Cross-Platform Development for Sailfish OS and Android: Architectural Patterns and “Dictionary Trainer” Application Case Study

Denis Laure, Andrey Vasilyev, Ilya Paramonov, Natalia Kasatkina
P.G. Demidov Yaroslavl State University
Yaroslavl, Russia
{den.laure, ilya.paramonov, andrey.vasilyev}@fruct.org, ninet75@mail.ru

Abstract—**With the widespread use of mobile devices, the role of mobile applications increases. Nevertheless, the variety of mobile platforms and the differences between them make the development of applications for multiple mobile platforms a highly resource-intensive and time-consuming task. The reason for this is the need for development of separate native applications for each platform. This issue can be overcome by developing cross-platform applications that, once created, can be launched on multiple platforms without any changes in source codes. In this paper, we present an approach for developing native cross-platform mobile applications for Android and Sailfish OS platforms, with the use of Qt Framework and Qt Quick based on the Flux architecture. Such applications have a native look and experience on each platform. The paper also presents the “Dictionary Trainer” application, that was developed using the described approach.**

I. Introduction

Nowadays, mobile devices of different types have become an integral part of people’s everyday lives [1]. Thus, the development of applications for mobile platforms becomes a highly essential task. Moreover, for a product to become successful, it is highly important and crucial for it to be released on multiple mobile platforms in order to cover a larger number of audience [2]. Notwithstanding, developing native applications for each platform can be a highly resource-intensive and time-consuming task, due to the fact that each platform requires to use its own tools for development and even different programming languages. This fact leads to the inability to reuse the source codes of an application from one platform and to the need to develop the application from scratch for each mobile platform [3].

Cross-platform development can be an adequate answer to meet this challenge, since it allows to create one application that can be launched on several platforms [4]. Therefore, such approach significantly reduces the amount of time needed to develop one cross-platform application, instead of having to develop several separate applications for each platform.

The development should be at first concentrated on most popular platforms. Currently, they are iOS and Android—two mobile platforms that almost completely share the mobile market [5]. Nevertheless, new platforms still appear aiming to receive a piece of market share. One of such platforms is Sailfish OS (https://sailfishos.org). The advantages of this platform are that it is open sourced and Linux-based. Moreover, it allows to launch native Android applications without any changes being made to them, thus bringing a lot of different applications to this platform and its users. Therefore, there are high chances of Sailfish OS breaking into the mobile market. However, because Sailfish OS is a relatively young platform, there are not as many native applications developed for it. Therefore, creating applications for this platform may allow developers to fill a niche on the market and gather a significant number of users for their product.

In this paper we describe an approach to creating native cross-platform mobile applications for two platforms: Android and Sailfish OS. For this purpose, Qt framework and Qt Quick are used. The applications are also follow Flux architecture.

The rest of the paper is organized as follows. In Section II we present and briefly describe possible architectures of cross-platform mobile applications developed with Qt framework and Qt Quick. Section III provides a thorough description of Flux architecture and describes how it can be implemented within the QML application. The approach for creating cross-platform mobile applications for Android and Sailfish OS is outlined in Sections IV. Section V presents the Dictionary Trainer application that was developed with the use of the described approach. Section VI concludes the paper.

II. Possible architecture models for QML applications

The architecture of the application plays a significant role in the whole development process. It determines quality attributes of the application, its high-level structure and behaviour, and the way it is going to evolve. Therefore, the choice of the architecture is one of the main aspects that determines the future success of the application [6].

Since there are no standard architecture guideline for the QML and Qt Quick applications, it needs to be chosen before starting the development process. The possibilities are: Model-View-Controller (MVC), Model-View-ViewModel (MVVM) and Flux.

The main idea of the MVC architecture is to separate the application into three components: Model, View and Controller [7], [8]. Model represents the data of the application and interfaces to manipulate it. View represents the visual part of the application, it presents the model in a user-friendly way and provides visual components to manipulate with it. Lastly, controller processes the user’s actions, allowing the user to manipulate the view.

The MVVM architecture is an evolution of the traditional MVC architecture. It modifies MVC and its approach to satisfy modern UI development platforms [9], [10]. Model in MVVM is the same as in MVC, while view takes the responsibilities of both view and controller of MVC, defining the GUI of the application. The viewmodel part of MVVM architecture represents a bridge between model and view, whose main aim is to convert data from model so that it can be easily managed by View.

The MVVM architecture is more suitable for QML applications, because it allows to logically separate the model and view. Following this architecture, view is declared in QML code, while model is defined with C++ code. Viewmodel can be described in QML code, as well as in C++ code. Nonetheless, spreading it between C++ and QML code leads to inconsistency and code fragmentation, which makes it difficult to maintain the application as it starts to grow [11]. On the other hand, implementing viewmodel in only one language may lead to an unnecessarily complicated code, because some functions are easier to be done in another language.

Moreover, MVVM has the same problem as MVC. The architecture works well for relatively small applications. Nevertheless, as the application grows, its architecture becomes increasingly complex and confusing. The growth may lead to a situation where the application has multiple views and models that are all inter-connected with each other. Making a small action in one view may result into a chain of changes in several other views and models that is difficult to track. Sometimes, these chain reactions may even result in an infinite loop. Maintaining such applications becomes an unnecessarily wasteful task [12].

Flux architecture was initially developed by Facebook for their JavaScript library React [13]. The main difference and advantage of such an architecture, compared to the traditional MVC architecture, is the well-structured unidirectional data flow. Such an approach allows to avoid the complexity of data flows in big applications that use the MVC or MVVM architectures.

Unlike MVVM and MVC, Flux architecture does not have the aforementioned issues and has additional benefits to these architectures, that were significant for our approach. Therefore, Flux was chosen as the core architecture for the approach.

III. FLUX ARCHITECTURE IN DEPTH

There are four crucial components in Flux: action, dispatcher, store and view. View has the same meaning as in MVVM—it represents the user interface of the application and its responsibility is to display information to the user. Whenever the user interacts with the view, it generates an action, which is just a message, containing the type and payload, i.e., some meaningful information. For example, if a user clicks on an item in a list, view may send an action of type *list_item_clicked* and a payload containing information about which item was clicked.

Store represents the state of the application and its logic. All actions are sent to dispatcher—the central hub of the application. There should be only one instance of dispatcher in the application. At the same time, stores can subscribe for particular actions in dispatcher, registering the callback that should be called whenever the action occurs. Each time an action comes to dispatcher, it notifies all stores subscribed for this action by calling the registered callbacks. A Store may then change its state and notify the corresponding views about this change, so that they may change themselves accordingly.

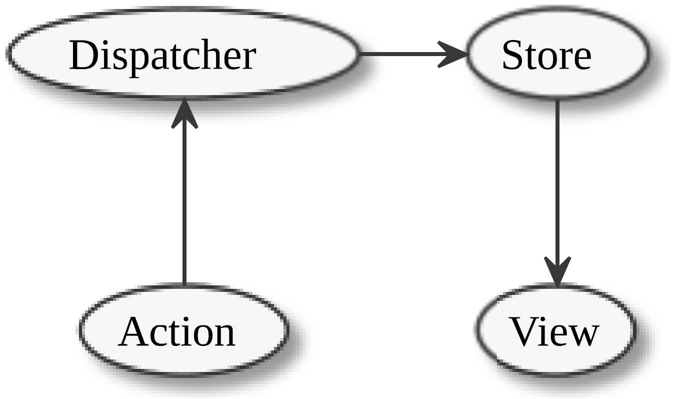


Fig. 1. The data flow in an application written with the use of Flux architecture

Such behaviour draws a clear unidirectional data flow that is shown in the Fig. 1. It can only start with an action going directly to dispatcher. Dispatcher then triggers all callbacks of the stores registered for this action that afterwards notify views about state changes. Views can then generate new actions according to the changes that go to dispatcher and start a new flow.

Flux architecture may be very helpful for applications with dynamic data, because it structures the data flow and, thus allowing to put connections between models and views in order [12].

Recently, Flux was implemented for QML as QuickFlux library (https://github.com/benlau/quickflux). The library allows to easily develop Qt Quick applications with Flux architecture. The main components of this library are: *AppDispatcher*, *AppListener*, and *ActionCreator*. *AppDispatcher* implements the dispatcher component of the Flux architecture. It is a singleton object that allows to add and remove listeners for particular actions. *AppDispatcher* is implemented in such a way, that it works with any actions that are created with the use of QuickFlux library tools. Therefore, there is no need for customization or modification of this component.

ActionCreator is a component that allows to create actions. The action is implemented by defining a signal within this component. The action is emitted by directly calling the corresponding signal implemented in *ActionCreator* passing the action payroll as a signal parameters. The component will then automatically process this action to *AppDispatcher*, which in turn informs stores that were subscribed for this action.

Finally, the *AppListener* component should be implemented by stores and is used to subscribe a store for a certain action and define the callback for this action.

Using QuickFlux library allows to easily and quickly create QML applications that follows Flux architecture. Thus, it was used for the approach described in the following section.

IV. Approach to development of cross-platform applications

This section describes a common approach for creating cross-platform applications for Android and Sailfish OS platforms with Qt Framework. The main idea is to separate the application sources for both mobile platforms into the following parts: C++ files, QML files, and other files. The approach describes the method of the organization for all of these groups of files. Its detailed explanation is presented below.

A. *C++ files*

In most cases, C++ sources are identical for both platforms, and therefore can be used without any platform-dependent differences. This is due to the fact that C++ sources are related to the non-visual parts of the application (until we do not create our own components, which is rare), that are already made platform-independent inside the Qt Framework. For instance, once the QuickFlux library is included in the project, it works on both Android and Sailfish OS platforms in the same way and without any changes.

Nonetheless, there are some parts of the the C++ code that should be compiled for only one platform. For instance, the initialization of the QML application differs on different platforms. The reason for that is that Sailfish OS uses libsailfishapp: Sailfish Application Library (https://github.com/sailfish-sdk/libsailfishapp), which is part of Sailfish SDK to initialize the application. It has *application()* and *createView()* methods that return instances of classes which are already prepared and boosted for Sailfish OS *QGuiApplication* and *QQuickView*. The simple example of the initialization code is presented below:

```
QGuiApplication* app = SailfishApp::application(argc, argv);
QQuickView* view = SailfishApp::createView();
view->setSource(SailfishApp::pathTo(
    "qml/sailfish/main.qml"));
view->showFullScreen();
QObject::connect(view->engine(), &QQmlEngine::quit, app,
    &QGuiApplication::quit);
return app->exec();
```

The Android platform is supported by the Qt Framework without a need for any additional components. Therefore, the initialization of the QML application for Android is done in a way common for QML applications. The example of such an initialization is presented below:

```
QGuiApplication app(argc, argv);
QQmlApplicationEngine engine;
engine.load(QUrl("qrc:/qml/android/main.qml"));
return app.exec();
```

Since such parts of the code should be separated depending on the platform, they can be framed with the *#ifdef* pre-processor directive, as presented below:

```
#if defined(Q_OS_ANDROID)
    // Android-specific code
#else
    // Sailfish OS-specific code
#endif
```

B. *QML files*

Since the native look and feel of applications on Android and Sailfish OS platform are completely different, QML files that describe views should be implemented separately for these platforms. The choice of the needed files is done while the application initializes, as described in the previous subsection. On this step, different main QML files are set for different platforms. These files represent the main application window and contain different information and components depending on the platform.

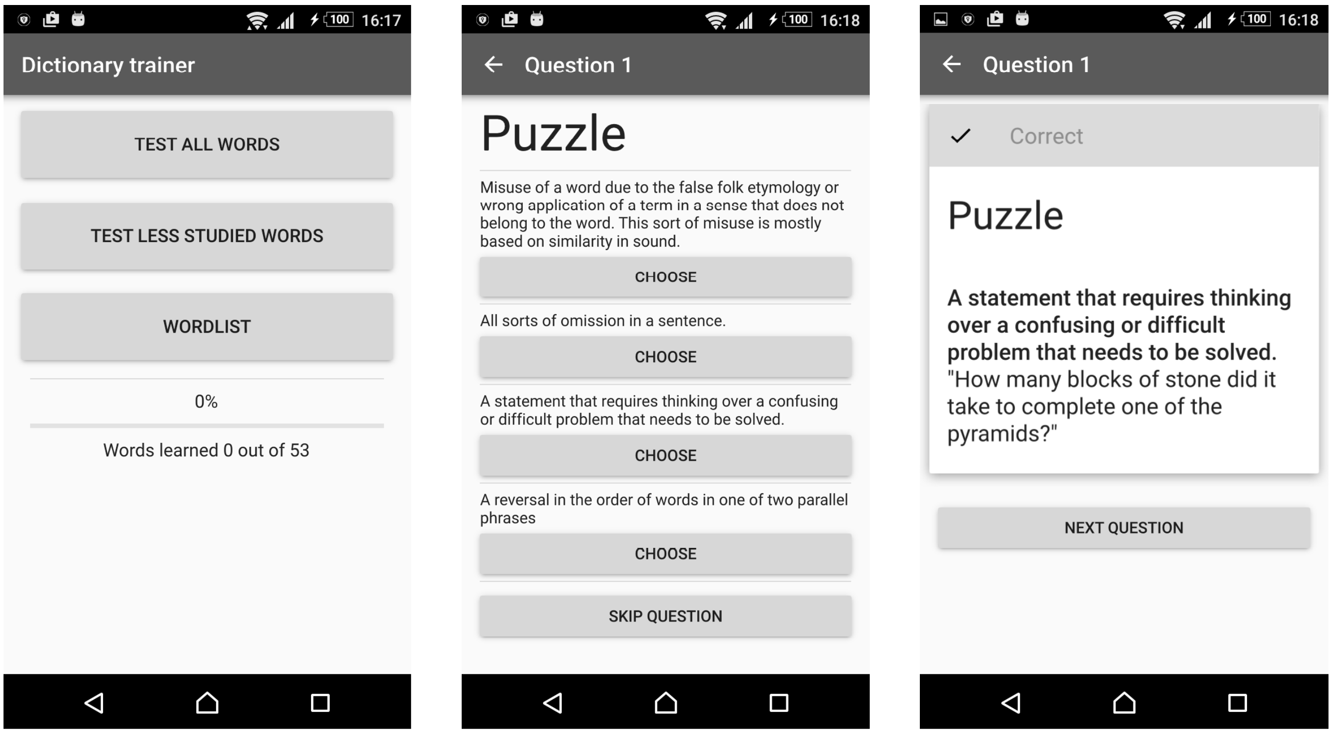
Sailfish SDK contains the Sailfish Silica framework that contains all common visual components of the application for Sailfish OS that ensure that the application looks natively on this platform. The main window is implemented as the *ApplicationWindow* component and, among other functions, also defines the initial page or screen of the application that is also defined separately for both platforms. The other pages are implemented separately as well.

Nevertheless, it is not only the look of the application that is different for both platforms. The behaviour of the application and its components also differs. Sailfish OS follows gesture-oriented interfaces, where the user mainly uses swipes to navigate through an application, show menus, etc. Components defined in the Sailfish Silica framework already implement these gestures. Moreover, the *PageStack* component in the Sailfish Silica framework allows to switch between application pages programmatically or by using common gestures.

Unlike Sailfish OS, there is no special library that provides natively looking Android components for Qt Framework. Therefore, one should use some external (for instance, Qt Quick Controls with Material theme) or a self-written library to attain the native look on this platform. The main window is implemented in the same way as for a regular QML application—using a *Window* component. To switch between the pages of the application, the *StackView* component Qt Quick Controls module from should be used.

Using different components to switch between the pages of the application (*PageStack* on Sailfish OS and *StackView* on Android) allows to ensure that particular pages will be shown only on the platform they were created for. In their turn, the pages are created with the use of particular components that are native for each platform, thus ensuring that every page looks and behaves natively for the platform it was created for.

The non-visual QML components (for instance, stores in terms of Flux architecture) can be used on both platforms without any changes, since they determine the internal application states and behaviour, such as data flows, that are common for the application, regardless of the platform it is running on.



Dictionary trainer

TEST ALL WORDS

TEST LESS STUDIED WORDS

WORDLIST

0%

Words learned 0 out of 53

Question 1

Puzzle

Misuse of a word due to the false folk etymology or wrong application of a term in a sense that does not belong to the word. This sort of misuse is mostly based on similarity in sound.

CHOOSE

All sorts of omission in a sentence.

CHOOSE

A statement that requires thinking over a confusing or difficult problem that needs to be solved.

CHOOSE

A reversal in the order of words in one of two parallel phrases

CHOOSE

SKIP QUESTION

Question 1

✓ Correct

Puzzle

A statement that requires thinking over a confusing or difficult problem that needs to be solved.
"How many blocks of stone did it take to complete one of the pyramids?"

NEXT QUESTION

(a) Main screen (b) Test question screen (c) Answer result screen

Fig. 2. Screens of the Android version of Dictionary Trainer

C. Other files

The *.pro* project file is shared between the platforms and contains the information needed to build the QML application. The application also shares its resource file between platforms that contains translation files, icons and other resources that are needed for the application (for instance, images).

Finally, there should be platform-dependent files, needed to compile the application for the particular platform. For Android it is the manifest file, gradle files, keystore that is needed to sign the application package. For Sailfish OS these files are: *.desktop* file, containing information for the application shortcut on the device's list of applications, and *.spec* and *.yaml* files that contain context information about the rpm package to be built. All of them should be included into the project.

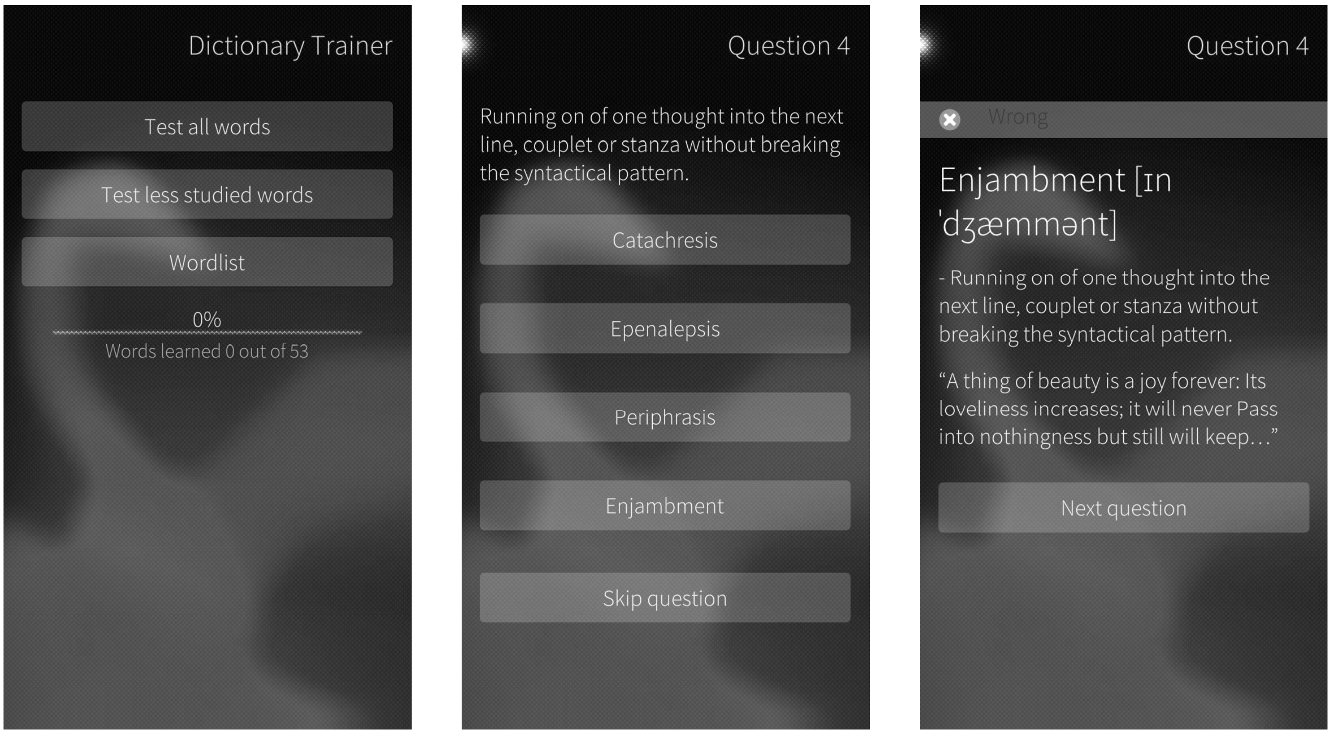
V. Case Study: Dictionary Trainer

One of the applications that we developed using the aforementioned approach and one that clearly illustrates it, is Dictionary Trainer. Its purpose is to help its users to learn new terms and enrich their vocabulary. The learning process consists of passing a test. There are three types of questions available in the test. The first one gives a definition of a term and the user has to choose an appropriate term for it. The second one gives a term and requires the user to choose the right definition. The last type of questions requires the answer to be entered manually. Additionally, the application provides a built-in dictionary which contains definitions and example sentences for every term. The progress of the user is tracked automatically as they advance in tests and learning.

The main screen of the application allows to start the testing or to go to the list of the words. The testing consists of showing one of the described above types of questions to the user one by one. After the user answers the question, the result page is shown, informing the user whether the answer was right or not. The list of the words represents all words in the application dictionary. By clicking on a particular word, the detailed information about the term is shown, including its definition, pronunciation and usage examples.

The user interface of Dictionary Trainer differs, depending on the platform that the application is running on. On Android it follows the Material design. Fig. 2 shows screens of the Android version of Dictionary Trainer. On Sailfish OS it looks native for this platform and uses swipes for navigation, which is common for this platform. The screens of the Sailfish OS version of the Dictionary Trainer are shown in Fig. 3.

C++ files are represented by the sources of the QuickFlux library, the *Settings* class, and the *dictionary_trainer.cpp* file that serves as the initialization point of the application. The first one is not customized for any of the platforms and is used as it is, as described in Section III. The *Settings* class is used to provide access to the standard Qt *QSettings* class from QML code allowing to call *QSettings*' methods from QML components. This class is used to save, store and retrieve application settings, and it works on Android and SailfisOS without any platform-specific code. Finally, the *dictionary_trainer.cpp* file consists of two blocks of code, where the initialization of the application is done separately for each platform. The initialization process looks the same for both platforms and follows the same steps: create instance of *QGuiApplication*, create instance of *Settings* class and register



Running on of one thought into the next line, couplet or stanza without breaking the syntactical pattern.

Enjambment [ɪn ˈdʒæmmənt]

- Running on of one thought into the next line, couplet or stanza without breaking the syntactical pattern.

“A thing of beauty is a joy forever: Its loveliness increases; it will never Pass into nothingness but still will keep…”

(a) Main screen (b) Test question screen (c) Answer result screen

Fig. 3. Screens of the Sailfish OS version of Dictionary Trainer

it for use within the QML code, set localization files depending on the default system locale, set the main QML file, and launch the application. Nonetheless, the applications uses the SailfishApp library to initialize on the Sailfish OS, which does not exist for the Android platform. Therefore, the initialization needs to be separate for both platforms, as it is explained in Section IV-A.

QML files that describe the pages and view components of the application are separated according to platforms, since these components look absolutely different on both platforms and use different frameworks and libraries. Moreover, the main QML file is also separated according to platform, because it is used as the starting point of the QML application and describes what page should be shown first.

In addition, there are some QML components that exist only on one of the platforms. For Android they are the special view components and the *StackView* component that manages the opened pages of the application. The example of the special component is the material card component. It is used, for instance, to display the results of the user's answer (Fig. 2c). The *StackView* component is not needed on the Sailfish OS, since such a component is already implemented in the Sailfish Silica library. Nevertheless, the Sailfish OS-only QML files include the description of the cover page of the application, which is not needed for Android, since this platform does not have such kinds of components. This cover page displays the current progress of the user and is shown in Fig. 4.

At the same time, source codes of the Dictionary Trainer contain QML files that can be used on both platforms without crucial changes, because they do not describe how the application should look, but how it should behave and the

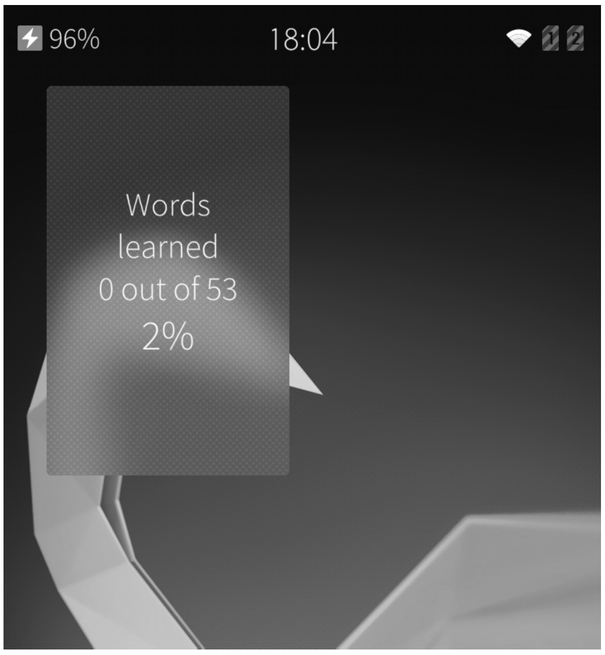


Fig. 4. Cover page of Dictionary Trainer

behaviour is equal on both platforms. These files can be described as stores, actions, and dispatcher in Flux architecture terminology. *TermInformationStore* and *TestStore* components represent stores for information about currently chosen term and current test question correspondingly and contain functions that are needed to work with this data. *ActionTypes* and

AppActions components describe all actions available within the application. Lastly, *PageOpenerScript* describes the actions of the application (mostly switches between the pages).

In some cases, in these components there might be a need to perform different actions on different platforms. For example, to manipulate the page stack, since this component is implemented separately on both platforms. In these cases, the *Qt.platform.os* property can be used to determine the current platform. For instance, *PageOpenerScript* contains a script to proceed to the given page that looks as follows:

```
if(Qt.platform.os === "linux") {
    pageStack.push(Qt.resolvedUrl(
        "../sailfish−only/views/pages/" + message.url));
} else if(Qt.platform.os === "android") {
    AppActions.stackViewPush("../pages/" + message.url);
}
```

In this piece of code, the application on the Sailfish OS platform will use a standard page stack component from Sailfish Silica framework to navigate to the given page, while on Android the *stackViewPush()* action will be emitted that is handled by *StackView* component, defined only for this platform.

The resource file is shared between both platforms and contains all files needed for both platforms: icons, xml file with data about terms, localization files. The localization files are also shared between the platforms, since the Qt localization mechanism is working as it is on both platforms.

The application sources also contain a manifest file, a file required for gradle to build the application, and a keystore. All of them are needed to build the application for Android and were generated by the Qt SDK.

To build the application for Sailfish OS the following files are required: *harbor-dictionary-trainer.yaml*, *harbor-dictionary-trainer.spec*, and *harbor-dictionary-trainer.desktop*. They are also included into the project.

VI. Conclusion

The paper describes the approach for developing cross-platform mobile applications for Android and Sailfish OS platforms that follows Flux architecture and uses Qt Framework and QML. The approach allows to quickly and easily develop applications that look and behave natively on each platform. Additionally, the paper provides an example of the application that was successfully developed for both platforms with the use of the described approach.

Alternatively, Sailfish OS allows launching native Android applications without making any changes to the code. However, since these applications were developed only for the Android platform, they look and behave as native Android applications that differs from native Sailfish OS applications. This fact leads to a poor user experience when launching applications in this way.

Unlike launching native Android applications on Sailfish OS, using the Qt framework and approach described in the paper allows to develop an application that will keep the native look and feel on all platforms. Moreover, the choice of the framework allows to easily compile the created application, not only for two selected mobile platforms, but for some others as well, such as iOS. Therefore, the approach may be extended to support more platforms.

Acknowledgment

The article was published with financial support by the project No. 549 of P.G. Demidov Yaroslavl State University within State Assignment for Research.

References

[1] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with md 2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 526–533.

[2] L. Corral, A. Janes, and T. Remencius, "Potential advantages and disadvantages of multiplatform development frameworks–a vision on mobile environments," *Procedia Computer Science*, vol. 10, pp. 1202–1207, 2012.

[3] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*. ACM, 2013, pp. 213–220.

[4] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," in *International Conference on Web Information Systems and Technologies*. Springer, 2012, pp. 120–138.

[5] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2013, pp. 323–328.

[6] H. P. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.

[7] T. Iulia-Maria and H. Ciocarlie, "Best practices in iPhone programming: Model-view-controller architecture—Carousel component development," in *EUROCON-International Conference on Computer as a Tool (EUROCON), 2011 IEEE*. IEEE, 2011, pp. 1–4.

[8] V. Kumar, A. Kumar, A. Sharma, and D. Singh, "Implementation of MVC (Model-View-Controller) design architecture to develop web based institutional repositories: A tool for information and knowledge sharing," *Indian Research Journal of Extension Education*, vol. 16, no. 3, pp. 1–9, 2016.

[9] J. Smith, "Patterns-wpf apps with the model-view-viewmodel design pattern," *MSDN magazine*, p. 72, 2009.

[10] R. Francese, M. Risi, G. Tortora, and G. Scanniello, "Supporting the development of multi-platform mobile applications," in *2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*. IEEE, 2013, pp. 87–90.

[11] B. Lau. (2016, March) QML application architecture guide with Flux. [Online]. Available: https://medium.com/@benlaud/qml-application-architecture-guide-with-flux-b4e970374635#.pka6a48w8

[12] L. Clark. (2015, September) A cartoon guide to Flux. [Online]. Available: https://code-cartoons.com/a-cartoon-guide-to-flux-6157355ab207#.ate2re92a

[13] A. Paul and A. Nalwaya, *React Native for iOS Development*. Springer, 2016.