check whether the widgets that use the values from the InheritedWidget have to be rebuilt. When an InheritedWidget is replaced with a new one, the old and new versions are compared based on the condition set in this method. If the method returns true, then the widgets using data from the InheritedWidget are rebuilt, otherwise – nothing is done.



```dart
final List<Note> notes;
final List<NewsItem> news;
final bool darkTheme;
final Function getNotes;
final Function createNote;
final Function updateNote;
final Function deleteNote;
final Function getNews;
final Function getSettings;
final Function setSettings;

@override
bool updateShouldNotify(DataStore oldWidget) {
  return !listEquals(oldWidget.notes, notes) ||
      !listEquals(oldWidget.news, news) ||
      oldWidget.darkTheme != darkTheme;
}
```

Figure 33. DataStore InheritedWidget

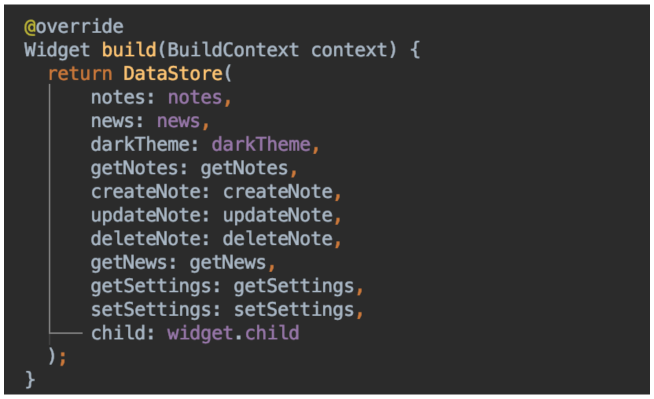
Then there is the ViewModel class which wraps the InheritedWidget. It takes the three repositories and a child (which is the widget that should be below the InheritedWidget) as the constructor parameters. It has its own state variables notes, news and darkTheme, and several methods to update these values (Figure 34).



```dart
void getNotes() {
  widget.noteRepository.getNotes().then((value) {
    setState(() {
      notes = value;
    });
  });
}

void createNote(Note note) {
  widget.noteRepository.createNote(note).then((_) {
    getNotes();
  });
}
```

Figure 34. Methods to get a list of notes and create a new note in the ViewModel

And following this, all these state variables and functions are passed to the InheritedWidget in the build() method of the stateful wrapper, as shown in Figure 35.

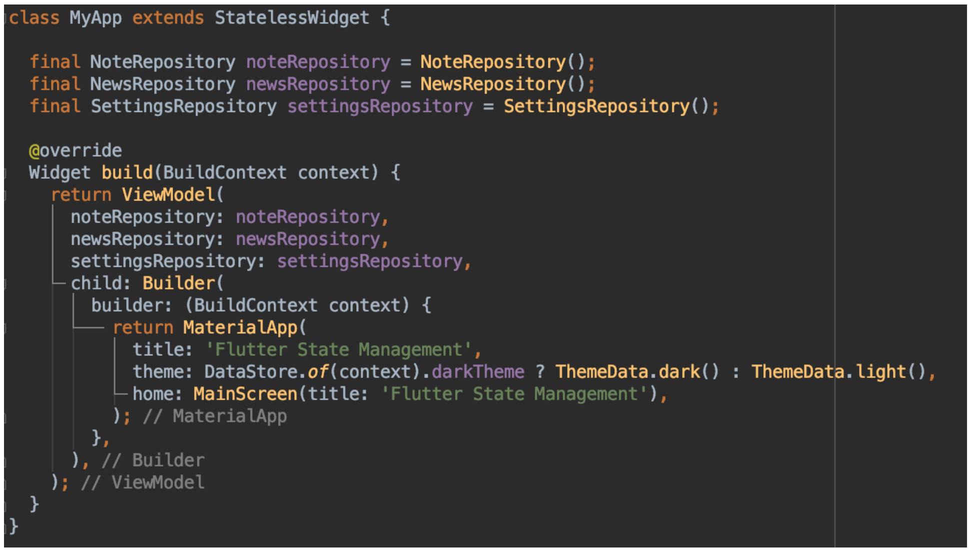


```
@override
Widget build(BuildContext context) {
  return DataStore(
      notes: notes,
      news: news,
      darkTheme: darkTheme,
      getNotes: getNotes,
      createNote: createNote,
      updateNote: updateNote,
      deleteNote: deleteNote,
      getNews: getNews,
      getSettings: getSettings,
      setSettings: setSettings,
      child: widget.child
  );
}
```

Figure 35. build() method of ViewModel

Such set-up gives us an interesting result: we now have access to the notes, news and darkTheme state variables of the ViewModel, as well as to the methods listed in Figure 35. The DataStore InheritedWidget serves as a proxy between the view classes and the ViewModel and also works as a dependency injection and update notifying mechanism. It means that we can call any of the methods (Figure 35) from anywhere in the app, which might cause changes to the notes, news or darkTheme of ViewModel, but since they are the state variables of ViewModel and are passed to the DataStore in the build method, the DataStore will be rebuilt with the new state values, which will cause the updateShouldNotify comparison explained before, and the widgets using the updated data will be rebuilt with the new data.

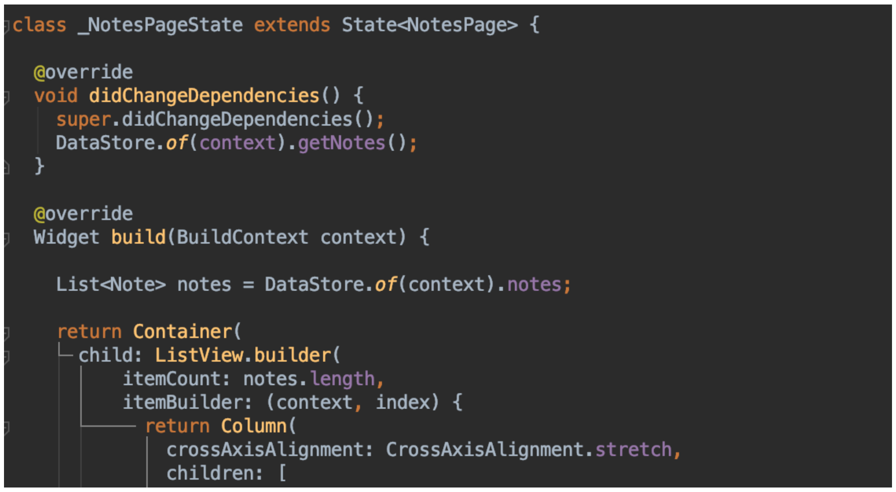
Then the ViewModel should be placed on top of the widget tree (Figure 36). Note that this allows us to use the darkTheme from DataStore to set dark or light theme already in the MaterialApp widget.



```dart
class MyApp extends StatelessWidget {

  final NoteRepository noteRepository = NoteRepository();
  final NewsRepository newsRepository = NewsRepository();
  final SettingsRepository settingsRepository = SettingsRepository();

  @override
  Widget build(BuildContext context) {
    return ViewModel(
      noteRepository: noteRepository,
      newsRepository: newsRepository,
      settingsRepository: settingsRepository,
      child: Builder(
        builder: (BuildContext context) {
          return MaterialApp(
            title: 'Flutter State Management',
            theme: DataStore.of(context).darkTheme ? ThemeData.dark() : ThemeData.light(),
            home: MainScreen(title: 'Flutter State Management'),
          ); // MaterialApp
        },
      ), // Builder
    ); // ViewModel
  }
}
```

Figure 36. MyApp class for the InheritedWidget state management

Further usage of the DataStore is pretty straightforward and similar to what can be seen in Figure 36. We don't need to look into each page of the app, it is enough to simply take a look at the "Notes" page (Figure 37). The getNotes() from ViewModel is triggered by calling the corresponding method of DataStore in didChangeDependencies(), which is called right after initState() – it is needed to populate the list of notes when the page is first opened. Then the list of notes is obtained from DataStore in build() method and used just as a normal list. For example, if we need to add a new note, we simply need to call *DataStore.of(context).createNote(Note(title, text))*.



```dart
class _NotesPageState extends State<NotesPage> {

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    DataStore.of(context).getNotes();
  }

  @override
  Widget build(BuildContext context) {

    List<Note> notes = DataStore.of(context).notes;

    return Container(
      child: ListView.builder(
          itemCount: notes.length,
          itemBuilder: (context, index) {
            return Column(
              crossAxisAlignment: CrossAxisAlignment.stretch,
              children: [
```

Figure 37. Using DataStore in NotesPage

It would be better if we created a separate InheritedWidget with a stateful wrapper (ViewModel) for each page (Notes, News, Settings), but I have decided to keep the things simple and use only one combination of

InheritedWidget+StatefulWidget as it is enough to test and demonstrate the capabilities of this state management approach.

Benefits and drawbacks

Below we can see a list of the most significant benefits of the InheritedWidget state management approach:

1. There is no need to use any external libraries, since it is a Flutter API
2. There is a possibility to separate the views and business logic which improves testability, readability and maintainability of the code
3. There is no need to provide the dependencies manually

Also, it has some drawbacks:

1. It is not obvious how it actually works, and how to write the code that will make this work. From my personal experience, when I was first trying to understand InheritedWidget and use it in a real app, I failed because Flutter resources didn’t make it obvious that InheritedWidget should be wrapped inside a StatefulWidget in order for it to work as a full state management solution.
2. It is easy to accidentally cause infinite loop of updates and rebuilds of some widget by making some unobvious mistakes in the model or in didChangeDependencies.
3. Still there is too much boilerplate code – for one ViewModel we need two classes: InheritedWidget and StatefulWidget wrapper.
4. This method is not actually used by many developers directly because there is a much more popular abstraction over it (Provider). As a result, there is not enough documentation/examples for it.

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 5.

Table 5. InheritedWidget assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
difficult	average	no	no	average	bad

This approach is difficult to study mainly due to the fact that there is no adequate documentation explaining how it works in simple words. It is quite low-level. The amount of boilerplate code is less than in setState, but it is still substantial (two classes for one ViewModel). Scalability is average because it is definitely better than in setState, but still the previously mentioned issues do not allow the app based on this method to scale easily. Testability is not different from setState.

When to use

To conclude the results regarding the InheritedWidget state management approach, it seems that it can be used by experienced developers or by those who are ready to invest their time into studying and understanding how it works, and when they don’t want to rely on an external library like Provider.

3.4.3 Provider

Provider is the recommended state management approach by the Flutter team (Flutter Website 2020i). It is a wrapper around InheritedWidget to make them easier to use and more reusable (Pub.dev 2020e). Flutter team says that this approach is the simplest to understand, and at the same time quite powerful.

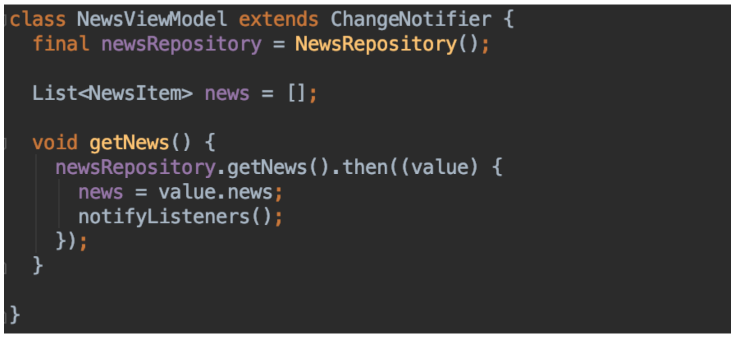
Provider, just like InheritedWidget, is primarily a dependency injection mechanism. In order to notify the listeners about any state changes, another Flutter API class – ChangeNotifier – should be used. However, most often you can see the mentions of “Provider state management”. By that people usually mean “Provider + ChangeNotifier”.

Provider is promised to provide a largely reduced boilerplate over making new classes every time and increased scalability (Pub.dev 2020e). Let’s see if it is true by implementing the demo app using this approach.

Practical implementation

This time I have created three ViewModel classes instead of one since Provider approach has much less boilerplate code which makes it reasonable.

I only show the NewsViewModel here (Figure 38), because two other classes are built similarly. The repository can be instantiated directly in the ViewModel, since the same instance of the ViewModel will be kept on top of the widget tree during the execution of the application, which means that the repository will not be disposed on a widget rebuild. NewsViewModel extends ChangeNotifier, which is an API for dispatching notifications of changes in the class, as the name implies.

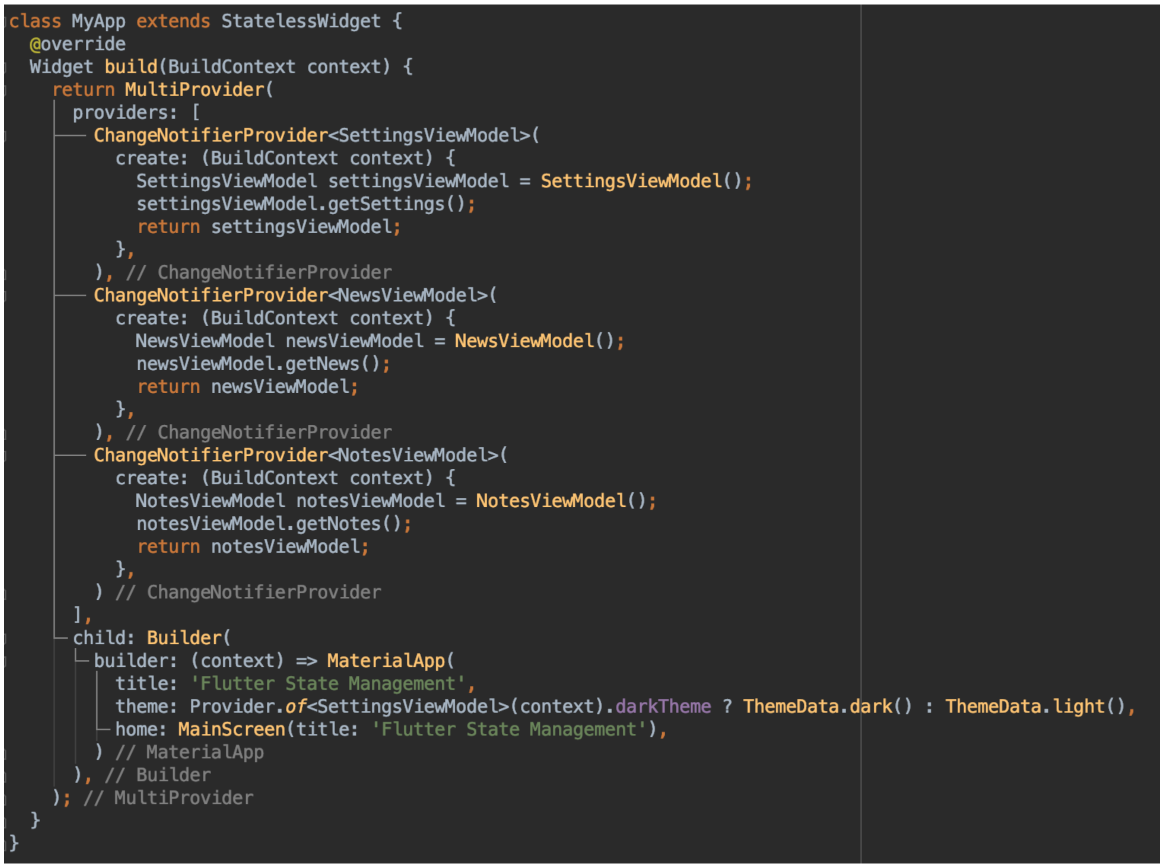


```
class NewsViewModel extends ChangeNotifier {
  final newsRepository = NewsRepository();

  List<NewsItem> news = [];

  void getNews() {
    newsRepository.getNews().then((value) {
      news = value.news;
      notifyListeners();
    });
  }

}
```

Figure 38. NewsViewModel

After creating the ViewModels, they have to be put on top of the widget tree using Provider package. This package has a MultiProvider widget which allows to provide an array of ChangeNotifierProviders, each of which is responsible for creating and then passing down the tree a ViewModel. The *create* property takes a function which should return a ChangeNotifier instance (a ViewModel). The good thing is that it is also possible to do some initial settings there before returning the instance. For instance, we can call a function to retrieve notes/news/settings, which will make them available even before the widgets are drawn on the screen, so the app will look faster than in two previous approaches where we had to first call these methods in didChangeDependencies. Figure 39 shows the code for MyApp class. You can see that the state variables stored in the ViewModels can be accessed immediately from the context which is below the context in which the MultiProvider widget is built.

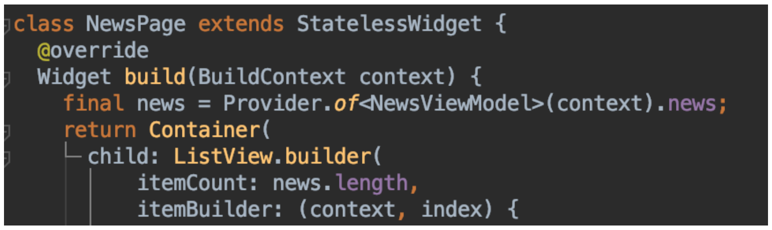
Provider.of<SettingsViewModel>(context).darkTheme gives the dark theme Boolean value from SettingsViewModel and automatically subscribes to its updates, so whenever it changes, the widget using this value will be rebuilt according to the new state.



```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider<SettingsViewModel>(
          create: (BuildContext context) {
            SettingsViewModel settingsViewModel = SettingsViewModel();
            settingsViewModel.getSettings();
            return settingsViewModel;
          },
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<NewsViewModel>(
          create: (BuildContext context) {
            NewsViewModel newsViewModel = NewsViewModel();
            newsViewModel.getNews();
            return newsViewModel;
          },
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<NotesViewModel>(
          create: (BuildContext context) {
            NotesViewModel notesViewModel = NotesViewModel();
            notesViewModel.getNotes();
            return notesViewModel;
          },
        ) // ChangeNotifierProvider
      ],
      child: Builder(
        builder: (context) => MaterialApp(
          title: 'Flutter State Management',
          theme: Provider.of<SettingsViewModel>(context).darkTheme ? ThemeData.dark() : ThemeData.light(),
          home: MainScreen(title: 'Flutter State Management'),
        ) // MaterialApp
      ), // Builder
    ); // MultiProvider
  }
}
```

Figure 39. MyApp code for Provider approach

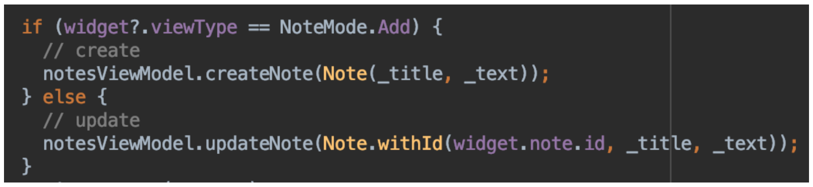
The lists of news/notes/settings are obtained exactly the same way, and they can be used right away (Figure 40). There is no need to make the page widgets stateful because the state management responsibility is fully lifted from them, which is a huge benefit.



```
class NewsPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final news = Provider.of<NewsViewModel>(context).news;
    return Container(
      child: ListView.builder(
          itemCount: news.length,
          itemBuilder: (context, index) {
```

Figure 40. Obtaining state from NewsViewModel in NewsPage

In NoteViewPage, which is used for creating/updating/deleting the notes and doesn't use any mutable state from the NotesViewModel, it is possible to define a *notesViewModel = Provider.of<NotesViewModel>(context, listen: false).* The listen property set to false indicates that we are not willing to subscribe to any updates of the corresponding ViewModel (ChangeNotifier), which means that we are only utilizing the dependency injection part of the Provider approach here. It allows us to call the methods which will cause the ViewModel state to change (Figure 41), and this change will be notified to the screens like NewsPage but won't cause any rebuilds for NoteViewPage itself.



```
if (widget?.viewType == NoteMode.Add) {
  // create
  notesViewModel.createNote(Note(_title, _text));
} else {
  // update
  notesViewModel.updateNote(Note.withId(widget.note.id, _title, _text));
}
```

Figure 41. Example usage of the injected ViewModel

In some cases, it can bring a significant performance improvement. At the same time, it is extremely simple to achieve – only one parameter has to be set to false.

Benefits and drawbacks

Below you can see a list of the most significant benefits of the Provider (+ChangeNotifier) state management approach:

1. It is easy to understand for most people, especially compared to InheritedWidget.
2. It is extensively used by the community, a lot of examples and tutorials are available online, recommended by the Flutter team.
3. It has the smallest amount of boilerplate code among all state management approaches.

4. It is easy to reach the high level of separation of views and business logic (the business logic and state management related tasks can fully be lifted to a separate class).
5. It is easy to implement MVVM or other architectures.
6. It is scalable.

Also, it has some drawbacks:

1. It is still easy to accidentally cause unneeded UI updates when there is a state change. In most cases, it can be mitigated by setting the *listen* property to false.
2. Some developers still find it difficult to use.
3. It is more difficult to implement for larger apps.

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 6.

Table 6. Provider assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
easy	a little	no	no	good	average

This approach is easy because there is quite a lot of documentation, and because it follows the principles similar to MVC, MVP, MVVM and other architectural patterns, which are quite extensively used in other fields of software development. The amount of boilerplate code is small. Scalability is good because of how easy it is to enforce the principles of separation of concerns with Provider. Testability is average because this approach doesn't force the developers to write testable code, which means that they can still write code for which there will be problems with writing unit tests.

When to use

Provider can be used in any case, if there is no explicit need for default "time travel" support, and no specific preference for Streams/Observables/reactive patterns. It might not be the best solution for large projects (because too many

ChangeNotifiers and listeners attached to them can potentially create some performance issues), but for small to average it is usually a good fit.

3.4.4 GetX

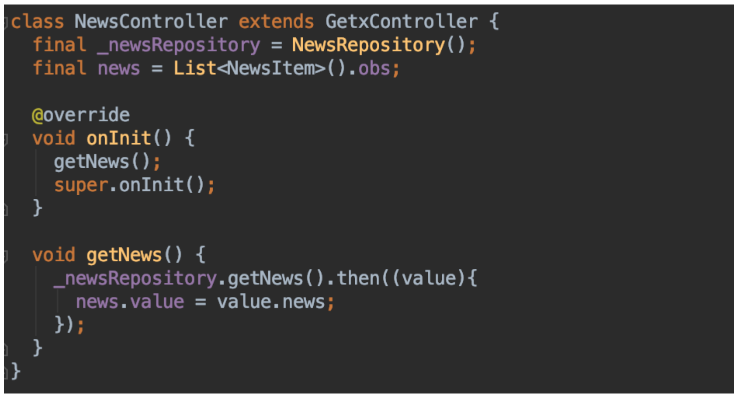
GetX is a simplified reactive state management solution (Flutter Website 2020i). It is an extra-light and powerful solution, which consists of three main elements: high performance state management, intelligent dependency injection, and quick and practical route management (Pub.dev 2020f).

GetX is the second popular state management solution after Provider. The fact that it has a big active community and positions itself as a framework that solves the main difficult issues of Flutter development (not only state management, but also the other two mentioned previously) definitely contributes to its enormous popularity.

Practical implementation

The business logic classes in GetX are called controllers. So, instead of creating three ViewModels, I have created three Controllers: NotesController, NewsController, SettingsController.

Similarly to the Provider section, I only include here the code for NewsController (Figure 42), since other controllers are built the same way. The repository is instantiated in the controller itself just like in Section 3.4.3. Then when a state variable is initialized, it can be made observable by adding ".obs" right after it. For example, if we want to create a Boolean observable which is false by default, we can simply type "*final darkMode = false.obs;*". There are other ways to initialize an observable, but this one is the simplest and the most explicit one. And that's it, we can now freely update the value of this observable whenever it is needed.

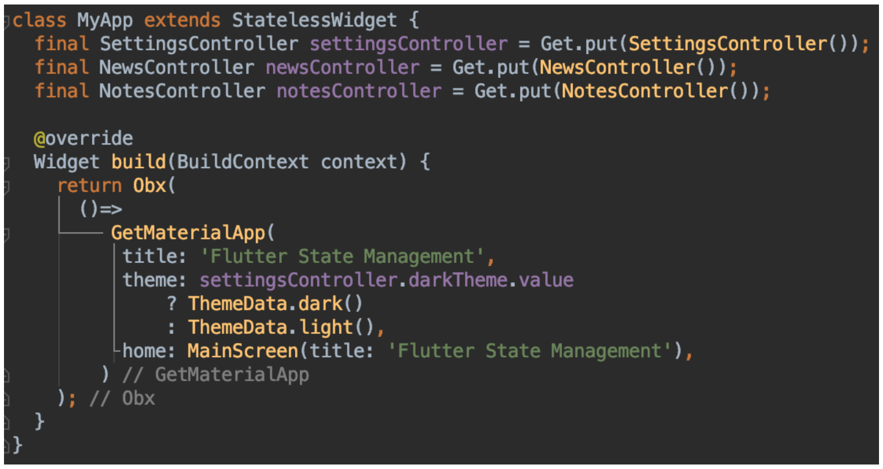


```
class NewsController extends GetxController {
  final _newsRepository = NewsRepository();
  final news = List<NewsItem>().obs;

  @override
  void onInit() {
    getNews();
    super.onInit();
  }

  void getNews() {
    _newsRepository.getNews().then((value){
      news.value = value.news;
    });
  }
}
```

Figure 42. NewsController for GetX

There is also an onInit() method which can be overridden and some initialization code that will be executed only when the controller is created can be put there. I have put getNews() call to this method so that the news loading process is started even before the interface is drawn. We were able to do the same thing with Provider, but there it was necessary to put this code in the view file (main.dart). The approach of GetX is much better as it allows to put all the business logic related code to the controller files and only bind the views to the corresponding values, which is good for enforcing the separation of concerns idea.

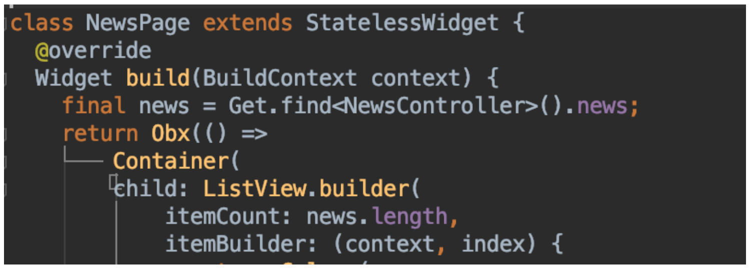
After creating the controllers, we can inject them using the dependency injection mechanism of GetX. It is indeed very simple – no need to use any widgets, we can simply wrap the instantiation of the controllers with Get.put() wherever we want and it will be automatically injected. The initialized instance can be immediately used like in Figure 43. In other classes, Get.find() can be used to inject the dependency.



```
class MyApp extends StatelessWidget {
  final SettingsController settingsController = Get.put(SettingsController());
  final NewsController newsController = Get.put(NewsController());
  final NotesController notesController = Get.put(NotesController());

  @override
  Widget build(BuildContext context) {
    return Obx(
      ()=>
        GetMaterialApp(
          title: 'Flutter State Management',
          theme: settingsController.darkTheme.value
              ? ThemeData.dark()
              : ThemeData.light(),
          home: MainScreen(title: 'Flutter State Management'),
        ) // GetMaterialApp
    ); // Obx
  }
}
```

Figure 43. MyApp code for GetX approach

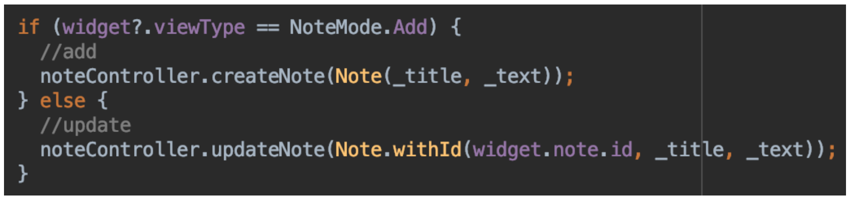
GetX has two options for consuming the reactive observables: Obx and Getx. Obx is a lightweight implementation which only takes an anonymous function which returns a widget and thus can be used simply as *Obx(() => MyWidget())*. The controller has to be manually injected with Get.find() in this case. Getx option allows to specify the controller class so it will be injected automatically and provided to the builder. Also, this option has some extra features for custom initialization, disposal and so on. All this results in the fact that Getx consumes more RAM than Obx, so Obx is a better choice in most cases. In Figure 44, we are injecting the list of news using Get.find() and then consuming it by wrapping the widget that uses it in Obx.



```
class NewsPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final news = Get.find<NewsController>().news;
    return Obx(() =>
      Container(
        child: ListView.builder(
            itemCount: news.length,
            itemBuilder: (context, index) {
```

Figure 44. Obtaining state from NewsController in NewsPage

In simple words, Obx tells the framework that its children are subscribed to the updates of the observables used in them. We don’t need to use Obx when we only need to call a method from a controller. In NoteViewPage, the controller is injected using *final noteController = Get.find<NotesController>()*. And then it is used as shown in Figure 45.



```
if (widget?.viewType == NoteMode.Add) {
  //add
  noteController.createNote(Note(_title, _text));
} else {
  //update
  noteController.updateNote(Note.withId(widget.note.id, _title, _text));
}
```

Figure 45. Example usage of the injected Controller

Another bonus of the GetX framework not related to state management but still making the life of the developers easier is the simplified navigation. It is possible to navigate to a new screen simply by using *Get.to(Screen())*. To go back, we only need to type *Get.back()*.

We can see that this approach is indeed reactive because we have several GetxControllers with observables inside them and Obx subscribers. Whenever the observable values change in the controllers, the subscribers are notified and react to the changes immediately. There is no need to manually issue any commands like notifyListeners() in ChangeNotifier from Section 3.4.3.

GetX also has a non-reactive state management mechanism which works exactly like ChangeNotifier from Section 3.4.3 but is not built upon InheritedWidget or ChangeNotifier. It is not covered here due to its extreme similarity to the already described approach.

Benefits and drawbacks

Below you can see a list of the most significant benefits of the GetX state management approach:

1. It provides the power of reactive programming.
2. Reactive patterns can be implemented with the minimum amounts of boilerplate code.
3. It is relatively easy to learn.
4. It supports both reactive and non-reactive state management, and also includes a convenient dependency injector and a wrapper around Navigator.
5. It offers true separation of business logic and user interface code.

It also has some drawbacks:

1. As a reactive approach, it consumes more system resources. Incorrect usage of GetX may potentially result in performance issues on some devices.
2. The package is reported to sometimes introduce breaking changes.
3. The community around this package, despite being quite big compared to others, is quite defensive and not always welcomes criticism (including the main developer).
4. The package itself has some bad design solutions like global mutable state which may result in difficulties in big projects.

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 7.

Table 7. GetX assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
easy	a little	no	no	good	average

There are many comments online stating that this approach is easy to understand. It aims to remove the boilerplate code by providing the Obx widget for which the developers don't even have to specify the controller type. They still have to inject it manually, though. Scalability is quite good thanks to Obx widget and to the fact that business logic and UI code can be separated using the controllers. Testability is average because this approach doesn't force the developers to write testable code just like the previous one.

When to use

GetX can be used by the developers who are new to Flutter and haven't found the most comfortable state management approach yet but want to have the benefits of reactive programming while keeping the things simple. I would not recommend using it in big projects because of the drawbacks described in the previous section.

3.4.5 BLoC

BLoC is a family of reactive steam/observable based patterns. Its most popular and widely used implementation is the bloc library by Felix Angelov, and its Flutter integration called flutter_bloc (Pub.dev 2020g). BLoC is often compared to MVVM pattern with the difference that a ViewModel from MVVM is replaced with BLoC. It was a recommended approach for state management before the Flutter team had decided to promote the simpler Provider.

BLoC stands for Business Logic Component. The idea behind it is that a Bloc receives events which trigger state changes which are then emitted from the Bloc. So, Bloc has decoupled inputs and outputs. You provide events to the input, and you receive state from the outputs. The outputs can be observed by the UI widgets which react to the state changes. Figure 46 illustrates how BLoC pattern works.

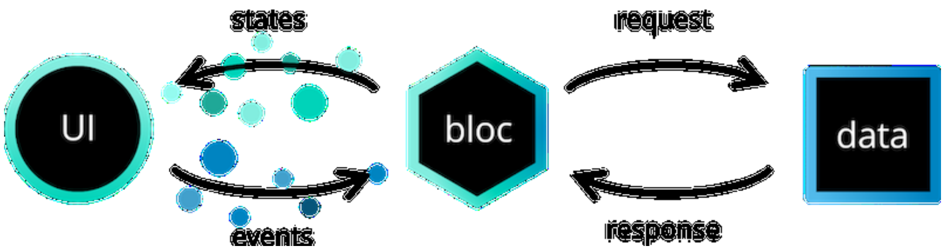


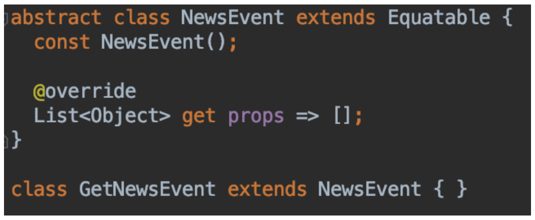
Figure 46. How BLoC works

Bloc itself acts as a middleman between the UI and the data access layer, which is quite often represented by a repository. This means that events trigger some business logic inside the Bloc, which can for instance ask the repository to make an API request. Once the response is received, the Bloc can process the data and emit the corresponding state.

Practical implementation

The first step is to define the classes for each Bloc. In our case, we will have three Blocs, and each Bloc consists of three parts: events class, states class and the business logic component with event to state mappings. The role of Bloc is similar to the ViewModel or Controller which were used in the previously studied state management approaches. However, the way it is implemented is different.

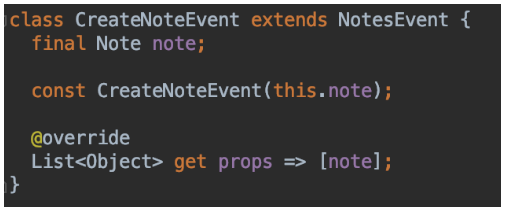
The events are implemented simply by defining an abstract class which represents a generic event, and then creating specific events by extending this class. For example, for the News feature we have only one event – GetNewsEvent (Figure 47). As this event doesn't need any input parameters, we don't even need to type anything in the class body.



```
abstract class NewsEvent extends Equatable {
  const NewsEvent();

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

class GetNewsEvent extends NewsEvent { }
```

Figure 47. Events definition for the News Bloc

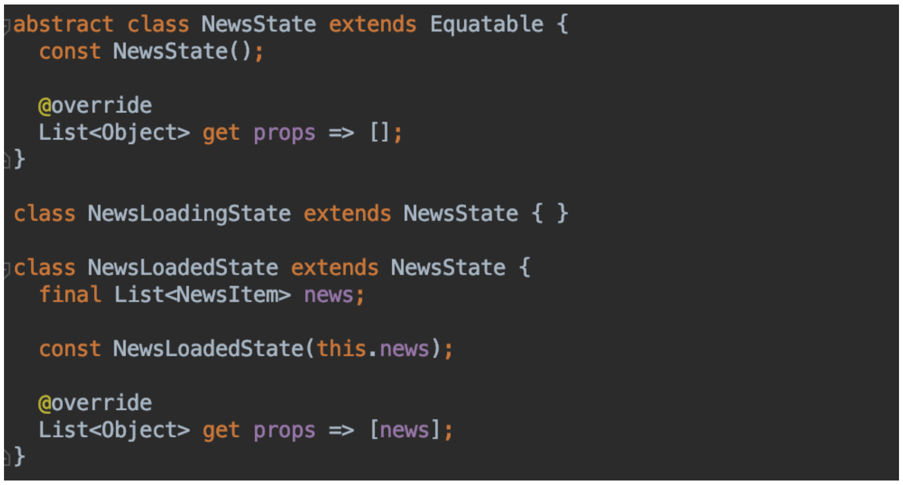
CreateNoteEvent from Notes Bloc (Figure 48), on the other hand, takes a Note object as input, and therefore its body is defined.



```
class CreateNoteEvent extends NotesEvent {
  final Note note;

  const CreateNoteEvent(this.note);

  @override
  List<Object> get props => [note];
}
```

Figure 48. CreateNoteEvent from Note Bloc

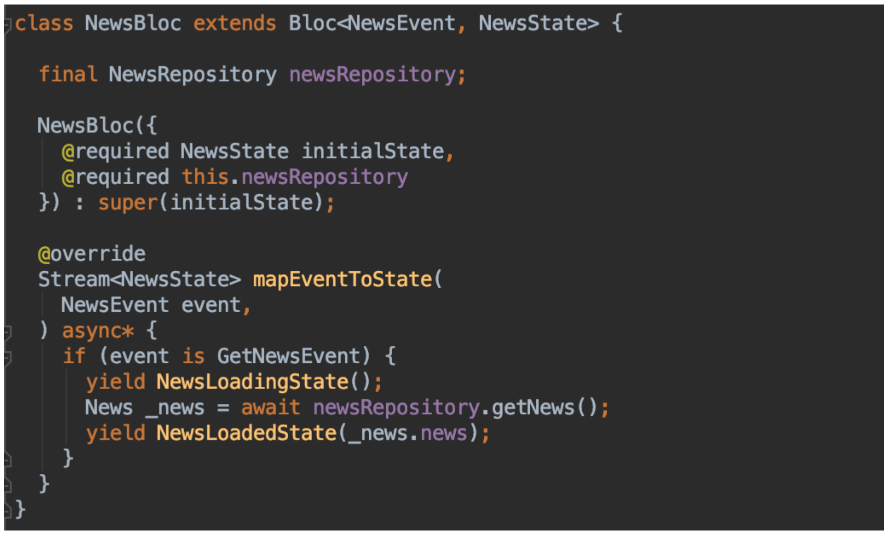
The next step after we have defined the events which serve as inputs for the Bloc is to define the states which serve as the reactive outputs. News Bloc has two possible states – NewsLoadingState and NewsLoadedState (Figure 49). The first one has no variables inside it as it is simply used to indicate that the news are still loading. The second one contains the list of news.



```
abstract class NewsState extends Equatable {
  const NewsState();

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

class NewsLoadingState extends NewsState { }

class NewsLoadedState extends NewsState {
  final List<NewsItem> news;

  const NewsLoadedState(this.news);

  @override
  List<Object> get props => [news];
}
```

Figure 49. States definition for the News Bloc

Finally, the NewsBloc class which contains the business logic is shown in Figure 50. It takes NewsRepository and initial state of the type NewsState as constructor parameters. Then there is an important mapEventToState method which actually defines what the Bloc has to do after it receives a new event, and which state should be emitted after some logic is executed. In our News example, on GetNewsEvent we first emit NewsLoadingState, then wait for the news to be asynchronously obtained from the repository, and after it's done, we emit NewsLoadedState with the loaded news list inside it.



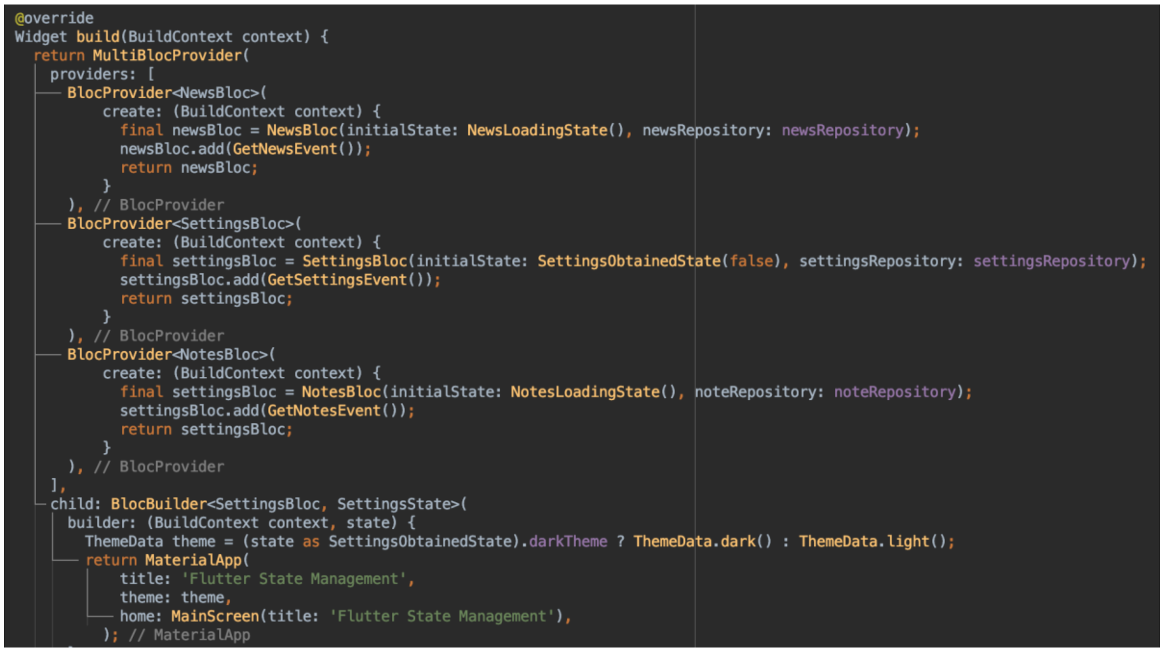
```
class NewsBloc extends Bloc<NewsEvent, NewsState> {

  final NewsRepository newsRepository;

  NewsBloc({
    @required NewsState initialState,
    @required this.newsRepository
  }) : super(initialState);

  @override
  Stream<NewsState> mapEventToState(
    NewsEvent event,
  ) async* {
    if (event is GetNewsEvent) {
      yield NewsLoadingState();
      News _news = await newsRepository.getNews();
      yield NewsLoadedState(_news.news);
    }
  }
}
```

Figure 50. News Bloc

Some keywords in Figure 50 may look unfamiliar to the reader, so let's clarify what they mean. **async*** means that this function is an asynchronous generator function. These functions can return a sequence of values instead of only one value in the regular functions. **yield** keyword is like a **return** for generator functions, but it does not stop the execution of the code when called, which is perfect for our goal of emitting states.

In a similar manner to News Bloc, I have implemented Settings and Notes Blocs. Settings Bloc features two events: GetSettingsEvent() and SetSettingsEvent(bool darkMode). Then it has only one state – SettingsObtainedState(bool darkMode). Notes Bloc has five events: GetNotesEvent(), CreateNoteEvent(Note note), UpdateNoteEvent(Note note), and DeleteNoteEvent(int noteId). It has two states: NotesLoadingState() and NotesLoadedState(List<Note> notes).

When Blocs are ready, it is time to start writing the code in the view files. Firstly, the repositories have to be instantiated on top of the widget tree in MyApp widget. Then we should use the MultiBlocProvider widget for dependency injection. If we look at Figure 51 and Figure 39, we can see that the dependency injection here is done exactly the same way as in the Provider approach.

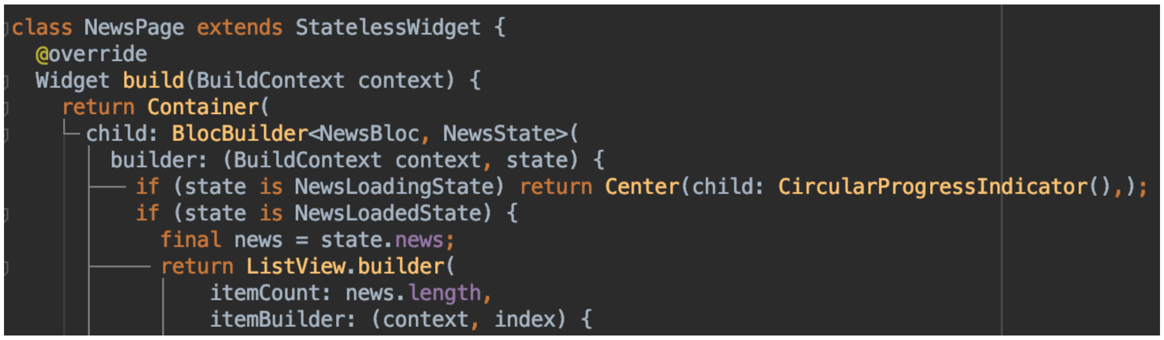


```
@override
Widget build(BuildContext context) {
  return MultiBlocProvider(
    providers: [
      BlocProvider<NewsBloc>(
          create: (BuildContext context) {
            final newsBloc = NewsBloc(initialState: NewsLoadingState(), newsRepository: newsRepository);
            newsBloc.add(GetNewsEvent());
            return newsBloc;
          }
      ), // BlocProvider
      BlocProvider<SettingsBloc>(
          create: (BuildContext context) {
            final settingsBloc = SettingsBloc(initialState: SettingsObtainedState(false), settingsRepository: settingsRepository);
            settingsBloc.add(GetSettingsEvent());
            return settingsBloc;
          }
      ), // BlocProvider
      BlocProvider<NotesBloc>(
          create: (BuildContext context) {
            final settingsBloc = NotesBloc(initialState: NotesLoadingState(), noteRepository: noteRepository);
            settingsBloc.add(GetNotesEvent());
            return settingsBloc;
          }
      ), // BlocProvider
    ],
    child: BlocBuilder<SettingsBloc, SettingsState>(
      builder: (BuildContext context, state) {
        ThemeData theme = (state as SettingsObtainedState).darkTheme ? ThemeData.dark() : ThemeData.light();
        return MaterialApp(
          title: 'Flutter State Management',
          theme: theme,
          home: MainScreen(title: 'Flutter State Management'),
        ); // MaterialApp
```

Figure 51. MyApp code for BLoC approach

We can also provide some initialization code for each Bloc. For example, I have added a “get” event for each mode so that whenever Bloc is first created, the data will be loaded in it automatically. Blocs are instantiated lazily, which means that the creation code will be executed only when a particular Bloc is used somewhere else in the app. This behavior can be easily changed if needed.

Then we can use BlocBuilder widget which handles building the widget in response to new states. For example, in Figure 51 we use BlocBuilder for Settings Bloc in order to obtain the darkTheme Boolean value from state. Since there is only one possible state type for this Bloc, it is safe to explicitly cast state to SettingsObtainedState. Another example of BlocBuilder usage can be seen in Figure 52.



```
class NewsPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: BlocBuilder<NewsBloc, NewsState>(
        builder: (BuildContext context, state) {
          if (state is NewsLoadingState) return Center(child: CircularProgressIndicator(),);
          if (state is NewsLoadedState) {
            final news = state.news;
            return ListView.builder(
                itemCount: news.length,
                itemBuilder: (context, index) {
```

Figure 52. NewsPage for BLoC approach

We use it to render the news page, and it selects what to display based on the current state. If state is NewsLoadingState, a circular progress indicator is shown in the middle of the screen. If state is NewsLoadedState, the news list is obtained from it and then used exactly the same way as in all other state management approaches.

If we don’t need to react to state changes and simply need to add the events to Bloc in order to trigger the updates elsewhere, we can obtain the non-reactive reference to Bloc instance as follows:
NotesBloc notesBloc = BlocProvider.of(context);

Then it is very easy to add events:
notesBloc.add(CreateNoteEvent(Note(_title, _text)));
notesBloc.add(DeleteNoteEvent(widget.note.id));

Benefits and drawbacks

Below there is a list of the most significant benefits of the BLoC state management approach:

1. It provides the power of reactive programming
2. It has better separation of concerns because not only business logic is fully lifted to a separate class, but also inputs and outputs are decoupled
3. It enforces the use of MVVM-like architecture
4. It provides built-in dependency injection mechanism for Blocs which works exactly like the Provider package and uses lazy loading by default
5. It is very good in terms of testability
6. It allows to have easy to reuse the business logic code written once in the same app and even in different apps
7. It offers very good scalability

It also has some drawbacks:

1. As a reactive approach, it consumes more system resources
2. It has a lot of boilerplate code which is a price of the second benefit
3. It has a steep learning curve

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 8.

Table 8. BLoC assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
difficult	a lot	no	no	the best	good

This approach is relatively difficult according to the developers' opinions which I read online. The reason is because it works differently from other approaches and has its own complicated structure. This results in a lot of boilerplate code. However, this is the price for having the best scalability – because a concise architecture is automatically enforced by this approach, which also results in good testability.

When to use

BLoC can be considered an advanced approach, because it is impossible to start using it right away like Provider or GetX, and the concepts which lie behind it are quite complicated. However, for the price of these complexity and a lot of boilerplate code, we can write the code that is easily testable, reusable, scalable. This approach is also suitable for clean architecture. Therefore, these facts lead us to the idea that a perfect use case for this approach is a complex medium to large application.

3.4.6 MobX

MobX is a popular library based on observables and reactions (Flutter Website 2020i). It utilizes annotations and code generation in order to reduce the amount of boilerplate code. Even though MobX is placed into the group of reactive state management approaches, in reality it uses transparent functional reactive programming.

Figure 53 shows three main concepts of MobX: Observables, Actions and Reactions. Observables represent the reactive state of the app. Actions are used to mutate the observables. Reactions observe the reactive state and get notified by observables when it changes, so that they are able to produce side effects, i.e. rebuilding a widget with new state.

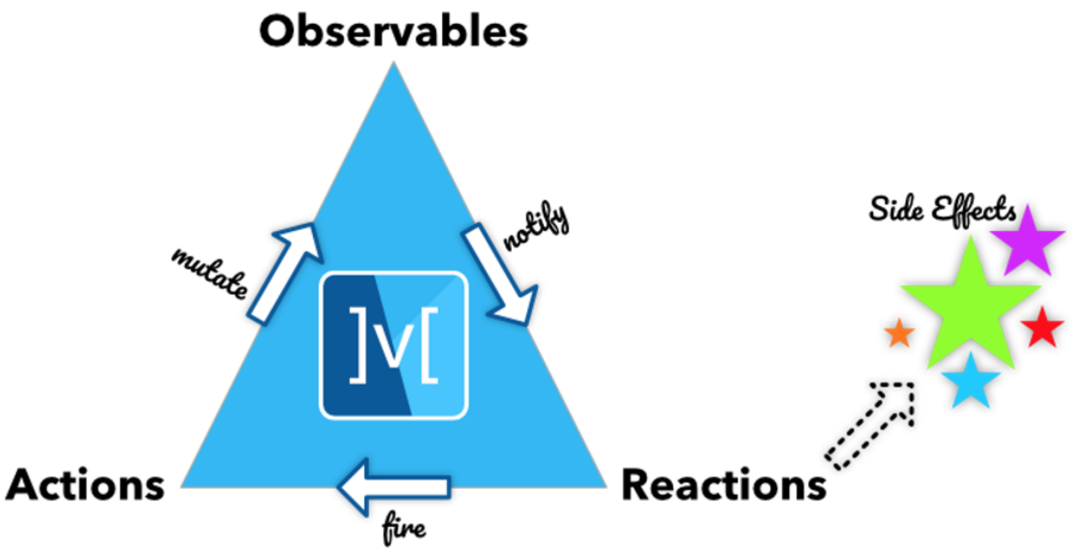
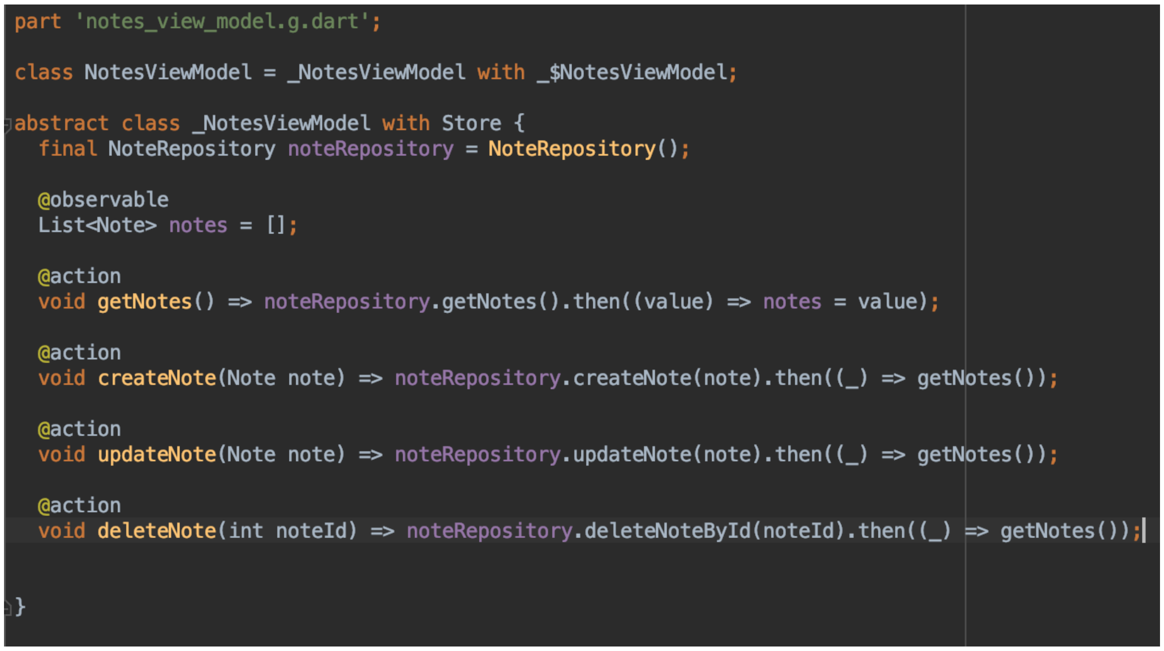


Figure 53. Core concepts of MobX

Observer widget is one of the most often used reactions. It does exactly what was mentioned before – rebuilds a widget with new state,

Practical implementation

The first step is, similarly to several previous approaches, is to create the ViewModel classes which contain observable state variables and actions which mutate these variables. This time only NotesViewModel is shown because it is the most complex one. It can be seen in Figure 54.



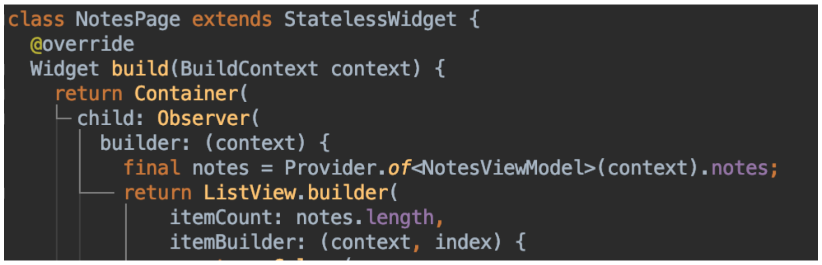
```
part 'notes_view_model.g.dart';

class NotesViewModel = _NotesViewModel with _$NotesViewModel;

abstract class _NotesViewModel with Store {
  final NoteRepository noteRepository = NoteRepository();

  @observable
  List<Note> notes = [];

  @action
  void getNotes() => noteRepository.getNotes().then((value) => notes = value);

  @action
  void createNote(Note note) => noteRepository.createNote(note).then((_) => getNotes());

  @action
  void updateNote(Note note) => noteRepository.updateNote(note).then((_) => getNotes());

  @action
  void deleteNote(int noteId) => noteRepository.deleteNoteById(noteId).then((_) => getNotes());


}
```

Figure 54. NotesViewModel for MobX

After creating the ViewModels, the code generation process should be started by running the following command: *flutter packages pub run build_runner build*. It will automatically generate the actual Dart code behind the @observable and @action annotations.

A huge drawback of MobX is that it doesn't have any built-in dependency injection solution, so you have to use Provider or something else in order to provide the instances of the ViewModels to the widgets which use them. It means that in our case we have to use the same code as in Figure 39, with the only difference that ChangeNotifierProvider should be changed to Provider.

The lists of news/notes/settings are obtained almost the same way as in the Provider approach. First, we need to wrap the lowest possible widget that is affected by the change in Observer(). Then we can obtain the required state value from the ViewModel using Provider. An example with notes can be seen in Figure 55. Whenever the state value changes, Observer will rebuild its contents based on the new state.



```
class NotesPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Observer(
        builder: (context) {
          final notes = Provider.of<NotesViewModel>(context).notes;
          return ListView.builder(
            itemCount: notes.length,
            itemBuilder: (context, index) {
```

Figure 55. NotesPage for MobX approach

Finally, calling the actions is as simple as obtaining an instance of ViewModel using Provider and calling the required action method.

Benefits and drawbacks

Below we can see a list of the most significant benefits of the MobX state management approach:

1. It provides the power of reactive programming
2. It has very small amounts of boilerplate code
3. Component update mechanism is abstracted – you just need to create a class with observables and actions and then just use the observables in Observer() children and actions anywhere in the app
4. It is easy to learn the basic concepts
5. Prototypes can be created really fast
6. It offers freedom in the selection of the architecture for the app

It also has some drawbacks:

1. It is resource consuming
2. It is based on the code generation
3. There is a lack of built-in dependency injection framework (need to use Provider)
4. Testability depends on the implementation (you have to write the code in a specific manner in order for it to be easily testable)

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 9.

Table 9. MobX assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
easy	small	yes	no	average	average

This approach is easy because it has a simple structure that can be understood even without special knowledge. The problem of big amounts of boilerplate code is solved by code generation. There is still a lot of boilerplate code, however, the developers don't need to write it manually, so it doesn't count. Scalability is average, because the bigger the app gets, the higher chances are that a developer will face some issues because of the fact that he or she is not in control of everything. Resource consumption also plays a negative role for scalability. Testability is average because this approach doesn't force the developers to write testable code.

When to use

The best use case for MobX is creating prototypes. It is very easy to learn and to implement, it has almost no boilerplate code and can fit into different architectures. You don't need to actually think what is happening behind the scenes, since the framework is very abstracted, and you can just trust it on doing its job. However, the freedom offered by MobX results in the fact that you need to be really careful when the app grows because the scalability and testability depends on your actual implementation. The framework doesn't force you to write testable and scalable code like, for instance, BLoC.

3.4.7 Redux

Flutter Website (2020i) describes this approach as a state container approach familiar to many web developers. Indeed, Redux is de-facto the default state management approach in React applications. Redux is a JavaScript library for managing application state. It has a simple, limited API designed to be a predictable container. It operates in a similar manner to a reducing function, which is a concept originating from the functional programming paradigm (Wikipedia 2020d).

Every state update in Redux goes through the same cycle which is illustrated in Figure 56. State is what defines the UI, as stated in Section 2.2.5. Redux has three main building blocks: Actions, Reducers, and Store. The Store simply contains the application state. UI triggers Actions, which are sent to Reducer functions, which, in turn, updates the Store. Then the UI is rebuilt based on the new state.

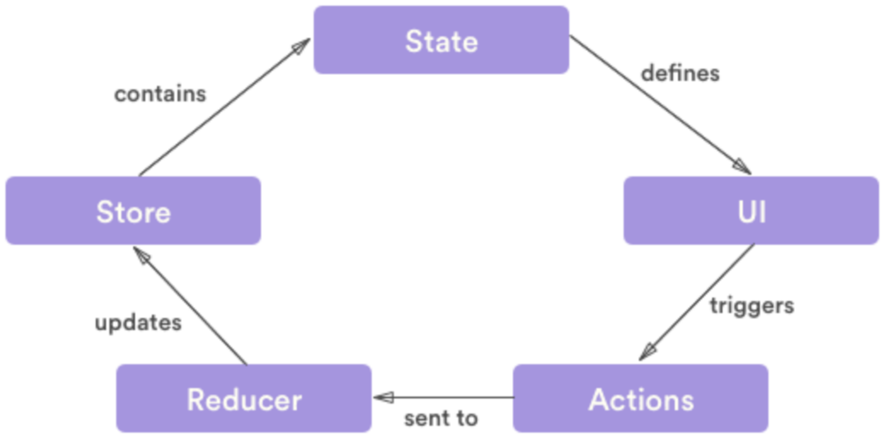


Figure 56. Redux cycle

Let’s study each component in a little bit more detail. **Store** contains the entire state logic of the application. It serves as a single source of truth and is globally accessible. Store allows to set the initial state, access state, and update state by dispatching an action. It is important to understand that state in Redux is immutable, so each time state updates it is not directly mutated, but rather a copy of the existing state with the updated values is created.

Actions are payloads of information which send data from the app to the store. They serve as the only input for the store. In Dart, they can be either simple enums or classes if some payload information has to be passed to the store. In simple words, actions describe the fact that something happened. However, it is not their responsibility to define how the app state changes in response.

Reducers are pure functions that take the previous state and an action and return a new state. It is not allowed to mutate the reducer arguments, perform side effects like API calls or local persistence operations, calling non-pure functions. A pure function is a function where the return value is only determined by its input values (e.g., like functions in math).

The logical question here is how to access the APIs and local storage if reducers have to be kept pure. Another part of Redux is responsible for this part – **middleware**. Middleware are special functions that run before the dispatched actions reach the reducer. They are usually used to listen for different actions and perform asynchronous calls, such as remote API requests. Once they get a response from the server, they can dispatch other actions to the reducer. Figure 57 shows the full picture.

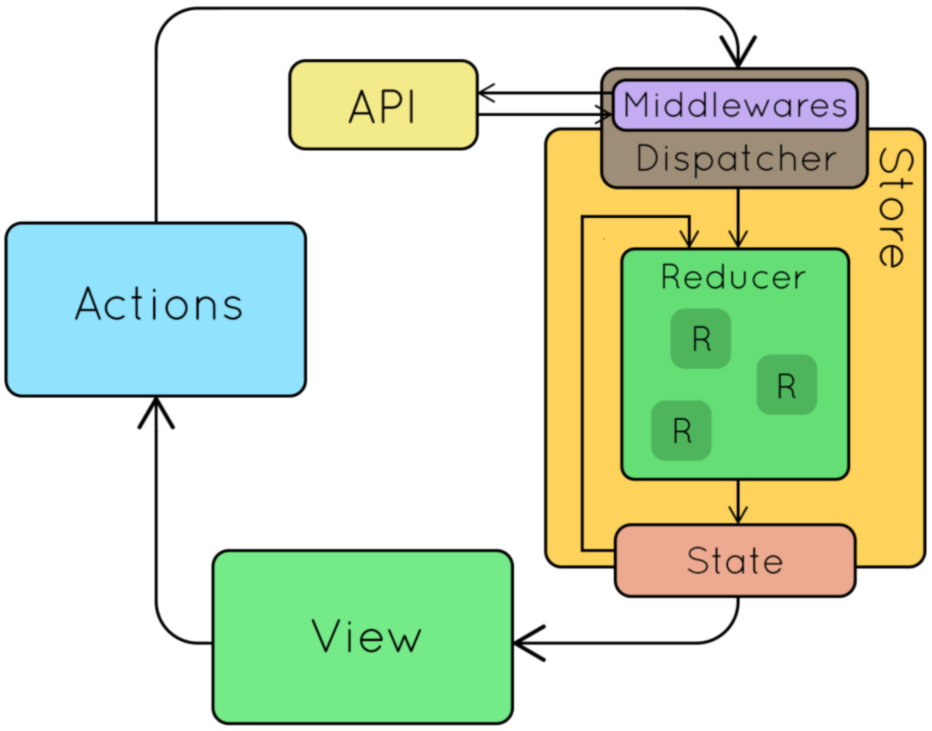


Figure 57. How Redux works with middleware

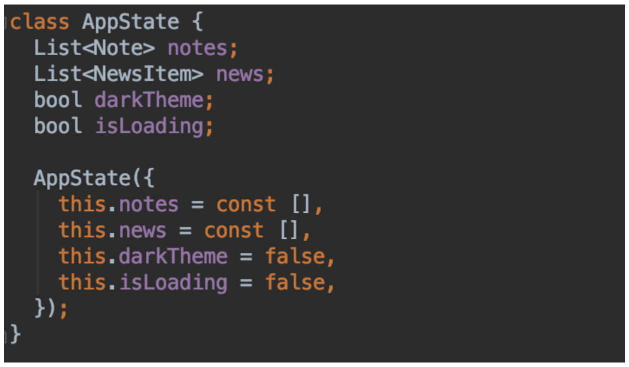
The UI sends an Action to the Store dispatcher, which first passes it through the middleware and then the middleware can either pass the same action to the reducer in case if it is not configured to do anything for this kind of action, or pass a different action to the reducer (which can, for instance, trigger the reducer to set state to loading, and the UI will display a spinner), then start an asynchronous operation, wait for its result, and based on the result, pass some other action to the reducer. This action can contain the results of the asynchronous operation.

Practical implementation

The practical implementation of this approach differs from the implementation of all of the previous approaches significantly. The reason for that is because of the functional programming nature of Redux. Reducers and middleware, the main state management mechanisms of Redux, are functions, which do not belong to any class.

This time I did not divide the code in files based on to which screen it belongs. For the simplicity, all actions are store in one file, no matter to which screen (notes/news/settings) they belong. The same is true for reducers, middleware and state. It is not a bad idea to create a separate file for each screen, but I have chosen to keep the things as simple as possible because otherwise I would have had three times more files. For our small demo app, the fact that I have done that does not affect the code readability or maintainability.

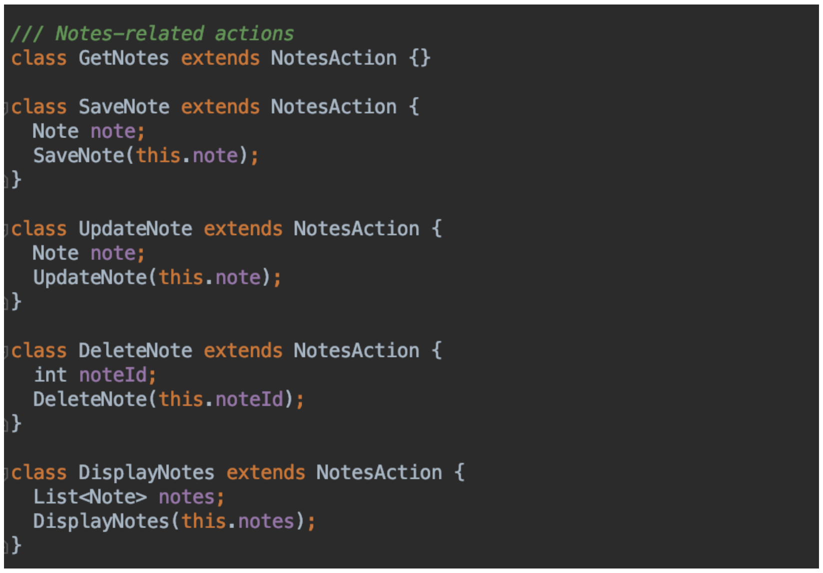
The first step is to describe the shape of the application state. Our app's state consists of the list of notes, news and darkTheme Boolean value. We can also add an isLoading variable to keep track of when something is being loaded. Figure 58 shows the code of the AppState class which is simply a data model class with the previously mentioned variables.



```
class AppState {
  List<Note> notes;
  List<NewsItem> news;
  bool darkTheme;
  bool isLoading;

  AppState({
    this.notes = const [],
    this.news = const [],
    this.darkTheme = false,
    this.isLoading = false,
  });
}
```

Figure 58. AppState class for Redux store

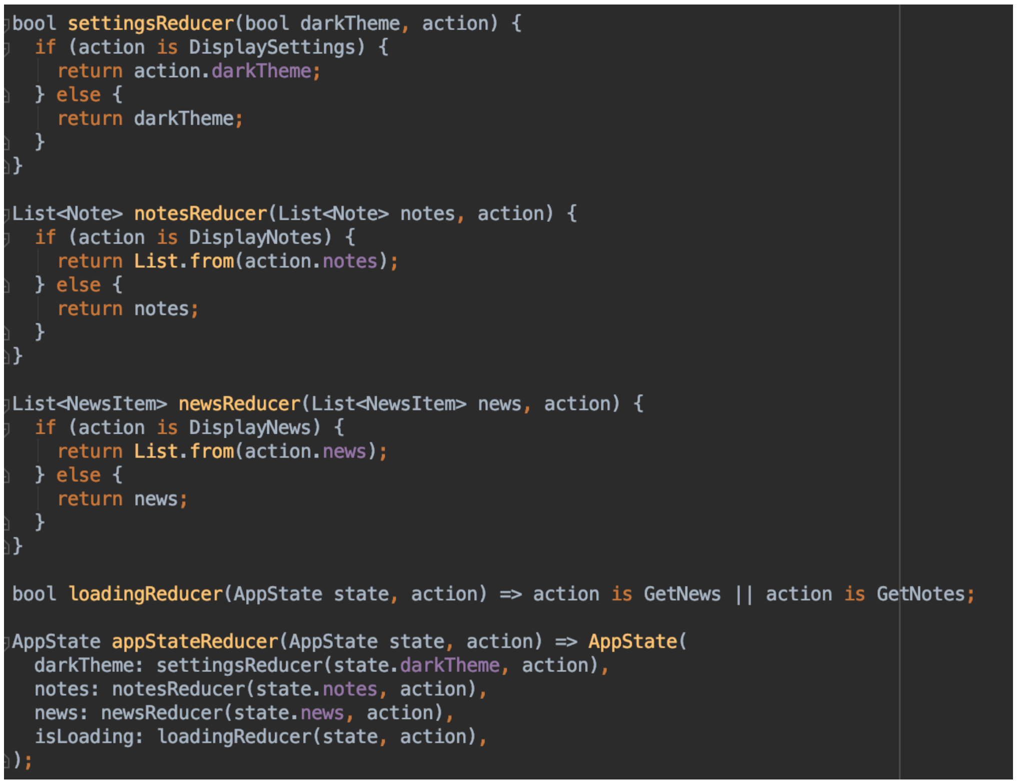
Following this, I have created actions for each possible trigger in the app that should be able to cause some state change. Figure 59 shows the actions related to notes, as it has the biggest number of possible actions in our app. GetNotes should trigger the loading of the notes list from sqflite. SaveNote, UpdateNote and DeleteNote actions are self-explanatory. Finally, there is a DisplayNote action. We need it because reducers cannot have asynchronous operations, only middleware can. One action is not enough to handle such operation. GetNotes triggers the note loading process when it goes through the middleware, and then it sets the loading state to true in the reducer. When the loading process in middleware is completed, a DisplayNotes action is issued by the middleware with the list of loaded notes. Then the reducer can update state with this list.



```
/// Notes-related actions
class GetNotes extends NotesAction {}

class SaveNote extends NotesAction {
  Note note;
  SaveNote(this.note);
}

class UpdateNote extends NotesAction {
  Note note;
  UpdateNote(this.note);
}

class DeleteNote extends NotesAction {
  int noteId;
  DeleteNote(this.noteId);
}

class DisplayNotes extends NotesAction {
  List<Note> notes;
  DisplayNotes(this.notes);
}
```

Figure 59. Notes-related actions

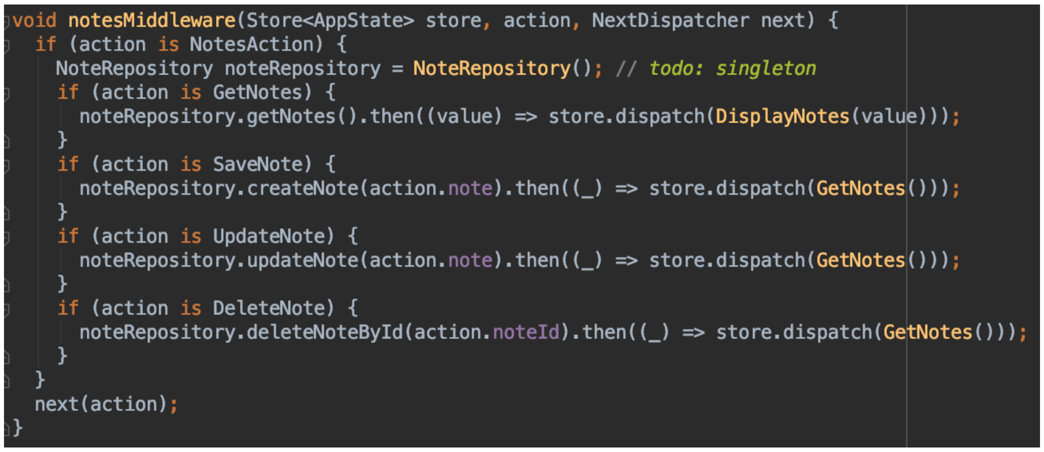
The next step is to create the reducer for the app. I have decided to not make this overcomplicated too and did not use the type safe reducer combination. I have simply created four reducers, and then combined them into one appStateReducer. Refer to the Figure 60. Each reducer updates the state value if the action is Display (DisplaySettings for settingsReducer, DisplayNotes for notesReducer, DisplayNews for newsReducer). Otherwise, the old value is returned. Loading reducer sets isLoading to true if the action that passes through it is GetNews or GetNotes. Every time the appStateReducer is called, each smaller reducer is called with the corresponding value and action, and then this reducer is responsible for determining whether its part of state should be updated or not.



```
bool settingsReducer(bool darkTheme, action) {
  if (action is DisplaySettings) {
    return action.darkTheme;
  } else {
    return darkTheme;
  }
}

List<Note> notesReducer(List<Note> notes, action) {
  if (action is DisplayNotes) {
    return List.from(action.notes);
  } else {
    return notes;
  }
}

List<NewsItem> newsReducer(List<NewsItem> news, action) {
  if (action is DisplayNews) {
    return List.from(action.news);
  } else {
    return news;
  }
}

bool loadingReducer(AppState state, action) => action is GetNews || action is GetNotes;

AppState appStateReducer(AppState state, action) => AppState(
  darkTheme: settingsReducer(state.darkTheme, action),
  notes: notesReducer(state.notes, action),
  news: newsReducer(state.news, action),
  isLoading: loadingReducer(state, action),
);
```

Figure 60. Reducers

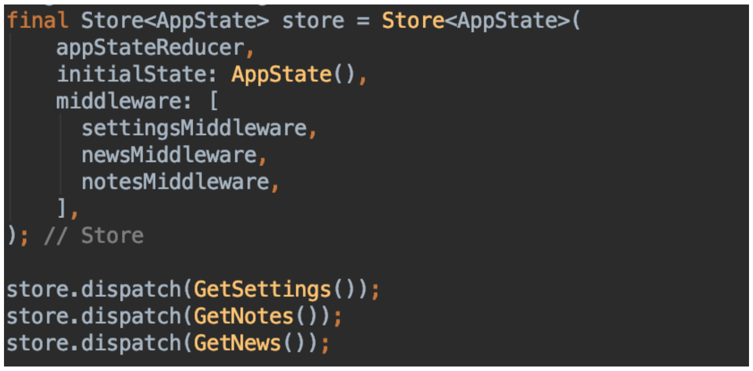
After writing the reducers, it is time to also create middleware. Middleware is provided to store as an array of functions, so we don’t need to think about how we should connect several functions like we did with reducers. We can take a look at the notes’ middleware, which can be seen in Figure 61. Firstly, this middleware checks if the action that goes through it is a NotesAction. If it is, then the middleware instantiates a repository. Here we should notice that it should be a singleton for the performance consideration, because it is now more difficult to inject dependencies. Then, in response to each action, the middleware calls a corresponding repository method, and also tells which actions should be dispatched after each promise is resolved. For example, after the notes are loaded, the DisplayNotes action is dispatched, and after all other actions, the GetNotes action is dispatched in order to refresh the views when a note is added/updated/deleted. After calling the asynchronous operations and defining what has to be done after they are completed, the middleware calls next(action) which calls the following middleware if there is one or the reducer. The middleware itself is synchronous, it doesn't wait for the asynchronous repository operations to finish.



```
void notesMiddleware(Store<AppState> store, action, NextDispatcher next) {
  if (action is NotesAction) {
    NoteRepository noteRepository = NoteRepository(); // todo: singleton
    if (action is GetNotes) {
      noteRepository.getNotes().then((value) => store.dispatch(DisplayNotes(value)));
    }
    if (action is SaveNote) {
      noteRepository.createNote(action.note).then((_) => store.dispatch(GetNotes()));
    }
    if (action is UpdateNote) {
      noteRepository.updateNote(action.note).then((_) => store.dispatch(GetNotes()));
    }
    if (action is DeleteNote) {
      noteRepository.deleteNoteById(action.noteId).then((_) => store.dispatch(GetNotes()));
    }
  }
  next(action);
}
```

Figure 61. notesMiddleware

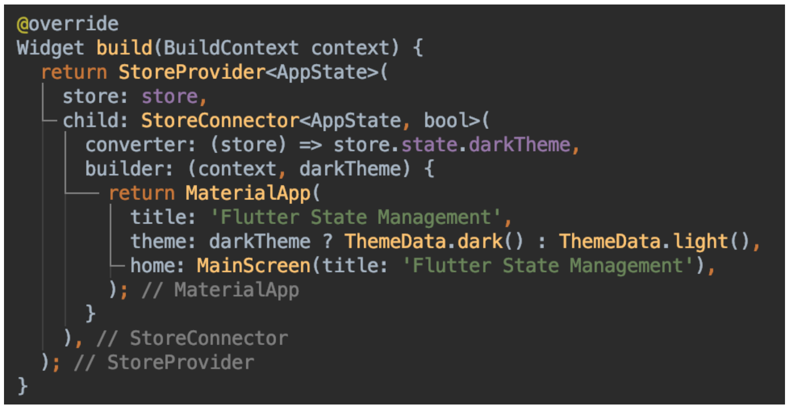
At this stage, all the Redux preparations are finished, we can now construct our store and use it for managing state. Figure 62 shows how the store is initialized. We have to provide the class that describes the shape of state, then the reducer, initial state and an array of middleware. Then I also dispatch the Get actions for each of our pages, which will result in our store having the updated lists of notes, news and the darkTheme setting right in the beginning of the app execution.



```
final Store<AppState> store = Store<AppState>(
    appStateReducer,
    initialState: AppState(),
    middleware: [
      settingsMiddleware,
      newsMiddleware,
      notesMiddleware,
    ],
); // Store

store.dispatch(GetSettings());
store.dispatch(GetNotes());
store.dispatch(GetNews());
```

Figure 62. Redux store initialization

Then this store is provided to the main MyApp widget as a constructor parameter. The topmost widget of MyApp is StoreProvider, which is able to provide our store to its children anywhere in the app, exactly the same way as the Provider package covered in the previous sections. Figure 63 shows the StoreProvider together with the StoreConnector, which is used to consume state from the store and to rebuild its children when there are any changes. Converter defines the particular part of the store in response to the changes of which we want to rebuild the children and that we want to use in the builder. In this case, we are watching for darkTheme setting, and then using it in the builder function to define which theme should be selected.



```
@override
Widget build(BuildContext context) {
  return StoreProvider<AppState>(
    store: store,
    child: StoreConnector<AppState, bool>(
      converter: (store) => store.state.darkTheme,
      builder: (context, darkTheme) {
        return MaterialApp(
          title: 'Flutter State Management',
          theme: darkTheme ? ThemeData.dark() : ThemeData.light(),
          home: MainScreen(title: 'Flutter State Management'),
        ); // MaterialApp
      }
    ), // StoreConnector
  ); // StoreProvider
}
```

Figure 63. StoreProvider

With notes, news and the settings screen the connection to the store is done exactly the same way as shown in Figure 63. Finally, if we just want to dispatch an action (for instance, to add a new note), but there is no need to rebuild the widget on state updates, we can use the same approach as with the Provider package:

final store = StoreProvider.of<AppState>(context, listen: false);

After obtaining the store in such a way, we can dispatch any action to it.

Benefits and drawbacks

Below you can see a list of the most significant benefits of the Redux state management approach:

1. Unidirectional circular flow of data is guaranteed
2. State is immutable
3. It is predictable in synchronous situations
4. It has the “time travel” feature – due to the circular and unidirectional data flow you are able to keep the previous states of the app with the actions performed and reverse everything easily
5. It is easy to debug because of “time travel” and ability to know what exactly happened with the app state and what actions led to an error
6. It offers very good testability

7. It is highly scalable

It also has some drawbacks:

1. It has a lot of boilerplate code
2. It is difficult to study, especially for people with no prior React experience
3. Redux is very good in synchronous operations, but when it comes to the asynchronous operations, it can become more complicated

Based on the benefits and drawbacks that were found, we can make an assessment of the criteria defined in Section 3.2. It can be seen in Table 10.

Table 10. Redux assessment

Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
difficult	a lot	no	yes	good	good

Redux is difficult for those unfamiliar with functional programming. Flutter does not require any functional programming knowledge, so chances are that Flutter developers might be not very confident with its concepts. It also has a lot of boilerplate code, just like BLoC. However, the fact that it enforces its own architecture with unidirectional flow of data results in good scalability. Its predictability and immutability of state provide good testability. It is the only approach out of the seven that has the time travel feature, which is also good for testability and debugging.

When to use

Redux is a good solution for large applications due to the benefits of testability, scalability, ease of debugging and predictability that it provides. However, it requires some special knowledge and the ability to understand its core concepts which are not easy to grasp. For smaller applications, the advantages of Redux usually do not outweigh the disadvantages. Also, React developers are usually familiar with Redux, so this state management approach can be used by developers with React background coming to Flutter.

3.5 Selecting the most suitable approach

Now we have the answer to the main question of this thesis, which can be formulated as How to choose the state management approach? The readers can simply define the importance or weight of each criteria for their project, and then see which approach matches their needs the most. Table 11 contains the assessment of each of the seven approaches based on the criteria defined in Section 3.2.

Table 11. The final comparison table of the approaches

Approach	Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
setState()	easy	a lot	no	no	bad	bad
InheritedWidget	difficult	average	no	no	average	bad
Provider	easy	a little	no	no	good	average
GetX	easy	a little	no	no	good	average
BloC	difficult	a lot	no	no	the best	good
MobX	easy	a little	yes	no	average	average
Redux	difficult	a lot	no	yes	good	good

The most suitable approach can be determined from this table, and then studied in more detail in Chapter 3. For example, if simplicity is the main criteria, we should consider setState, Provider, GetX and MobX. If the time travel feature is vital, then we go for Redux. If we need good scalability, BLoC, Redux, Provider and GetX are the options. If we also need something that is easy to test and debug, we should choose between BLoC or Redux, depending on whether we prefer reactive or functional programming. Combining the criteria will usually leave one to several possible options. Even if there is a situation when it is not possible to determine the right approach from the table, Chapter 3 still has enough information to figure it out.

4 CONCLUSION

The main purpose of the study was to categorize the existing state management approaches in Flutter based on some common technologies lying behind and find out a way to select the most suitable approach for different use cases. Instead of focusing on the particular use cases, a set of decision-making criteria was determined, and the approaches were analyzed and compared based on these criteria. A detailed proof of relevance of the problem was given in the theoretical part of the thesis. We made sure that it is highly possible that Flutter can become the most popular cross-platform development tool in the near future. Then the state management approach selection ambiguity problem was explained. In short, there are too many different state management approaches which makes the selection very difficult. Since Flutter becomes more and more popular, this problem is very relevant.

The goals of this thesis were fully achieved. The theoretical part contains a decent introduction to the mobile development market and the Flutter technology in particular. The most important result of the theoretical part is that the most popular state approaches were listed and categorized by a technology/programming paradigm (Figure 9). This is something new and valuable for the field of study. The main result of the practical part is a very detailed answer to the question How to choose the state management approach? It was achieved by describing seven most popular state management approaches, showing how to implement the apps based on these approaches in practice, analyzing the benefits and drawbacks and comparing them based on the predetermined criteria. In short, developers should determine which criteria from Section 3.2 are more important than the others for their particular use case. Then they can use Table 11 to find out the most suitable state management approach, and also refer to the practical implementation in Chapter 3 for more detailed comparison. The source code of the apps is available in my Github repository (Slepnev 2020). A person who has read the practical part should see the complete picture of the existing state management approaches and understand how to select the most suitable approach for his or her particular needs.

Since Flutter is a new and fresh technology in the world of mobile development, there are not so many materials on this topic, and very few studies have been made on that. The existing studies generally were not covering all existing approaches. It makes this thesis one of the most complete works on this topic. It combines a lot of research and practice in one place. It definitely contributes to the existing literature and helps the Flutter developers who read this by explaining the pros and cons, differences and similarities, the best use cases of each of the most popular state management approaches. This thesis brings order to the field of study that was very chaotic before.

The results that we have in this thesis mean that we are now able to choose the most suitable state management approach for each particular use case based on the requirements for this case. Previously, the selection process was quite random. It was most usually based on some personal preference of the developer and could result in increasing the time spent on the project, which automatically means that the costs are increased as well. Since Flutter is a tool for reducing the costs, developers should make the full use of it, and choosing the most suitable state management approach is an important point.

The study had some limitations. For example, we could not conduct a proper study on scalability and testability of the approaches and had to rely on the general experiences of other people found online. The complexity criteria is quite subjective even though we tried to use information from various sources. These limitations may have affected the results in some way, but definitely not significantly, so the general results are true in any case.

I am very happy with the results. I have also obtained a lot of useful information for myself; I understand the theoretical concepts behind state in the apps much better now. I am also now familiar with many approaches to state management and know how to use them in practice. This is a very good professional skill, because state management is needed not only in Flutter, but also in such frameworks as React, Vue and so on. Understanding how the things work under the hood can help me apply my state management knowledge in other fields as well. I am also very happy that I have managed to create some new information by putting together a lot of previously existing information.

State management selection is one of the most difficult issues in Flutter development which may sometimes define the success of the whole project. There are so many solutions existing out there, and it is very important to select the most suitable one. The public information that existed before this thesis was written did not provide a clear picture of all existing approaches and their best use cases. This gap is now filled.

REFERENCES

Android developers. 2020. Guide to app architecture. WWW Document. Available at: https://developer.android.com/jetpack/guide [Accessed 25 October 2020].

Chamley, C. 2014. When demand creates its own supply: saving traps. Review of Economic Studies, 81(2), 651–680.

Dart Programming Language Specification. 2019. p.9.

Dart Website. 2020. Main page. WWW document. Available at: https://dart.dev/ [Accessed 5 October 2020].

Eisenman, B. 2015. Learning React Native. O'Reilly Media, pp.1–2.

Flutter API Docs. 2020. InheritedWidget class. WWW Document. Available at: https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html [Accessed 29 October 2020].

Flutter Website. 2020a. Flutter for React Native developers. WWW document. Available at: https://flutter.dev/docs/get-started/flutter-for/react-native-devs [Accessed 4 October 2020].

Flutter Website. 2020b. Main page. WWW document. Available at: https://flutter.dev/ [Accessed 5 October 2020].

Flutter Website. 2020c. Flutter architectural overview. WWW document. Available at: https://flutter.dev/docs/resources/architectural-overview [Accessed 6 October 2020].

Flutter Website. 2020d. FAQ. WWW document. Available at: https://flutter.dev/docs/resources/faq [Accessed 6 October 2020].

Flutter Website. 2020e. Introduction to widgets. WWW document. Available at: https://flutter.dev/docs/development/ui/widgets-intro [Accessed 8 October 2020].

Flutter Website. 2020f. Differentiate between ephemeral state and app state. WWW document. Available at: http://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app [Accessed 8 October 2020].

Flutter Website. 2020g. Introduction to declarative UI. WWW document. Available at: https://flutter.dev/docs/get-started/flutter-for/declarative [Accessed 8 October 2020].

Flutter Website. 2020h. Start thinking declaratively. WWW document. Available at: https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative [Accessed 8 October 2020].

Flutter Website. 2020i. List of the state management approaches. WWW document. Available at: https://flutter.dev/docs/development/data-and-backend/state-mgmt/options [Accessed 14 October 2020].

Flutter Website. 2020j. Flutter Community. WWW Document. Available at: https://flutter.dev/community [Accessed 15 October 2020].

Flutter Website. 2020k. Differentiate between ephemeral state and app state. WWW Document. Available at: https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app [Accessed 18 October 2020].

Flutter Website. 2020l. Adding interactivity to your Flutter app. WWW Document. Available at: https://flutter.dev/docs/development/ui/interactive [Accessed 21 October 2020].

Global Mobile Market Report. 2020. Newzoo.

Grand View Research. 2020. Mobile Application Market Size & Share Report, 2020-2027. WWW document. Available at: https://www.grandviewresearch.com/industry-analysis/mobile-application-market [Accessed 1 October 2020].

Hu, H., Wang, S., Bezemer, C.P. and Hassan, A.E. 2019. Studying the consistency of star ratings and reviews of popular free hybrid Android and iOS apps. Empirical Software Engineering, 24(1), pp.7-32.

Nayebi, M., Adams, B. & Ruhe, G. 2016. Release Practices for Mobile Apps--What do Users and Developers Think? IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, in Suita, Osaka, Japan, March 14-18, 2016.

Omotunde, T., 2019. Flutter: Everything is a Widget Series -- Part 1: Where Flutter fits in. WWW document. Available at: https://dev.to/topeomot/flutter-everything-is-a-widget-series-part-1-where-flutter-fits-in-940 [Accessed 6 October 2020].

Pub.dev. 2020a. Pub.dev policy. WWW Document. Available at: https://pub.dev/policy [Accessed 17 October 2020].

Pub.dev. 2020b. Package scores & pub points. WWW Document. Available at: https://pub.dev/help/scoring [Accessed 17 October 2020].

Pub.dev. 2020c. sqflite. WWW Document. Available at: https://pub.dev/packages/sqflite [Accessed 25 October 2020].

Pub.dev. 2020d. dio. WWW Document. Available at: https://pub.dev/packages/dio [Accessed 25 October 2020].

Pub.dev. 2020e. provider. WWW Document. Available at: https://pub.dev/packages/provider [Accessed 30 October 2020].

Pub.dev. 2020f. getx. WWW Document. Available at: https://pub.dev/packages/get [Accessed 1 November 2020].

Pub.dev. 2020g. flutter_bloc. WWW Document. Available at: https://pub.dev/packages/flutter_bloc [Accessed 3 November 2020].

Reddit. 2020. Redux Compact: Make Redux fun again. FlutterDev subreddit. WWW Document. Available at: https://www.reddit.com/r/FlutterDev/comments/in4rwu/redux_compact_make_redux_fun_again/ [Accessed 16 October 2020].

Slepnev, D. 2020. Flutter state management. Github repository. Available at: https://github.com/sdim2016/flutter-state-management [Accessed 7 December 2020].

Statcounter. 2020a. Mobile Operating System Market Share Worldwide. WWW document. Available at: https://gs.statcounter.com/os-market-share/mobile/worldwide [Accessed 2 October 2020].

Statcounter. 2020b. Mobile Operating System Market Share United States Of America. WWW document. Available at: https://gs.statcounter.com/os-market-share/mobile/united-states-of-america [Accessed 2 October 2020].

Statista. 2019. Number of smartphone users worldwide from 2016 to 2021. WWW document. Available at: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ [Accessed 1 October 2020].

Statista. 2020. Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020. WWW document. Available at: http://statista.com/statistics/869224/worldwide-software-developer-working-hours/ [Accessed 3 October 2020].

Wikipedia. 2020a. Flutter (software). WWW document. Available at: https://en.wikipedia.org/wiki/Flutter_(software) [Accessed 4 October 2020].

Wikipedia. 2020b. Skia Graphics Engine. WWW document. Available at: https://en.wikipedia.org/wiki/Skia_Graphics_Engine [Accessed 7 October 2020].

Wikipedia. 2020c. State management. WWW document. Available at: https://en.wikipedia.org/wiki/State_management [Accessed 8 October 2020].

Wikipedia. 2020d. Data access object. WWW Document. Available at: https://en.wikipedia.org/wiki/Data_access_object [Accessed 27 October 2020].

Wikipedia. 2020e. Redux (JavaScript library). WWW Document. Available at: https://en.wikipedia.org/wiki/Redux_(JavaScript_library) [Accessed 6 November 2020].

Yale University. 2020. World Population: 2020 Overview. WWW document. Available at: https://yaleglobal.yale.edu/content/world-population-2020-overview [Accessed 1 October 2020].

YouTube. 2018. Keep it Simple, State: Architecture for Flutter Apps (DartConf 2018). Video. Available at: https://www.youtube.com/watch?v=zKXz3pUkw9A [Accessed 20 October 2020].

YouTube. 2019. Pragmatic State Management in Flutter (Google I/O'19). Video. Available at: https://www.youtube.com/watch?v=d_m5csmrf7I [Accessed 20 October 2020].