State Management and Software Architecture Approaches in Cross-platform Flutter Applications

Michał Szczepanik[a] and Michał Kędziora[b]
Faculty of Computer Science and Management,
Wroclaw University of Science and Technology, Wroclaw, Poland

Keywords: Mobile, Flutter, Software Architecture, State Management.

Abstract: Flutter is an open-source cross-platform development framework. It is used to develop applications for Android, iOS, Windows, Mac, Linux, and web. This technology was released on December 4, 2018, and it is quite young technology with a lack of good architectural patterns and concepts. In this paper authors compared state management approaches used for Flutter applications development and architecture. They also proposed a combination of two approaches that solve the main problem of existing approaches related to global and local state management. The proposed solution can be used for development even complex and big Flutter applications.

1 INTRODUCTION

Nowadays, almost all type of business needs a mobile application to existing. The cost of its development depends on complexity and requirements according to market coverage. To reduce it usually hybrid or multiplatform (cross-platform) solutions are used. Unfortunately, this kind of solution usually uses totally different patterns and architectural concepts compared to native Android or iOS applications. There is typically a blocker, or the main reason of project delays or even fail. There were a lot of hybrid (cross-platform) technologies like PhoneGap, Sencha, Cordova, Ionic, Xamarin and many more. Most of them are not in use or it is their endgame now, mostly because they were limited and needed knowledge from many areas including framework itself and platforms natively. Currently, only React Native and Flutter are in the game for most of the new hybrid projects.

1.1 React Native vs Flutter

The way how these two frameworks work is totally different: React Native uses the same fundamental UI building blocks as regular iOS or Android apps and the Java Script code runs in a separate thread and communicates with native modules through a bridge.

Flutter, on the other hand, is ahead of time compiled to a machine code (arm/x86) and provides better performance and even security related to difficulties of reverse engineering (Kedziora, 2019). Not only the UI components are compiled, but the whole logic also. Sometimes Flutter apps are even faster than native Android application, but it depends mostly on device type and operating system version.

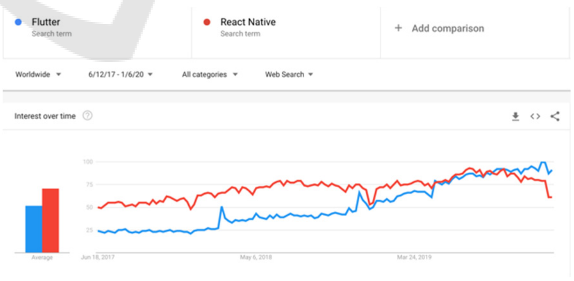

Figure 1: Flutter and React Native search trend during last 3 years, based on https://trends.google.com/.

A developer can use JavaScript for React Native and Dart for Flutter. The biggest mistake done by many companies is to ask JS/React developers to design and develop a big and complex mobile app.

[a] https://orcid.org/0000-0001-9801-992X
[b] https://orcid.org/0000-0002-7764-1303

Szczepanik, M. and Kędziora, M.
State Management and Software Architecture Approaches in Cross-platform Flutter Applications.
DOI: 10.5220/0009411604070414
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 407-414
ISBN: 978-989-758-421-3

Web and mobile are different worlds and understanding of iOS and Android ecosystem is much more difficult to learn than React Native. Additionally, for most of the hybrid applications there is a risk that some parts of functionalities will need separate native implementation and developer who knows it is a big value. For React Native application the knowledge about the mobile operating system and even its edge cases is mandatory. For Flutter applications, components code is system independent, which does not require that deep level of platform knowledge. That why the popularity of this platform and its community growing that fast, see figure 1.

1.2 Why Flutter

Flutter as a framework is very promising and right now has a big dev community. Even currently we can find complex apps in the market which are based on Flutter, like Alibaba, Google Ads, Reflectly, Birch Finance, Hamilton Musical, Hookle (Skuza, 2019).
In the Authors opinion, this technology is a good choice for small and medium-size applications or when content and basic features require constant iteration.
The technology potential is also big as during Flutter interact conference Google introduces support for web applications (Sneath, 2019). Dart language is also the fastest-growing programming language nowadays. Its list features added during the last two years is also big and includes extension functions, null safety support.

2 SOFTWARE ARCHITECTURE

There are many definitions of software architecture. According to The Institute of Electrical and Electronics Engineers Standards Board definition (IEEE std 1471-2000, 2007) it is the basic structure of the system which includes its components, interrelationships, way of work and rules establishing the way of its construction and development. Other definitions from literature (Knodel, 2017) (Abboud, 2017) (Martin, 2017) are similar to general one: software architecture is the defining and structuring of a solution that meets technical and operational requirements. Software architecture optimizes attributes involving a series of decisions, such as security, performance, and manageability. These decisions ultimately impact application quality, maintenance, performance, and overall success. For current research Authors define it as a structure of structures of the system which comprise the software elements, the externally visible properties of those elements and relationships among them.

3 EXISTING STATE MANAGEMENT APPROACHES

In Flutter everything is a widget and additionally, user interface (UI) depends on the state (Zammetti, 2019). Most of the samples and first Flutter applications were designed in a way where logic and UI are mixed, this caused that code was really difficult to manage and maintain (Fayzullaev, 2018). That why managing state in an application is one of the most important and necessary processes in the life cycle of a Flutter application. Unfortunately, this technology is very young and there are no general patterns and good practices defined. During the last two years only, few patterns mostly form React world (Paul, 2016) was ported to Flutter:

- ScopedModel,
- Redux (Paul, 2019),
- BLoC,
- MocX.

They mostly use data flow and reactive programming. Each of them supports global or local state. Global state is the main state which can be accessed in the whole app for example a user is logged in. Local state is related to only one component of the application for example screen or widget.

3.1 ScopedModel

ScopedModel is a set of utilities that allow passing a data Model from a parent Widget down to its descendants. It is one of the most basic concepts for Flutter application and was designed by the Fuchsia OS development team. ScopedModel concept uses 3 classes:

- Model,
- ScopedModel,
- ScopedModelDescendant.

A Model is a class that holds the data and business logic related to the data. It is implemented as an observable (listenable) interface and can notify others who might be interested in knowing when a change was applied.

ScopedModel is a main component, similar to a Provider, which holds the Model and allows:

- the retrieval of the Model,

- the registration of the context as a dependency of the underlying InheritedWidget, when it is requested.

The ScopedModel is based on an AnimatedBuilder which listens to notifications sent by the Model and then rebuilds an InheritedWidget which, will be requested all the dependencies needed to rebuild.

ScopedModelDescendant is used to find the appropriate ScopedModel in the Widget tree. It will automatically rebuild it whenever the Model notifies that change has taken place.

This concept is good for small applications when only a few shared (global) states are used.

3.2 Redux

Redux is an Application State Management framework and its main objective is to manage a global state in the application (Paul, 2019). Mainly used in React applications, but it also ported to the Flutter framework.

Redux architecture uee the following principles:

- Unidirectional data flow,
- one Store,
- Actions,
- MiddleWare,
- Reducers.

A Store acts like the orchestrator of Redux. The Store mainly:

- stores only one State,
- exposes one entry point, called dispatch which only accepts Actions in arguments,
- exposes one getter to fetch the current State,
- allows to register or unregister to be notified via StreamSubscription of any changes applied to the State,
- dispatches the actions and the store to the first MiddleWare
- dispatches the actions and the current state to a Reducer (which might be a façade for several reducers)

Actions are the only types of input accepted by the Store access point. Actions, combined with the current State are used by the Middleware and Reducer to process some function, which could lead to amending the State. Actions only describe what happened and do not store any data.

A Middleware is a function which is usually running asynchronously, based on an Action or state. A Middleware simply uses a State or an Action as a trigger but does not change the State itself.

A Reducer is a synchronous function which does some processing based on the Action and the State. The outcome of the process might lead to set a new State. The Reducer is the only component allowed to change the State.

It is important to note that, according to Redux recommendations and good practices, there can be only one single state store per application. To split the data handling logic, it is advised to use reducer composition instead of many stores. It is also not recommended for application which required integration with cloud base storage like Firebase according to limited state management and usually complex data storage which need to be managed.

3.3 BLoC

The Business Logic Component pattern or as it is widely known the BLoC pattern is a state management system for Flutter. It is recommended by Google developers to be used in the applications. It helps in managing state and make access to data from a central place in your project.

The BLoC pattern does not require any external library or package as it simply relies on the use of the Streams. The concept is very similar to MVVM (Model – View – ViewModel) but required usage of streams. However, for more friendly features (e.g. Subject), it is very often used with the RxDart (ReactiveX extension for Dart) package. In this pattern data are flowed from the BLoC to the UI or from UI to the BLoC in the form of streams.

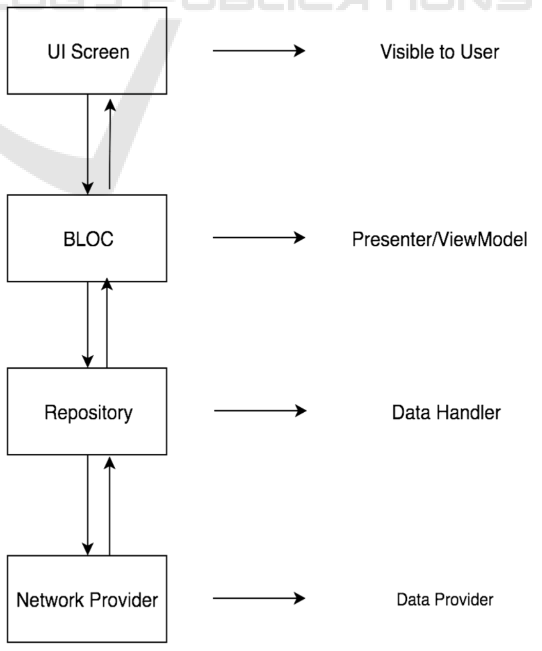

Figure 2: BLoC architecture schema.

The Flutter version of BLoC pattern relies on:

- StreamController
- StreamBuilder
- StreamSubscription
- BlocProvider

A StreamController exposes a StreamSink to inject data into the Stream and allow Stream to listen to data which are inside the Stream.

A StreamBuilder is a Widget which listens to a stream and rebuilds when new data is emitted by the it.

A StreamSubscription is a interface that allows to listen to the data being emitted by a stream and react.

A BlocProvider is a convenient Widget, used to hold a business logic and rules. It makes them available to descendant Widgets.

3.4 MobX

MobX is a state management solution that helps in managing the local state within Flutter application.

Some of the core principles of MobX are:

- it can have multiple stores to handle the state of the application,
- anything that can be derived from the state without any further interaction is a derivation,
- action is any piece of code that can change the state,
- all derivations are updated automatically and atomically when the state changes.

Unlike other state management patterns in Flutter such as BLoC, which was built on the principle of using streams to propagate changes, and Redux, which was built on the philosophy that an application possesses a single source of truth from which widgets inherit, MobX was built on the simple philosophy that anything that can be derived from the application state, should be derived. It uses transparent functional reactive programming, MobX provides coverage for all properties in an application state that are defined with the likelihood to change and rebuilds the UI only when properties change. Unfortunately, implementation of this pattern without any external library is very difficult, which provide some limitations and additional dependencies

4 BLoC WITH REDUX LIKE STORE

Both most popular state management concepts, which are BLoC and Redux, have some disadvantages.

BLoC should not be responsible to keep the application's state. It was designed to control many more local states. Pure BLoC should delegate this responsibility to some other component which is dedicated to state management in way which Redux concept is better.

The BLoC pattern is a great way to encapsulate business logic and Redux is a great state management paradigm. Combining the two of them can create a clean logic layer in the application, see figure 3. Combination of BLoC which use many state stores in Redux design way should allow developers to design clean and easy to maintain code (Martin, 2008).

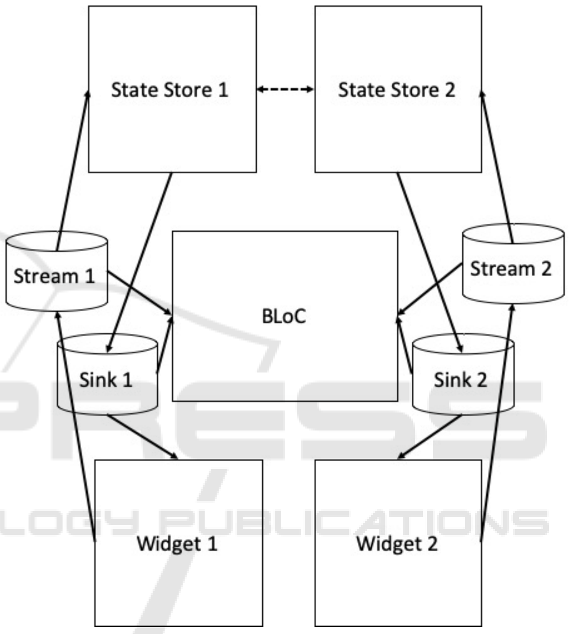

Figure 3: BLoC with multi state stores (own work).

5 ARCHITECTURES COMPARISON

The comparisons of the architecture approaches is done on two samples Flutter project. Each project is written in four versions:

- ScopedModel,
- Redux
- BLoC,
- BloC with store (proposed).

The MobX solutions required specific library which hide most of the complexity, so it cannot be correctly compared with other solutions. Each architecture will be compared based on: code complexity, code execution, architecture and flow complexity, number of rebuilds (UI performance) and code isolation.

5.1 Case 1: User Login Screen

This common use-case for many applications is very interesting since it involves some type of application State. In this example the page to act as follows:
before login

- two text areas to provide username and password with error handling,
- login button which start process,
- progress indicator which simulated authentication process is on-going,
- the username of the authenticated user together with a button to log out.

5.1.1 Code Complexity

There is no big difference in code complexity because this sample is basic. It handles only few states like password and username are incorrect, user is logged in or not. There are only two logic points which are changing the state login button and logout which can be done from any scree in the application. For both ScopedModel and BLoC solutions there were need to inject their respective model and bloc on top of the MaterialApp, to be available to logout from anywhere later on.

In Redux, the solution needs to use to many more files compered to otherers even when all actions are stored in one place. The ScopedModel solution requires fewer files as the model control and store both the data and the logic. The BLoC solution requires one additional file, compared to the ScopedModel, because it requires to split model and logic. Proposed solution which is BLoC with store works in similar way as standard BLoC, but depends on components in the app the number of file can be increased, additionally sharing of user login state can provide some complexity in the application.

5.1.2 Code Execution

The number of lines of code which is executed is the biggest in Redux. It is mostly caused by the way how a reducer is written which based on condition evaluations such as: "if action is … then", and the same applies to the MiddleWares.

Because of the implementation made by the flutter_redux package, a StoreConnector requires a converter, which sometimes is not necessary. This converter is meant to provide a way of producing a ViewModel.

Other solutions like ScopedModel and both BLoC based solutions seem to be the ones which require less code execution.

5.1.3 Architecture and Flow Complexity

In case of Redux the code is relatively simple and easy to follow because there is only an Action that triggers all MiddleWares to be run in sequence and then the Reducer which needs to do things based on a comparison on an Action type. Unfortunately, when additional logic and use cases will be added to the application, it will require refactoring and usage of reducer composition.

The ScopedModel solution is the one which leads to the simplest code: there is only call a method which updates the model that notifies the listeners. However, it is not obvious for the listeners to know the reason why they are being notified since any modification to the model generates notifications, even when it is not required by current listener.

The BLoC solutions are a bit more complex as it involves the notion of Streams.

5.1.4 Number of Widgets Rebuilds

The number and part of Widgets tree which is affected by rebuilds is one of the mandatory parameters for the architecture. Flutter was designed to provide 60 and more frame per seconds, that why it is important to reduce number of Widgets which are rebuild after any state change in the application. Each rebuild may affected performance of the application an reduce number of frames.

The ScopedModel solution is the one that produces the more builds since each time a Model notifies its listeners, it rebuilds the whole tree under the ScopedModel.

The flutter_redux library internally uses the notion of Streams to react on changes applied to the State, so for basic implementation without context only the StoreConnector will be rebuilt. This makes the flutter_redux implementation the most optimal for that case from a rebuild perspective.

In the BLoC solutions which based on StreamBuilder there is similar situation as for Redux – only related to current state part of widget tree will be rebuilt.

5.1.5 Code Isolation

In Redux Reducers and MiddleWares are mostly top-level functions and methods and not part of a class. As a consequence, nothing would prevent calling them outside of the Redux Store scope and it can difficult to manage in big, complex applications.

ScopedModel and BLoC tend to prone code isolation: one specific class for the model or the BLoC.

5.1.6 Conclusions

For this specific case in which there was mostly global state the Redux solution can be used, but the advantage of it is only little. It was the best in case of performance and number of rebuilds, but the code isolation can be problematic for maintenance.

For ScopedModel and BLoC some additional effort is required for implementation to handle logout functionality in proper way. Proposed solution which is BLoC with storage do not provide any benefits and works in similar way as standard known form literature BLoC patter.

5.2 Case 2: Dashboard Application

This case is application in which user can dynamically add some panels to the dashboard which presents different source of data. In the sample application user can see currency exchange rate. Additionally, the user may turn on or off the real-time data fetching for each of panels, individually.

5.2.1 Code Complexity

Pure Redux principle required to use one Store per application which caused that this application is really difficult to implement in pure Redux. ApplicationState required to remember and handle each individual panel. When multiple Storage will be used (break Redux principle) the implementation is simpler, and the code is cleaner.

The ScopedModel and BLoC versions are very similar. Proposed solution (BLoC with storages) additionally provides isolations between panels and their states which allow to reduce complexity of data synchronization.

5.2.2 Code Execution

This is similar to case 1 (chapter5.1.2).

Redux executes much more code than ScopedModel and both BLoC solutions as the reducer is based on condition evaluations. In addition, three instances of StoreConnector are needed: to add new panel, to fetch currency data and to control synchronization status.

ScopedModel requires additional code execution than BLoC, because it relies on listenable widget to rebuild each time the Model changes. This requires, additionally injector (ScopedModel) and two ScopedModelDescendant (for history and synchronisation status) per Panel.

BLoC is the solution which executes the less code. Per Panel, it requires: StreamBuilder to display the stats and additional StreamBuilder to handle the synchronization status.

For proposed solution compered to bloc additional store is needed per panel, but it allows to reduce data synchronisation complexity and call update of the UI only when data are really changed.

5.2.3 Architecture and Flow Complexity

The Redux solution is the most complex as it requires the dispatching of Actions at 3 different levels: to add new panel, to fetch currency data and to control synchronization status.

The complexity of ScopedModel and BLoC solutions is only located at the Model and BLoC levels. Each panel has its own Model or BLoC and the code is much less complex end easier to maintain compared to Redux. In proposed solution architectural complexity is similar.

5.2.4 Number of Widgets Rebuilds

Pure Redux solution is the one that causes the most of rebuilds. As the implementation, based on one Store per Application, each time a change applies to the ApplicationState and everything need to be rebuilt. No meter which action user do add new panel or turn off/on synchronization for one of them or there will be an update of data for one of currency the widgets tree will get information about changed state.

In non-standard implementation of Redux with multi stores the number of rebuild will be reduced and separated per panels.

As regards the ScopedModel solution, the number of rebuilds is more limited than in Redux and it is done only per panel.

The standard BLoC block is the one that requires rebuilds only for specific widgets not whole panel as for ScopedModel.

The proposed solution additionally can reduce number of rebuilds which are additionally limited to cases when data related to state is really change.

5.2.5 Code Isolation

The problem of code isolation occurs only for Redux as data (state) can be changed from each part of the code. For other solutions they are isolated by Model or BLoC.

5.2.6 Conclusions

For this specific case, from both code complexity and rebuilds perspectives the BLoC and proposed BloC with storages are the best.

The Redux architecture is not optimal for this solution, but it is still possible to use it, mostly with multiple storages.

6 TESTABILITY

Automated testing falls into a few categories:

- A unit test tests a single function, method, or class.
- A widget test (in other UI frameworks referred to as component test) tests a single widget.
- An integration test tests a complete app or a large part of an app.

A well-tested app has many unit and widget tests, tracked by code coverage, plus enough integration tests to cover all the important use cases. This advice is based on the fact that there are trade-offs between different kinds of testing, seen below.

Table 1: Tests levels comparison.

Unit	Unit	Widget	Integration
Confidence	low	higher	highest
Maintenance cost	low	higher	highest
Dependencies	low	more	most
Execution speed	quick	quick	slow

For all architectures testability is on similar level. The only difference is between unit and widgets type of tests. For ScopeModel and Redux the logic isolation is minimal and most of test cases requires Widget level or on unit level with mocking. For BLoC and proposed solution logic is separated and implemented on pure Dart level which allow to easily implement unit level test even without mock.

7 COMPARISON SUMMARY

7.1 ScopedModel

Pros:

- ScopedModel allows to easily regroup the Model and its logic in a single location.
- ScopedModel does not require any knowledge of Streams, which is good entry point for beginners.
- ScopedModel can control global and local states.

Cons:

- ScopedModel does not provide any logic which allow to provide knowledge which parts of the Model were changed.
- Cause too many rebuilds – each time when a Model notifies its listeners.
- It can be used only for small not complex application.
- Requires the use of an external package with the risks that the package evolves with breaking changes.

7.2 Redux

Pros:

- Redux allows to centralize the management of state.
- Makes the state transition perfectly predictable and thoroughly testable.
- Support for MiddleWares in the flow to track logs or statistics.
- It forces the developer to structure the application and use Event, Action and MVVM.

Cons:

- One single Store (pure version).
- Use of top-level functions/methods.
- Too many “if then” comparisons at Reducers and MiddleWares levels.
- Too many rebuilds related to changes in stare.
- Lack of logic isolation.
- It can be used mostly for global state management.

7.3 BLoC

Pros:

- It allows easy to regroup the business logic in a single location.
- BLoC allows to determine with precision the nature of any changes.
- It allows to reduce the number of rebuilds.
- It uses streams which allows to add middlewares like logging or statistic collectors.
- It could be used for global and local state and logic control
- It does not require the use of any external package, as it can be easily implemented.
- It can be used for complex.

Cons:

- Required knowledge of streams
- Control of global state can be problematic

7.4 BLoC with Storage

Pros:

- It allows easy to regroup the business logic in a single location.
- It allows to separate storage per application component
- It allows to determine with precision the nature of any changes.
- It allows to determine real changes of data (according to store changes).
- It allows to provide store to control global state.
- It allows to reduce the number of rebuilds.
- It uses streams which allows to add middlewares like logging or statistic collectors.
- It could be used for global and local state and logic control.
- It does not require the use of any external package, as it can be easily implemented.
- It can be used for complex.

Cons:

- Required knowledge of streams.

8 CONCLUSIONS

The main goal of this research was a comparison of architectures and state management approaches, which can be used in Flutter applications. The proposed solution additionally reduces the number of rebuilds in a similar way as Redux for the global state, but it allows to work with both global and local state-oriented applications.

REFERENCES

Abboud M. 2017. Software Architecture Extraction: Meta-model, model and tool. Génie logiciel [cs.SE]. Université de Nantes.

Fayzullaev, J., 2018. Native-like cross-platform mobile development: Multi-os engine & kotlin native vs flutter.

IEEE std 1471-2000, 2007. The Institute of Electrical and Electronics Engineers Standards Board: Recommended Practice for Architectural Description of Software-Intensive Systems, ISO/IEC 42010:2007(E) IEEE Std 1471-2000, 2007

Kedziora, M., Gawin, P., Szczepanik, M., & Jozwiak, I., 2019. Malware Detection Using Machine Learning Algorithms and Reverse Engineering of Android Java Code. International Journal of Network Security & Its Applications (IJNSA) Vol, 11.

Knodel J. and Naab M., 2017. How to Evaluate Software Architectures: Tutorial on Practical Insights on Architecture Evaluation Projects with Industrial Customers. In IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, 2017, pp. 183-184.

Martin, R. C., 2017. Clean architecture. Prentice Hall

Martin, R. C., 2008. Clean code. Prentice Hall

Paul A., Nalwaya A. (2016) React Native Supplements. In: React Native for iOS Development. Apress, Berkeley, CA

Paul A., Nalwaya A. (2019) Solving Problems Differently with Flux and Redux. In: React Native for Mobile Development. Apress, Berkeley, CA

Skuza B., Mroczkowska A., Włodarczyk D., 2019. Flutter vs React Native – what to choose in 2020?, https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2020 (last accessed 12/30/19).

Sneath T., Scaramuzzi R., Fabbro A., 2019, Product Keynote (Flutter Interact '19), https://www.youtube.com/watch?v=ukLBCRBlIkk (last accessed 12/30/19).

Zammetti F., 2019. Practical Flutter: Improve your Mobile Development with Google's Latest Open-Source SDK, Apress.

APPENDIX

Code of tested applications in all variants is available on github https://github.com/pwr-mszczepanik/flutterarch.