Cross-Platform Mobile Application Development for Smart Services
Wheeler, D.; Olszewska, Joanna Isabelle

Published in:
IEEE Joint 22nd International Symposium on Computational Intelligence and Informatics and 8th International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics

Published: 22/11/2022

Document Version
Peer reviewed version

Link to publication on the UWS Academic Portal

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the UWS Academic Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact pure@uws.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Cross-Platform Mobile Application Development for Smart Services

D. Wheeler and J. I. Olszewska

University of the West of Scotland, United Kingdom

Abstract—Current technological advances and the growth of smart applications and services require the development of multi-platform mobile applications. Writing software to run on multiple computing platforms can involve a large amount of duplicate effort. This duplicate effort can take the form of re-implementing the business logic in a different language, re-implementing the user interface or, in many cases, re-implementing both for each platform. There are many cross-platform frameworks to help reduce or eliminate this effort, but they make compromises on performance or user experience. Hence, this paper presents a cross-platform framework whose goal is to reduce this compromise by aiming to minimise platform-specific code, but not eliminate it. Using this method will allow all platforms to have a feature-rich, native interface, with most of the code, including user interface code, contained in a cross-platform library used by all platforms. Hence, this work establishes a framework for developing cross-platform GUI applications, involving the design and development of a common library as well as translation libraries for iOS, Android, and Windows. This developed framework has been successfully applied to the implementation of a real-world application for smart lift services.

Index Terms—Web Infrastructures and Devices Mobile Web, Intelligent Applications, Smart City Applications and Services, Smart Living, Smart Lifts, Industry 4.0 and Smart factory, Connected Intelligence.

I. INTRODUCTION

Nowadays, the mobile application development is not only carried out for smartphones [1] and tablets [2], but also for a wide range of app-enabled devices [3], whatever wearable [4] or embedded ones [5], used for smart applications and smart services [6], generated by the rise of smart manufacturing [7] and smart cities [8].

Indeed, the development of Industry 4.0 and Society 5.0 applications leads to challenges such as cost-efficient customization of products and services [9], to the heterogeneity of devices [10], operating systems (OS) [11], software development platforms [12], and intelligent agents (IA) [13], as well as to the growth of advanced technologies such as machine virtual reality (VR) [14], digital twins (DT) [15], internet of things (IoT) [16], secure cloud storage [17], big data processing [18], machine learning [19], etc., which all require multi-platform-based solutions. Therefore, software engineers have to develop applications running on different types of platforms.

Native application development approaches imply the use of the respective Software Development Kits (SDKs), development environments, and programming languages native to the target mobile platform [20]. With multiple major competing platforms available on the market (e.g., iOS from Apple, Android from Google, and Windows from Microsoft), mobile application developers need to be proficient in numerous platform-specific tools and languages such as the Xcode environment and Swift/Objective-C for iOS development as well as the Android Studio environment and Java/Kotlin for Android development [21]. Since each platform has its own programming languages, architecture and application programming interfaces (API), developing native applications for multiple platforms is thus a tedious and daunting task for any app developer. Furthermore, it is rather expensive and time consuming to port native applications from one platform to another [22], because writing these applications natively involves the duplication of much of the code.

Cross-platform application development frameworks have thus been proposed in the literature [23] to avoid such code duplication. Current efforts of cross-platform development approaches [24] can be classified into (i) web-based approach (using HTML, CSS, JavaScript); (ii) hybrid approach (using JavaScript); (iii) interpreted approach (using Java/Script), and (iv) cross-compiled approach (generated) (using C#). All these cross-platform mobile development frameworks allow the reuse of code across multiple platforms [21] and allow a single code base to target multiple mobile platforms [20], leading to effort, time, and money savings. However, they vary significantly in the technologies, cost, and performance characteristics of the applications produced [20]. Common cross-platform environments [23] (e.g., Xamarin by Microsoft, React Native by Facebook, or Flutter by Google) are of great help in reducing or eliminating effort in developing multi-platform applications. Whereas these tools support the development of applications for multiple platforms in less time, they make compromises on performance or user experience.

In particular, graphical user interfaces (GUI) significantly affect the lifetime of mobile applications. The diversity of mobile technologies, platforms, and devices makes the development and testing of mobile-app GUI a challenging task [25], and thus, a great amount of research is currently dedicated to improve the efficiency of GUI code development and generation [26].

All the major platform owners provide guidelines on how an application should look and behave. Users of these platforms have come to learn and understand the idioms that these guidelines lay out, and it can be detrimental to veer too far from them [21]. Many of the behaviours of controls outlined in these guidelines are similar, e.g., selecting items in a list, but the underlying controls supplied by the platform look and behave in different ways. Whereas simple controls, such as buttons and images, are not problematic, more complex controls can be. In particular, complex items can include
controls like the ‘Listbox’ which is provided by Windows. At its simplest, this displays a list of selectable items to the user. A problem may appear when the developer wishes to present this same ‘Listbox’ in an iOS application, where there is no list box control. The most alike control available is then ‘Table View’ which can be made to act like the Windows ‘Listbox’, but offers more flexibility. Issues like this plague the creation of common user interfaces across multiple platforms, and this was solely comparing Windows to iOS.

All approaches to cross-platform development must address these inconsistencies in user interfaces. Older approaches often mimicked platform controls or ignored the guidelines – that is the approach taken with Java. Most modern toolsets, e.g. Xamarin and React Native, wrap instead their user-interface architecture around the platform which enables the ‘look and feel’ of a native application. From these two approaches, it would seem obvious to follow the modern ‘wrapping’ method. Our work aims to go beyond such approach to allow more flexibility in the ‘look and feel’ of the final application.

Hence, in this paper, we present a cross-platform framework (XPF) that can deliver applications for smart services running on very different platforms (whatever mobile-specific or not, whatever maintained or not) with the maximum amount of code re-use. Indeed, by embracing, but minimizing, the native element of applications, our framework aims to allow fully platform-guideline-compliant applications to be developed.

The contribution of this paper is thus the elaboration of a framework to maximise platform-agnostic code in cross-platform app development to be deployed on heterogeneous device platforms and IoT-enabled devices for smart applications and smart services such as smart lifts.

The paper is structured as follows. In Section II, we present our base framework to develop cross-platform applications, while in Section III, we report and discuss the results of the smart-service mobile app development, using the proposed base framework to develop cross-platform applications to be deployed on heterogeneous device platforms and IoT-enabled devices for smart applications and smart services such as smart lifts.

The proposed cross-platform framework development environment is described in Section II-A, while its development approach is explained in Section II-B. The development approach consists of two stages: the first step delivering the base framework (Section II-B1) and the second one enhancing the framework by adding lists and grids (Section II-B2).

A. Development Environment

To develop an application for smart services running e.g. on both Windows and iOS, two personal computers were required. The first computer was running Windows 10 and was used for development of the base framework, Windows components, and apps. The second computer ran macOS Big Sur and was used for developing the iOS components, because iOS applications need to be digitally signed by a macOS computer before installing. It is worth noting that access to the macOS computer was provided by way of NoMachine which is a remote desktop solution allowing the Windows computer to view the macOS desktop.

Furthermore, Microsoft Visual Studio 2017 Enterprise was used – this is an integrated development environment (IDE) which provides all the tools required to develop the C++ and C# components. The IDE selected for macOS was Xcode – this tightly integrates all the tools required to create iOS projects and components. Moreover, both these IDEs support Git integration for source control. On one hand, Visual Studio provides Team Explorer which can be used for all Git interactions without leaving the IDE. On the other hand, Xcode shows the status of files in the project, e.g. if it is changed, while the command line is used to perform Git actions.

B. Development Approach

To establish the base framework (Fig. 1), a large part of the work involved building the communication architecture for elements to communicate with each other and consisted in the first phase, as explained in Section II-B1, while the second phase was concerned with adding controls to the base framework and each platform’s native component, as mentioned in Section II-B2. These additional controls were selected from analysis of the final prototype application for smart services, as described in Section III. To succeed at this point, the application’s functionalities and user interface had to be decided so that the required framework elements could be identified and implemented. This phase did not have a single application as its deliverable but, rather, the tested expanded framework for use in the smart service application.

1) Base Framework: The first phase involved the creation of the initial framework. The base framework provides the methods for routing events between the cross-platform elements and their corresponding native siblings.

Events (or messages) are identified by an integer value; an integer was chosen, since it is easily transferred across languages and platforms and is fast to search for. In a similar way, event parameters are also identified by an integer value. Figure 2 exemplifies the events and parameters for a button.

Messaging: The aim of the framework is to supply an interface to native components which can be manipulated by the cross-platform code. There are two main problems to overcome to allow this manipulation to happen. The first
problem is that there is little commonality in the underlying way that operating systems implement their user interfaces. Therefore, the framework needs to be flexible enough to allow for this. The second problem is that native applications are written in multiple, different, programming languages. It would be possible to create bridging components that translate every function required between each language and the framework, but it would be an onerous task. The selected solution to solve these problems is to create two representations of the application and communicate between them by way of messages. The two representations reside in different components of the application, and these both need to be created for a functional application. When creating a cross-platform application, the developer creates and targets a virtual representation of the user interface and components which are provided by the framework. To create a native application, the developer needs to implement the same representation in native code. The framework provides methods to make creating these two views as straightforward as possible. These representations need to be able to communicate with each other and synchronise states across the component boundaries – this communication takes the form of messages. Using messaging to transfer information greatly reduces the breadth of the interface required to communicate between different components implemented in different languages. In essence, this approach means that the required bridge component only needs to implement two functions, one for sending and another for receiving messages. Another benefit of this approach is that adding functionality, such as new controls, will have no impact on these interfaces, since all information is passed through this narrow, generic API.

Routing: Messages need to know where they are being sent to. Hence, every element that expects to send and receive messages must have a unique address. The approach taken here is to have a hierarchy of elements; each element having a unique identifier within its namespace, as shown in Fig. 3. The shown example utilises unique identifiers throughout, e.g. sub-elements start with their parent’s identifier. This hierarchy of identifiers searches the search for a specific element, since each element has a unique path. This path can be constructed from any element by walking back through the element’s parents inserting the identifiers into an ordered list until the root element is found, as illustrated in Fig. 4. These paths are key to how messages are routed between the cross-platform and native representation of the application.

Anatomy of a Message: There are three components contained within a message; an example of these can be seen in Fig. 5. The first two parts are straightforward, since the path has been discussed above, and the message type is the kind of message being sent, e.g. a button press. However, when different message types are examined, the message parameters could be significantly diverse. To provide a generic solution to this, the parameters take the form of a dictionary, and in this way, there is no limit to the number of parameters and types that can be sent.

Message Handling: All elements within the framework that can process messages are referred to as ‘event controllers’. When an element, such as a button, is added to the framework, the messages that it can handle are registered. The new element can then provide a default handler for none, some, or all these messages, depending on whether this would make sense. An example of this would be with a button, it makes sense to implement handlers for getting and setting the button text, but it would be up to the application developer to decide what action to perform when the button is pressed. There are scenarios where it might be desirable to hook or override the default behaviour of a message and the framework uses polymorphism to implement this feature with message overriding.

Cross Language Considerations: Cross-platform applications using the proposed framework are written in C++. If the native application is written in a different language, messages need to be translated. This translation is performed by a bridging component, which is a relatively low-level component that can convert parameter types from C++ to the native language and vice versa. The process of translating a message consists of converting and packing the parameters into the destination format before passing this to the destination’s message handler.

Classes: The base framework is responsible for the cross-platform representation and message routing of an application. In particular, executing a framework application on iOS requires the iOS Native Services component to function. This component creates a routable representation of the cross-platform application within the native element. When creating native elements, they must be connected to the native services by means of a method call. The majority of the iOS Native
Applications (i.e. the developers) to build applications is the priority.

Minimising the effort required by users of the framework (i.e. the focus on maximising code re-use across platforms while satisfying all the business logic) was not achieved as planned. The second phase, NativeObjects contains the classes for event parameters required by the .NET application, NativeServices, and CLIBridge. This design may seem quite complex, but it was found that this created a circular dependency between NativeServices and CLIBridge, which had to be broken by extracting the common requirements into a separate component: NativeObjects. In this first phase, NativeObjects contains the classes for event parameters that are required by the .NET application, NativeServices, and CLIBridge. This design may seem quite complex, but the focus on maximising code re-use across platforms while minimising the effort required by users of the framework (i.e. the developers) to build applications is the priority.

**Message Routing:** When the text is changed, the cross-platform application must be informed of this event, since it needs to update its internal state. There may also be a need to perform some processing on this text, such as validation. If that is the case, the application would register a handler for the message.

**Component Linking (static/dynamic/compiled in):** As this project involves multiple languages, there are several logical units that are required to enable an application to be created. Objective C, due to native C++ support, allows all the logical units to be compiled into a single executable, but that is not possible with the Windows implementation. For a Windows application, there are six required compilation units, as depicted in Fig. 6. The initial design envisioned there would only be five separate units. However, it was found that this created a circular dependency between NativeServices and CLIBridge, which had to be broken by extracting the common requirements into a separate component: NativeObjects. In this first phase, NativeObjects contains the classes for event parameters that are required by the .NET application, NativeServices, and CLIBridge. This design may seem quite complex, but the focus on maximising code re-use across platforms while minimising the effort required by users of the framework (i.e. the developers) to build applications is the priority.

**Swizzling:** The way that messages are routed in UIKit on iOS is different to how it is done in .NET on Windows. On Windows, there is a public method to add additional handlers to an object, whereas on iOS, there is not. During the implementation, there were two methods identified to circumvent this. The first method was to require the application developer to derive a new class from UIViewController for each window in the application and then add the handler. This method was possible but went against the ethos of the framework to minimise development effort. The second method identified is called swizzling. This takes advantage of a low-level feature in Objective C and as such was not decided on lightly. The premise of this process is to replace the Objective C method in a class with a different implementation at run-time. This process must be performed carefully, since any error may cause an application crash. It is important to ensure that if there is a current handler, this should still be called within the injected method. Figure 7 shows the ‘before’ and ‘after’ structure of a UIViewController that has been swizzled. With this method all the routing of Windows messages can be implemented within the framework, and no additional workload is placed on client developers.

2) **Framework Build-out:** While the first phase was focused on proving that messages can be sent backwards and forwards through the different layers of the framework, the second phase is concerned with the addition of controls that would be required by the prototype lift control application delivered at the end of the project. For that, the lift service requirements have to be analysed, and the application, including the user interface, has to be designed, following a methodological approach such as D7-R4 [27]. Any identified control has to be implemented in the framework before the final prototype could be built. The amount of effort required to design and implement these controls varies considerably. Some of the controls, list, and grid need to contain child controls adding to their complexity. Functionality for the label and multi-line text controls, list, and grid need to contain child controls adding to their complexity. Functionality for the label and multi-line text box can be derived from the already-implemented text box.

It is worth noting that during the development of the base framework, all the C++ libraries had to be converted to dynamic link libraries (DLLs) which create a looser dependency, since they are resolved at run-time rather than at compile-time. Besides, it was found that some of the Windows Forms controls were not flexible enough to easily support the required design, and to work around the ‘Listbox’ limitations, the Windows user interface was switched over to Windows Presentation Foundation (WPF).
III. CASE STUDY: SMART LIFT SERVICES

To prove that the elaborated base framework could be used to develop cross-platform application for smart services, a prototype application for smart lifts has been designed, implemented, and tested.

The main functionalities of smart lift services include use cases such as Control Lift and Check Lift Status. The corresponding wireframes are displayed in Fig. 8.

On one hand, the Lift Control page (on Fig. 8(a)) is designed such as its functions mimic a lift control panel and thus, has three control areas, as follows:

- **Lift Buttons**: These are standard buttons that perform the stop, open doors, and call functions on the lift.
- **Floor Buttons**: This is a grid of buttons, with a button for each floor. As the number of floors is unknown, buttons are dynamically added for the lift it is connected to.
- **Menu**: Opening the menu is done by means of a button on the bottom of the page.

On the other hand, the Lift Status page (on Fig. 8(b)) is populated from the physical lift and the sensors within its mechanism. Again, the central control is the list control that displays related information within segments. All information uses label controls to inform the lift engineer/agent.

In the remainder of this section, we focus on the description and illustration of the lift controlling element of the application. The corresponding classes are shown in Fig. 9.

For the purpose of testing, a dummy lift application, consisting in a server holding the state of several lifts, has been also implemented. Each lift’s state is made of static and dynamic items. The static items include the lift name and the number of floors, while the dynamic items are elements such as the current floor, the state of the doors, and other buttons. Such server was implemented in C# and provided a REST interface to be accessible through a TCP port. Thence, clients would send requests to this port, and the server would respond with either the data requested or update the lift’s state. Using TCP to access the lift server meant that to avoid writing bespoke code for communication, a cross-platform library needed to be found. Ideally, when selecting a library for a cross-platform application, the library will support all the platforms required. It would be possible to used different libraries, but this would introduce additional native code. So, the ‘libCurl’ library was adopted, since it is available for many platforms including Windows and iOS targeted by this application. Indeed, used by many projects, ‘libCurl’ is a mature library that performs communications over TCP/IP and also provides a C++ interface. The process of linking the cross-platform framework with this library follows an identical process as for other applications. Hence, as the dummy lift server is used to provide the data for the lift control applications and as this uses TCP/IP to pass data, it is possible for both applications to run at the same time accessing the same current lift.

Figure 10 shows the applications running both on Windows and iOS. There are three images of each to show how the application reacts when connected to a different lift. The dynamic grid control is the most obvious change, as the number of floors changes; the application ordering these floor buttons so that they follow the pattern that would be expected on a lift panel.

Further testing consisted of a blend of automated and manual unit testing of components [28]. After the initial testing, the automated element forms the regression testing for any change made to the code base. Visual Studio automatically runs these tests after a successful build, enabling the developer to immediately see any problem they may have injected.
Besides, from a user-experience point of view, when using our proposed base framework, the development of the smart service app has been eased by using the ‘AppBuilder’ class; this being the only class that the C++ developer is required to interact with. For the native application elements, much of the plumbing of events is automatically handled by the framework without developer interaction. Initial measurements of the framework performance are based on a count of the lines of code (LOC) required for a native application [29]; this metric displaying the amount of effort a developer could save by using the framework. When both Xcode and Visual Studio create source items (e.g., projects and classes), there can be a large amount of ‘boiler plate’ code inserted. This code is necessary but involves no effort from the developer. To provide a fair comparison, all automatically generated code was discarded from the LOC count. Table I shows the results of the LOC analysis. From this table, it can be seen that nearly 70% of the application code is shared between both platforms.

These results from the measurement of code re-use are very encouraging. Indeed, on a small application, having a near 70% of code re-use across platforms is a big gain. This code re-use was also found beneficial in the time saved creating native applications, where the iOS version of the Lift Control prototype was completed within half an hour. Productivity gains like these are a huge asset for smart service development and customization. Furthermore, the proposed cross-platform framework reduced the amount of platform-specific code without impacting the ‘look and feel’ of the native application.

Hence, the smart service application shows some of the possibilities offered by the elaborated framework and how it has been created allowing a maximum amount of flexibility. This flexibility is available as whenever possible the framework has avoided making assumptions about the type of objects it is dealing with. The messaging and parameters are particularly amenable. Indeed, new messages can easily be created with parameters of any type and, if these are application specific, there is no need to alter the underlying framework. Implementing the framework as flexibly as possible makes the current implementation a solid foundation for further work.

With this solid foundation, building on our framework with new controls and features is possible. Indeed, this low coupling of components was enabled by the use of good object-oriented design and implementation at the outset of the project. Such low coupling means that it is relatively easy to carry out further work without impacting on the current framework.

| TABLE I  |
| LIFT CONTROL APPLICATION’S LINES OF CODE (EXCLUDING BOILER PLATE AND UI LAYOUT). |
| Lines of code | %  |
| Cross-platform code | 323 | 69.16% |
| iOs code | 85 | 18.20% |
| Windows code | 59 | 12.63% |
| Total | 467 | 100% |

IV. CONCLUSIONS

As the development of smart cities and smart manufacturing technologies tends to require more and more multi-platform applications, we present in this paper a cross-platform application development framework for smart services. In the first stage, our work successfully delivered the base framework, while the second stage of the framework elaboration enhanced the base framework by adding lists and grids. The overall cross-platform framework was then used for smart-service application development in context of lift industry. It proved it was flexible enough for complex controls and led to productivity gains by speeding up the development of cross-platform mobile applications.

ACKNOWLEDGMENT

The authors would like to thank Innovate UK and Consult Lift Services Ltd for the support of this work.

REFERENCES