A comparison of native and multiplatform development of mobile applications following the MVVM pattern

Bc. Veronika Dzúriková

Master's thesis
2022

Tomas Bata University in Zlín
Faculty of Applied Informatics

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Veronika Dzúriková**
Osobní číslo:	**A19349**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**Kombinovaná**
Téma práce:	**Porovnání nativního a multiplatformního vývoje mobilních aplikací v architektuře MVVM**
Téma práce anglicky:	**Comparison of Native and Multiplatform Development of Mobile Applications Following MVVM Pattern**

Zásady pro vypracování

1. Popište současný stav technologií mobilního vývoje.
2. Zaměřte se na framework Xamarin.Forms, MVVM-Cross a nativní vývoj pro platformy Android a iOS.
3. Porovnejte implementaci architektury MVVM v různých frameworcích.
4. Navrhněte a popište ukázkovou aplikaci.
5. Vytvořte vzorová řešení demonstrující klíčové prvky navržené aplikace na různých platformách.
6. Zhodnoťte výhody a nevýhody jednotlivých řešení.

Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

1. MARTIN, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ: Prentice Hall PTR, 2008. ISBN 978-0-13-235088-4.
2. HERMES, Dan. Building Xamarin.Forms mobile apps using XAML: Mobile cross-platform XAML and Xamarin.Forms fundamentals. New York, NY: Springer Science Business Media, 2019. ISBN 978-148-4240-298.
3. VERSLUIS, Gerald. Xamarin.Forms Essentials: First Steps Toward Cross-Platform Mobile Apps. New York, NY: Springer Science Business Media, 2017. ISBN 978-148-4232-392.
4. Xamarin.Forms Documentation | Microsoft. [online]. Dostupné z: https://docs.microsoft.com/en-us/xamarin/xamarin-forms
5. MvvmCross Documentation | MvvmCross. [online]. Dostupné z: https://www.mvvmcross.com
6. Android Jetpack | Android Developers. [online]. Dostupné z: https://developer.android.com/jetpack
7. Human Interface Guidelines | Apple Inc. [online]. Dostupné z: https://developer.apple.com/design/human-interface-guidelines
8. Apple Developer Documentation | Apple Inc. [online]. Dostupné z: https://developer.apple.com/documentation

Vedoucí diplomové práce:	**Ing. Erik Král, Ph.D.** Ústav počítačových a komunikačních systémů
Datum zadání diplomové práce:	**3. prosince 2021**
Termín odevzdání diplomové práce:	**23. května 2022**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.
①

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

ABSTRAKT

Práce se zaměřuje na porovnání implementace návrhového vzoru Model-View-View-Model (MVVM) ve vývoji mobilních aplikací pro nativní vývoj pro platformy iOS a Android, a pro multiplatformní vývoj v technologii Xamarin.Forms. Motivací pro tento výzkum byl fakt, že v reálném pracovním prostředí se každý tým nebo firma potýká s problémem jak správně tento návrhový vzor implementovat a jaký způsob (nativní nebo multiplatformní) vybrat pro vývoj každé nové aplikace. Rešerše proběhla tak, že pro každou platformu byl proveden výzkum platformních zásad, dokumentací a možností, jak tento návrhový vzor implementovat, a pak byl vybrán nejčistší a nejvíce doporučený způsob pro vývoj ukázkové aplikace. Pro tuto aplikaci byla zvolena množina klíčových komponent a dat, která tvoří jejich obsah, a byly implementovány tři aplikace – nativní iOS aplikace v programovacím jazyku Swift, nativní Android aplikace v jazyku Kotlin a multiplatformní aplikace pro obě platformy v jazyku C# v technologii Xamarin.Forms. Vývoj těchto tří aplikací byl následně detailně popsán s hlavním zaměřením na popis implementace návrhového vzoru (MVVM). V závěru jsou všechny tyto způsoby implementace vyhodnoceny a porovnány.

Klíčová slova: MVVM, iOS, Swift, SwiftUI, Android, Kotlin, Xamarin, Xamarin.Forms, MvvmCross

ABSTRACT

Thesis focuses on comparison how the Model-View-ViewModel (MVVM) architecture pattern is implemented for mobile applications in native iOS and Android development, and in multiplatform development using Xamarin.Forms. Motivation for this research was a fact that in real working environment, every team or company deals with this problem how to correctly implement this architecture or which approach (native or multiplatform) to choose for each new application. The research was conducted in such way that guidelines, documentation and possibilities how this architecture can be implemented were researched for each platform and then the most clean and recommended way was chosen for development of an example application. For this example application were set key components and example data to fill its content, and then three applications were developed – native iOS in Swift programming language, native Android in Kotlin, and multiplatform application for both iOS and Android in C# using Xamarin.Forms. Development of these three applications was described in detail with focus on MVVM architecture implementation. In the end all these implementation approaches are evaluated and compared.

Keywords: MVVM, iOS, Swift, SwiftUI, Android, Kotlin, Xamarin, Xamarin.Forms, MvvmCross

I would like to thank my thesis supervisor Ing. Erik Král, Ph.D. for allowing this research and for supervision of this thesis.

TABLE OF CONTENTS

INTRODUCTION

This thesis is about comparison how the Model-View-ViewModel (MVVM) architecture is implemented across currently available mobile platforms – iOS and Android. In theoretical part are first described methods how mobile applications can be implemented, then the description of the MVVM architecture and MvvmCross framework. In analytical part is performed research how this architecture may be implemented in native iOS, native Android and in Xamarin.Forms for multiplatform development. Practical part includes description of the example application and its components and behaviour which is then implemented for native iOS, native Android and in Xamarin.Forms in a way which demonstrates the MVVM implementation. Implementation of all these three applications is then described in detail with examples of important source code samples. In the end is explained which approaches are suitable for different use cases, summarized main differences between MVVM implementation across these three approaches, and all three (respectively four) example applications are compared.

I. THEORETICAL PART

1 Current methods in mobile applications development

As of January 2022, Android with 70.05 % of the worldwide market share and iOS with 29.21 % are the leading worldwide Operating Systems [1]. Manufacturers such as Nokia with their not very triumphant Windows Phone and BlackBerry quickly switched their environment to Android as well. So this thesis will further deal with these two mobile OSs, Android and iOS.

Currently there are three approaches of how to develop mobile applications. The most standard and oldest way is making native applications separately for each platform. To cut time costs, lower human resources and eliminate code duplicity multi-platform development increased its popularity. This approach still had not been enough for some companies financial policies so they started to develop applications, which in reality are just web sites wrapped into native application packages or single-use web browser applications.

1.1 Native single-platform approach

This is the oldest and the most straightforward way of mobile applications development. Separate native application for each platform using standard Software Development Kit (SDK) provided by the corresponding platform maintainer. Native applications generally have high performance and better User Experience (UX) as the User Interface (UI) elements are tailor-made especially for each platform and there is direct access to hardware interfaces through provided Application Programming Interface (API) capabilities. Developers have immediate access to new features and do not have to wait for secondary companies to prepare wrapping or plugins as is common in multi-platform development. It is also easier to comply with the corresponding app store guidelines.

1.1.1 iOS

iOS is a mobile operating system developed in-house by Apple and used exclusively in the company's iPhone mobile devices. Apple tablets, iPads, use iPadOS which is derived from iOS version 12. It is closed-source and written in various programming languages: C, C++, Objective-C, Swift and Assembly. The kernel holds the name Darwin and is derived from Berkeley Software Distribution (BSD) [1)], later NeXTSTEP

[1)] Discontinued OS based on Research Unix, developed and distributed by Computer Systems Research Group (CSRG) at the University of California in Berkeley

[2] and other projects [2]. Besides already mentioned iPadOS, Apple furthermore uses other marketing names for their OSs for different devices – tvOS for Apple TV and WatchOS for their wearable smartwatches. iOS and iPadOS both use same libraries provided by Apple in their SDK for developing applications, although tvOS and watchOS use only some of the components and their UI is developed in a slightly different way due to their nature.

Objective-C

Historically Objective-C programming language was used for developing iOS applications. It is object-oriented programming language based on the C programming language. It uses message passing similar the the one in the Smalltalk [3] programming language. The main difference between Simula-style model (used for example in C++) is how the code referenced by a method of a message name is executed. Simula-style method names are pointed to a section of code by the compiler, whereas targets (receivers) of the messages (in Objective-C and Smalltalk) are resolved at runtime, and the receiving object itself resolves the message [3]. Unlike in more traditional languages, a function or a method is represented by the selector in Objective-C, which is a unique identifier for each message name. So we can say that a selector is the message's signature. This is important because most of the OS and OS libraries still use Objective-C selectors as their interface which developer needs to call from a more current, modern, language which is used today.

Swift

Currently used language is Swift. It was first released in 2014 and quickly became the main choice for developers for the Apple platform development. It's current stable version 5.6 released in March 2022. Unlike Objective-C, this programming is open-source but mostly developed by Apple. It is multi-paradigm programming language, supporting object-oriented approach with heavy focus on functional programming. It is strongly typed, uses automatic reference counting (whereas Kotlin uses Garbage collection) for its memory management and in later versions (5.5 and up) adds support for concurrency and fully asynchronous code by introducing its own version of the actor model [4]. It is built with the Low Level Virtual Machine (LLVM) compiler framework as well, uses Objective-C runtime library which allows iOS and other systems to run

[2] Discontinued object-oriented, multitasking OS based on the Mach kernel and BSD

[3] Object-oriented, dynamically typed reflective programming language influenced by Simula

C, Objective-C, C++ and Swift code within one application.

Application development

Both Objective-C and Swift are supported by Xcode IDE which is main IDE for building applications for all Apple OSs. It is complete environment, providing editor, project manager (along with certificate and other management tools), Git client, profiler, debugger and compiler support for the whole iOS platform [5]. Secondary option is AppCode from the Czech company JetBrains but developers still need to have Xcode installed on their macOS computers to build iOS applications.

Main system libraries provided by Apple are:

- Foundation

  This is the core SDK and provides basic data types and functions for working with them, like Date, Calendar or APIQuery. It provides a base layer of functionality for applications and frameworks, including data storage and persistence, text processing, date and time calculations, sorting and filtering, and networking [6].

- UIKit

  This is the most used UI library which contains not only basic UI elements, but also more general objects and related functions like UIImage for images or UIColors for colours.

- CGGraphics

  This is lower level Graphics Processing Unit (GPU) accelerated 2D graphics library for drawing everything on the device's screen, for example geometry, images, blur and other effects. It is also used by the UIKit as well and also provides its own version of more optimized objects for representing colours – CGColor.

Application composition and UI implementation

Each application consists of at least the `UIApplicationDelegate` and one Scene. `UIApplicationDelegate` is very similar to a `main.hpp` class in C++. it is an entry point of the application and along with other implementable protocols serves as a main interface between the application and the OS itself. It receives application's launch parameters and also the application lifecycle events (like events about entering background and foreground).

`UIViewController` is a controller class, instantiates `UIViews`, handles their lifecycle and sometimes implements their delegates. `UIView` is any part of the UI, actually the whole screen in an `UIView`, but also any smallest element on the screen is also `UIView` [6]. There are four states that the `UIViewController` can acquire as show in the figure 1.1. Historically iOS applications used to consist of at least one `UIView` and one `UIViewController`, though it is no longer true since iOS 13 when SwiftUI was introduced and UI implementation style transitioned from Window usage to Scene usage. Windows used to be an object representing the application drawing space in which the `UIViewControllers` along with their `UIViews` were hosted. The caveat is that every application could have only one Window so Apple transitioned to Scene usage when they introduced support for multiple application windows and decided to call them Scenes. That means that the application can now only contain one Scene with one SwiftUI view, and without any `UIViewControllers`. However, when developing applications with more than one screen developers usually choose some sort of architecture depending on their preferred frameworks. This also helps to separate objects' responsibilities, therefore structure of the applications is usually more advanced and less straightforward. So depending on the chosen technology developers must choose particular architecture because these technologies and frameworks usually require some sort of objects to carry responsibilities defined by Apple, so the applicable architecture patterns were designed around these requirements.

UI declaration

There are four ways how to create UI:

- xib files

  They are actually XML-based files for storage of UI declaration made in Xcode Interface Builder. This is the oldest way of defining UI for iOS applications, and it is possible to use only one xib file per view – xib file represents single isolated screen.

- Storyboards

  These were introduced with iOS 5 and they facilitate development because they allow prototyping and designing multiple view controller views within one file,

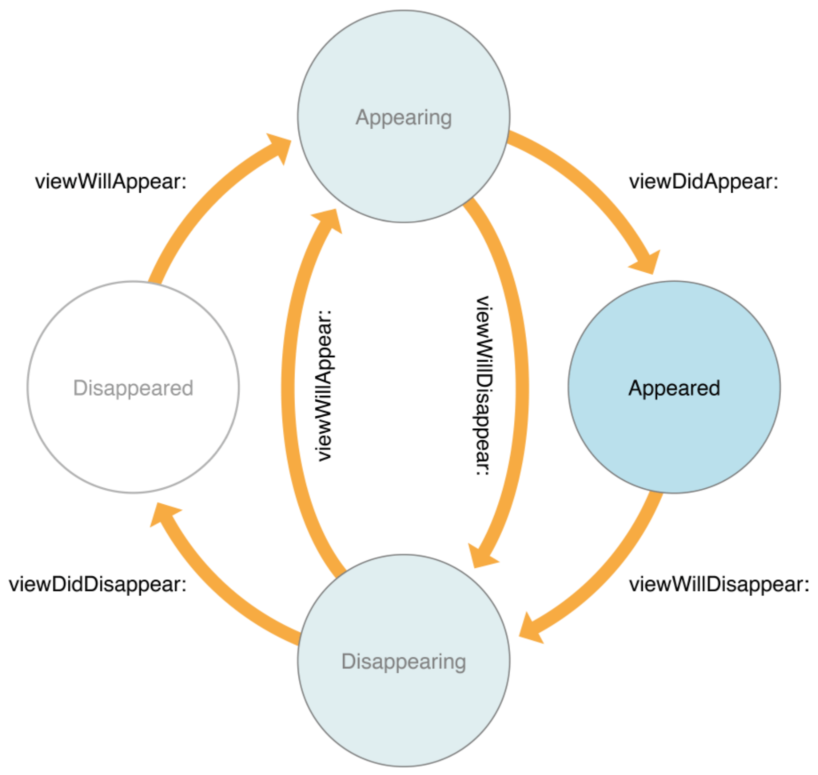


Fig. 1.1 `UIViewController` lifecycle
[7]

and also let developers create transitions between view controllers [8] which handle the navigation. Declaration of UI is similar like with xib files, using UIKit elements.

- Programmatically using UIKit

  Defining the UI in the code provides absolute freedom, developers can completely mix up objects' responsibilities as they wish. Any object can implement any protocol, just needs to set proper references to the required properties. As UI elements are instantiated purely in the code, it is done so in the `UIViewController` delegates. These delegates also instantiate the whole Views and their lifecycle. AutoLayout can be used, usually with conjunction of Domain-specific Language (DSL) and wrappers to set it up more easily.

- Programmatically using SwiftUI

  This is the most modern way of defining the UI, from the code, possible since iOS 13. It is completely different from the UIKit. There is declarative approach instead of imperative – developers define how should something look instead of setting up properties to make something look certain way. Views need to conform to predefined structure and there is less architectural freedom, yet still more efficient than xib or Storyboards approach. Even though this is done programmatically from the code, it supports real-time live previews in Xcode IDE using `PreviewProviders` [6].

1.1.2 Android

Android is mobile OS based on Linux kernel and distributed as a free and open-source project under Apache license, however most Android devices ship with pre-installed proprietary software as well. The first device shipped with Android OS was HTC Dream in 2008 [9]. It is being developed by Open Handset Alliance (OHA) consortium though commercially sponsored and led by Google. With the annual trend, Google releases its own smart phone called Pixel (previously Nexus) with pure Android OS, and other phone manufacturers have to scale this system for their phones. Then they often override its UI with their own, for example Samsung with its One UI or Xiaomi with MIUI, and pre-install their proprietary vendors' applications.

Kotlin

Developing native Android applications has been possible in Java and C++ programming languages but in recent years Java has been gradually replaced by Kotlin. It is actually a cross-platform supported language (same as Java) so there will be more information about this language in section about multiplatform development later in this thesis.

Application composition

Each android application consist of app components. There are four types of components [10]:

- Activities

  An activity is the entry point for interacting with the user [10]. It is a basic building block of an Android application and all the action happens there. Every activity has to be declared in the `manifest.xml` file. Each activity can host one or more fragments.

  Fragment represents a reusable portion of the application UI. It is strongly dependent on an activity and can not exist on its own, has to be hosted by an activity.

It manages its own layout, has its own lifecycle separate from an activity and can handle its own input events [10]. During the development there must be sure full understanding of connection between activity and fragment lifecycle (as seen in figure 1.2) to ensure the proper functionality of the application.

- Services

  A service handles performance of long-running operations in the background while not blocking the UI and supplies functionality for the others applications to use. It might continue running even after the application is not currently open and/or the user is even interacting with different application.

- Broadcast receivers

  Broadcast receivers are used to respond to system-wide and application events. Basically they send or receive messages from the system, application itself or other applications. These messages can be events or intents. All registered receivers are notified when an event occurs by the Android runtime. An Android application can then receive broadcasts in two ways, through static receivers (declared in the `manifest.xml` file) and dynamic receivers (context-registered) [10].

- Content providers

  Content providers are mostly used for supplying data from one application to another by request. They manage access to a central repository of data. Data stored by certain application or other applications stored in a database, files, or over a network. They encapsulate these data and offer granular control over the permissions for accessing them. This ensures full security over them. Then obviously, they support all four basic CRUD operations – create, read, update and delete the data.

UI implementation

Since the first Android released in 2008 applications UI has always been defined in XML files in a flat structure. This supports creating different layouts for different screen orientations or screen sizes. Android provides basic elements such as views, layouts, textViews, buttons, et cetera, together with styling and theming definitions for developers to define UI of the applications. However, along with the release of Kotlin language for Android and new Jetpack libraries, Jetpack Compose was released to replace XML structure.

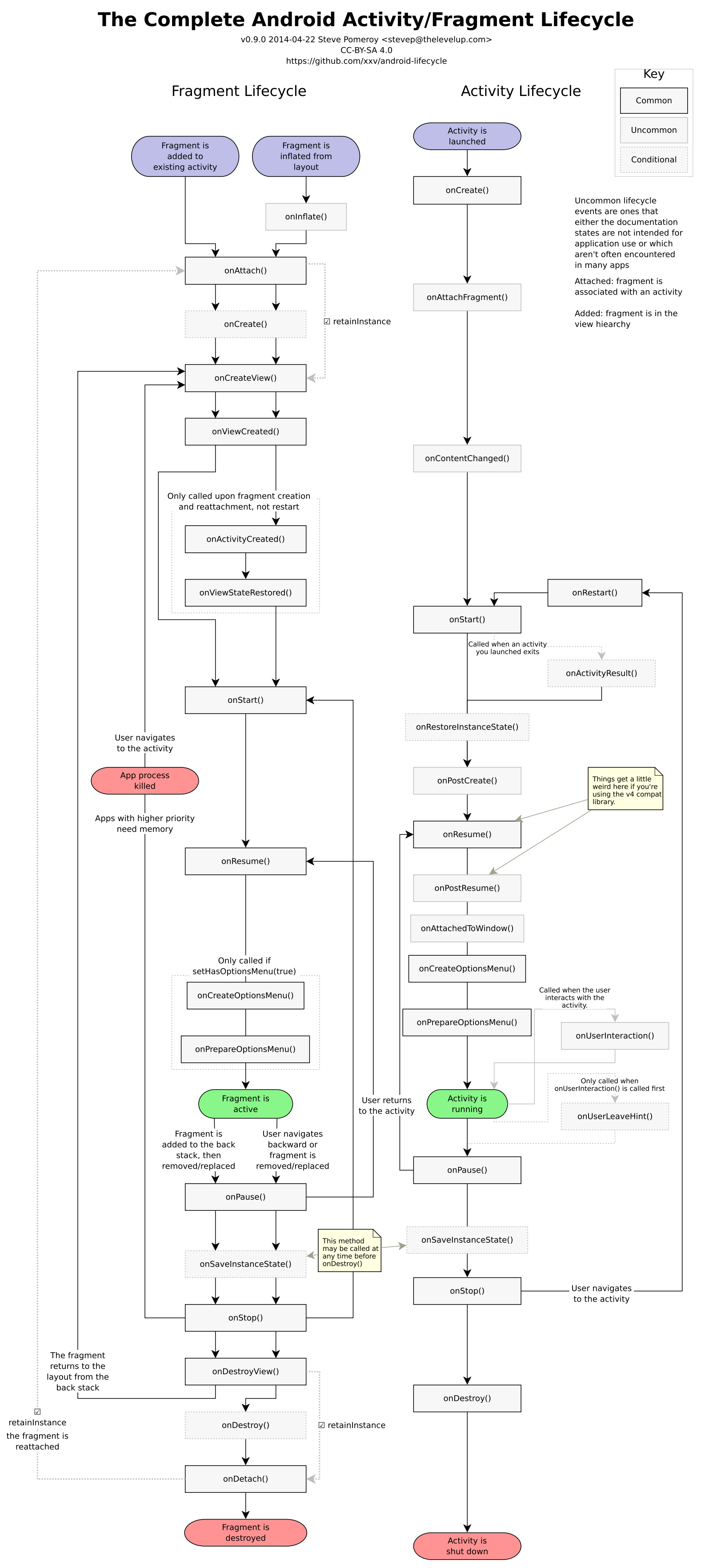

The Complete Android Activity/Fragment Lifecycle
v0.9.0 2014-04-22 Steve Pomeroy <stevep@thelevelup.com>
CC-BY-SA 4.0
https://github.com/xxv/android-lifecycle
Fragment Lifecycle
Activity Lifecycle
Key
Common
Uncommon
Conditional
Uncommon lifecycle events are ones that either the documentation states are not intended for application use or which aren't often encountered in many apps
Attached: fragment is associated with an activity
Added: fragment is in the view hiearchy
Fragment is added to existing activity
Fragment is inflated from layout
onInflate()
onAttach()
onCreate()
☑ retainInstance
onCreateView()
onViewCreated()
Only called upon fragment creation and reattachment, not restart
onActivityCreated()
onViewStateRestored()
onStart()
User navigates to the activity
App process killed
Apps with higher priority need memory
onResume()
Only called if setHasOptionsMenu(true)
onCreateOptionsMenu()
onPrepareOptionsMenu()
Fragment is active
Fragment is added to the back stack, then removed/replaced
User navigates backward or fragment is removed/replaced
onPause()
onSaveInstanceState()
This method may be called at any time before onDestroy()
onStop()
The fragment returns to the layout from the back stack
onDestroyView()
☑ retainInstance the fragment is reattached
onDestroy()
☑ retainInstance
onDetach()
Fragment is destroyed
Activity is launched
onCreate()
onAttachFragment()
onContentChanged()
onRestart()
onStart()
Called when an activity you launched exits
onActivityResult()
onRestoreInstanceState()
onPostCreate()
Things get a little weird here if you're using the v4 compat library.
onResume()
onPostResume()
onAttachedToWindow()
onCreateOptionsMenu()
Called when the user interacts with the activity.
onPrepareOptionsMenu()
onUserInteraction()
User returns to the activity
Activity is running
Only called when onUserInteraction() is called first
onUserLeaveHint()
onPause()
onSaveInstanceState()
onStop()
User navigates to the activity
onDestroy()
Activity is shut down

Fig. 1.2 Activity vs. Fragment lifecycle [11]

Jetpack Compose is a new toolkit for building UI. In Jetpack Compose layouts are defined in a completely different way in Kotlin classes. They are built around composable functions. These functions let developers define the UI programmatically describing how it should look and providing data dependencies, rather than focusing on the process of the UI construction [12].

1.2 Native multiplatform approach

This approach takes advance of fast performance of native applications but at the same time cuts costs by code reusability. These applications share core codebase between both Android and iOS (but can be even more) platforms and only platform-specific operations are implemented for each of them separately. This unintentionally follows SOLID [4)] architecture as it forces to use interfaces and keep the implementation independent and substitutable [13]. UI can be defined separately for the best visual experience, however some frameworks (for example Xamarin) provide tools to define single UI for both platforms. Though this way is not very popular between the companies with strong focus on brand design and end-users because results of this way are usually not very pleasant for a designer eye. But for basic use-cases such as internal applications for company employees, this is usually enough and cost-effective way.

1.2.1 Xamarin

Xamarin is free and open-source .NET framework currently under the hood of Microsoft corporation for developing Android, iOS, MacOS and Universal Windows Platform (UWP) native applications. Xamarin enables developers to share an average of 80-90 % of the application across these platforms [14]. Source code is written in C# programming language using Visual Studio Integrated Development Environment (IDE). Developing Xamarin applications using Visual Studio IDE is possible on any OS – Windows, Linux and MacOS with limitation that iOS and MacOS applications can be build and deployed only using Mac computers (directly or via network connection). Usage of this framework has no negative effect on performance because Xamarin applications are build into native applications for each platform, but application package is larger because of .NET libraries. What is interesting is that application is not compiled into Dynamic-link Library (DLL) as is standard with Microsoft projects, but the code is compiled into each platform package. All platforms APIs, services and methods are

[4)]Mnemonic acronym of five software design principles of object-oriented programming stated by Robert C. Martin, also known as Uncle Bob

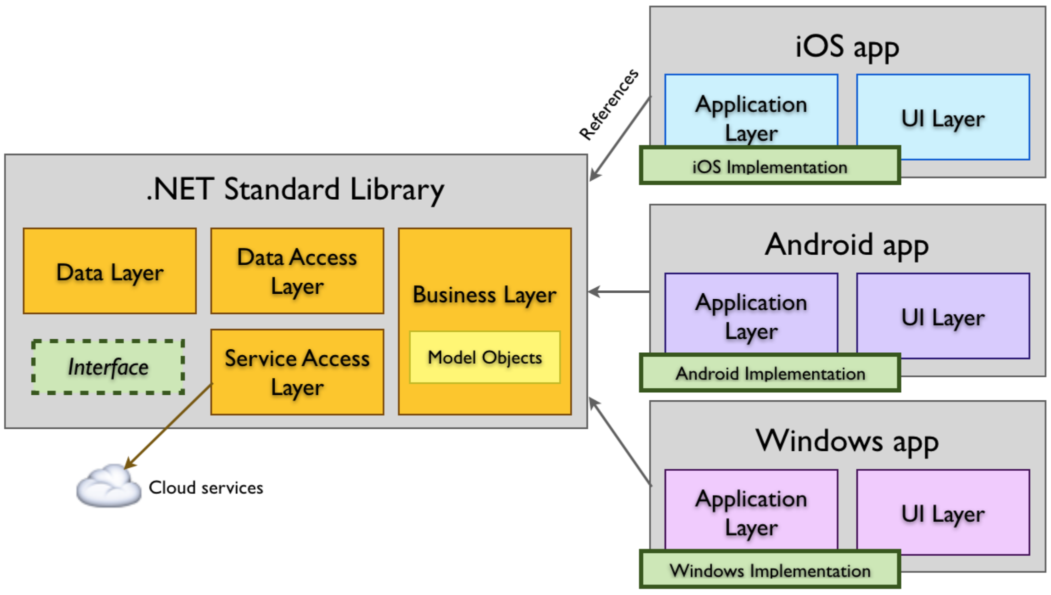


Fig. 1.3 Xamarin application architecture
[14]

available, just called from wrapped functions written in C# programming language. There is usually no any delay with updates reflecting the platform maintainers changes as Microsoft is responding to these changes immediately and updates use to be available the same day as native platform releases. Some problems may happen when programmers want to use third party library written for native development, then they have to rewrite it, find different one supposed for Xamarin, or use Xamarin Bindings library which allows to connect third party library into C#.

Application composition

.NET Standard library version 2.0 is the preferred option for sharing source code between the platforms. It is uniform API for all .NET platforms including Xamarin and .NET Core. Xamarin application consists of core project (.NET Standard library) which includes data layer (all model classes), data access layer, service access layer, business model layer and interfaces, and each platforms projects which consists of UI layer, application layer and platform-specific implementation of above mentioned interfaces. Examples of these platform specific implementations may be authentication, permissions, I/O access and file storage or databases.

UI implementation

There are three approaches of how to handle UI implementation:

- Native

  Project for each platform has its own platform's native implementation which means double (or triple) amount of work. For iOS it is using ViewController classes and UI can be designed in Storyboards or xib files. Android has Fragment classes and design is defined in XML files. This approach ensures the most authentic native UI for each platform using native libraries and tools, and best UX.

- Xamarin.Forms

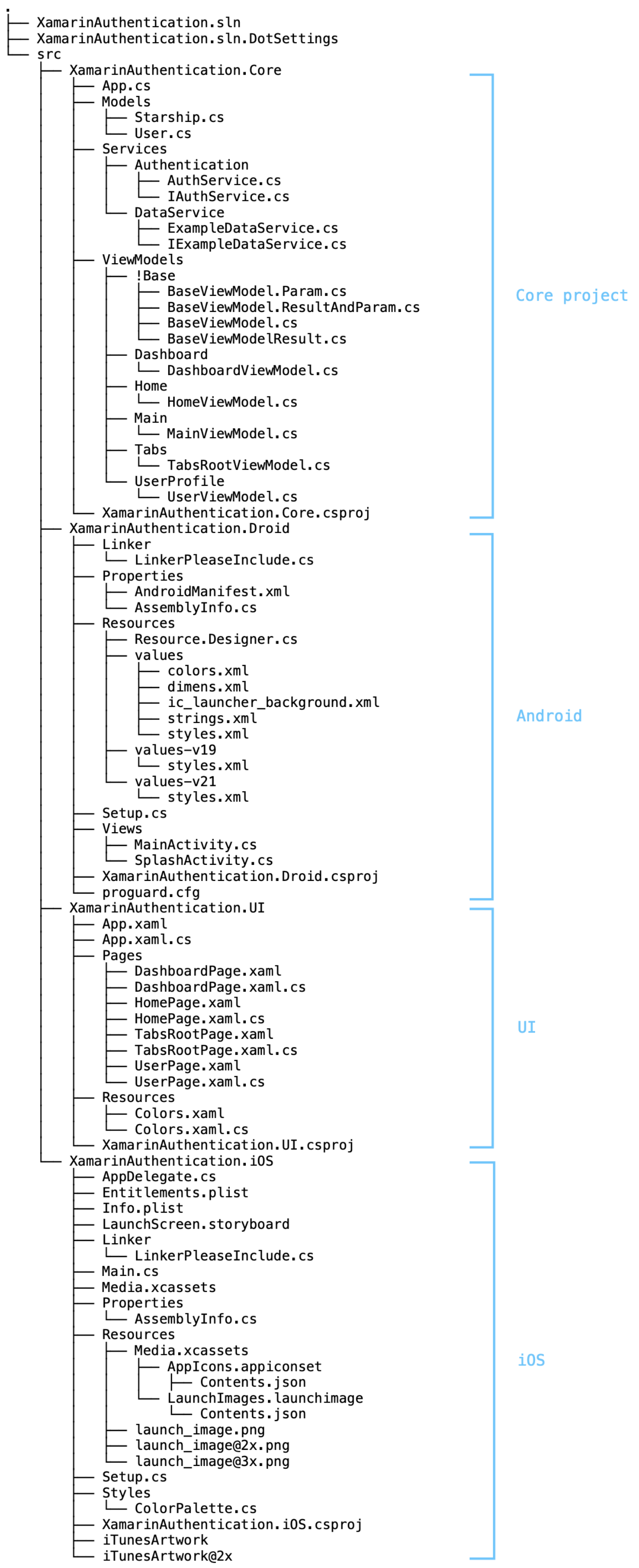
  When design of the application is not so important for the company and the project budget is tight, the UI can be implemented in .NET Xamarin.Forms cross-platform framework. UI is defined in Extensible Application Markup Language (XAML) files, which are basically "smarter" standard XML. There is limitation to the UI elements already defined in Xamarin.Forms so there may be some platform specific missing. We simply use, for example `<Button>`, element and then during the run time of the application, it is mapped into native component using each platforms rendered. These renderers are extendable and it is also possible to develop own custom renderers for custom elements. With this approach UI can be implemented only once for all platforms as a separate project next to the iOS, Android (and UWP) project as can be seen in figure 1.4 - structure tree of the example application, or for each platform separately inside each corresponding project.

- .NET MAUI

  This is open-source and also cross-platform framework for creating mobile and desktop applications. It is an evolution of Xamarin.Forms currently still under the main active development by Microsoft. Current version (12 by the time of writing this thesis) of pre-release preview was published January 19th 2022 [15].

1.2.2 Kotlin multiplatform

Multiplatform programming in Kotlin language is currently still in alpha stage so its API changes a lot and the most features are still experimental. It allows to share



```
.
├── XamarinAuthentication.sln
├── XamarinAuthentication.sln.DotSettings
└── src
    ├── XamarinAuthentication.Core
    │   ├── App.cs
    │   ├── Models
    │   │   ├── Starship.cs
    │   │   └── User.cs
    │   ├── Services
    │   │   ├── Authentication
    │   │   │   ├── AuthService.cs
    │   │   │   └── IAuthService.cs
    │   │   └── DataService
    │   │       ├── ExampleDataService.cs
    │   │       └── IExampleDataService.cs
    │   ├── ViewModels
    │   │   ├── !Base
    │   │   │   ├── BaseViewModel.Param.cs
    │   │   │   ├── BaseViewModel.ResultAndParam.cs
    │   │   │   ├── BaseViewModel.cs
    │   │   │   └── BaseViewModelResult.cs
    │   │   ├── Dashboard
    │   │   │   └── DashboardViewModel.cs
    │   │   ├── Home
    │   │   │   └── HomeViewModel.cs
    │   │   ├── Main
    │   │   │   └── MainViewModel.cs
    │   │   ├── Tabs
    │   │   │   └── TabsRootViewModel.cs
    │   │   └── UserProfile
    │   │       └── UserViewModel.cs
    │   └── XamarinAuthentication.Core.csproj
    ├── XamarinAuthentication.Droid
    │   ├── Linker
    │   │   └── LinkerPleaseInclude.cs
    │   ├── Properties
    │   │   ├── AndroidManifest.xml
    │   │   └── AssemblyInfo.cs
    │   ├── Resources
    │   │   ├── Resource.Designer.cs
    │   │   ├── values
    │   │   │   ├── colors.xml
    │   │   │   ├── dimens.xml
    │   │   │   ├── ic_launcher_background.xml
    │   │   │   ├── strings.xml
    │   │   │   └── styles.xml
    │   │   ├── values-v19
    │   │   │   └── styles.xml
    │   │   └── values-v21
    │   │       └── styles.xml
    │   ├── Setup.cs
    │   ├── Views
    │   │   ├── MainActivity.cs
    │   │   └── SplashActivity.cs
    │   ├── XamarinAuthentication.Droid.csproj
    │   └── proguard.cfg
    ├── XamarinAuthentication.UI
    │   ├── App.xaml
    │   ├── App.xaml.cs
    │   ├── Pages
    │   │   ├── DashboardPage.xaml
    │   │   ├── DashboardPage.xaml.cs
    │   │   ├── HomePage.xaml
    │   │   ├── HomePage.xaml.cs
    │   │   ├── TabsRootPage.xaml
    │   │   ├── TabsRootPage.xaml.cs
    │   │   ├── UserPage.xaml
    │   │   └── UserPage.xaml.cs
    │   ├── Resources
    │   │   ├── Colors.xaml
    │   │   └── Colors.xaml.cs
    │   └── XamarinAuthentication.UI.csproj
    └── XamarinAuthentication.iOS
        ├── AppDelegate.cs
        ├── Entitlements.plist
        ├── Info.plist
        ├── LaunchScreen.storyboard
        ├── Linker
        │   └── LinkerPleaseInclude.cs
        ├── Main.cs
        ├── Media.xcassets
        ├── Properties
        │   └── AssemblyInfo.cs
        ├── Resources
        │   ├── Media.xcassets
        │   │   ├── AppIcons.appiconset
        │   │   │   ├── Contents.json
        │   │   └── LaunchImages.launchimage
        │   │       └── Contents.json
        │   ├── launch_image.png
        │   ├── launch_image@2x.png
        │   └── launch_image@3x.png
        ├── Setup.cs
        ├── Styles
        │   └── ColorPalette.cs
        ├── XamarinAuthentication.iOS.csproj
        ├── iTunesArtwork
        └── iTunesArtwork@2x
```

Fig. 1.4 Structure tree of the example application in Xamarin.Forms

common codebase across various build targets and platforms. It is important to distinguish Kotlin multiplatform development of mobile applications for iOS and Android from development for other platforms and environments (for example web or desktop). This section will furthermore focus on Kotlin multiplatform development of mobile applications as it is aim of this thesis.

This is provided by Kotlin Multiplatform Mobile (KMM) which is an SDK specially designed for developing multiplatform applications for iOS and Android. According to JetBrains, it combines benefits of both cross-platform and native applications development and allows maintain a single codebase for networking, data storage, analytics, and other logic of Android and iOS applications [16]. UI is then designed separately using Jetpack Compose for Android and SwiftUI for iOS. Development is possible in Android Studio IDE or Intellij IDEA using KMM plugin.

Application composition

Each KMM application consist of three basic components [16]:

- Shared module

  A Kotlin module that contains common logic for both Android and iOS applications. It builds into an Android library and an iOS framework and uses Gradle as a build system with the Kotlin Multiplatform plugin applied. There is housed all the common codebase of the application. Because some logic may be platform specific, Kotlin offers so-called "expect/actual" mechanism. Source code of this shared module is then organized into three source assets:

  - `commonMain` – Contains source code working for both platforms including "expect" declarations
  - `androidMain` – Contains Android-specific code including "actual" implementation
  - `iosMain` – Contains iOS-specific code including "actual" implementation

- Android application

  A Kotlin module that builds into native Android application. It uses Gradle as a build system.

- iOS application

  An Xcode project that builds into native iOS application. It uses its own build system from Xcode.

Root project is Gradle project which holds these mentioned components: shared module, Android and iOS applications and global configuration files (`build.gradle`, `gradle.properties`, `local.properties` and `settings.gradle`). It does not hold source code of the KMM application.

Kotlin

Kotlin is a modern, free and open-source statically typed programming language, yet is still 100 % interoperable with Java and targets the Java Virtual Machine (JVM), however can also compile into machine code (through LLVM) or JavaScript. It was founded by JetBrains and announced as a main programming language for Android applications by Google in 2019 [16]. It combines object-oriented and functional programming. Kotlin was designed to eliminate danger of null pointer reference, so called "the billion dollar mistake", by introducing nullable and non-nullable data types. Despite functional programming it also supports extension functions, usage of lambdas and reflection. Asynchronous programming is solved by threading, callbacks, futures and promises, and the most appraised coroutines.

Key differences between Kotlin and Java:

- Kotlin combines object-oriented and functional programming while Java is limited to object-oriented programming.
- Kotlin supports extension functions (also for primitive data types) but Java does not.
- Kotlin does not support static members while Java uses them.
- Kotlin does not support implicit conversions but Java supports.
- Kotlin handles null-safety but Java does not.
- Kotlin handles variables of primitive types as objects whilst Java does not take them as objects.
- Kotlin does not require any variable datatype specifications while Java requires.

- Kotlin supports lambdas and inline functions while Java supports only lambdas (since Java version 8).
- Kotlin does not require semicolons whilst Java needs them.
- Kotlin does not support Java raw types.
- Kotlin does not have checked exceptions but Java does.
- Kotlin syntax is less verbose than Java.

1.2.3 Flutter

Flutter is an open-source framework developed by Google for multi-platform development of applications. It was first released in May 2017 and currently supports Android, iOS, macOS, Windows, Linux, Google Fuchsia and web applications. They are compiled into native code for each platform which results in super fast performance, same as original native applications.

Development of Flutter applications is done in the Dart programming language which is also developed and maintained by Google. Intellij IDEA, Android Studio, Visual Studio and Eclipse IDE can be used as IDEs. On macOS, Windows and Linux Flutter runs on Dart Virtual Machine (VM) which features Just-In-Time Compilation (JIT) of source code allowing so-called "hot reload" which allows modification of the source code and code injection into running application which saves a lot of development time [17].

Dart

Dart is object-oriented and class-based programming language. Its syntax is quite similar to C programming language but in addition it has garbage collection functionality. Originally Dart was compilable only into JavaScript programming language but developers later added compilation into native code. Web applications are still compiling into JavaScript in Chrome web browser, standalone applications are shipped together with Dart SDK and Dart VM, and (multiplatform) native applications are compiled into native code for iOS and Android OSs [18].

1.3 Web-based multiplatform approach

These applications are deployed in a native container that uses mobile WebView object. They are in fact web sites and just adapted for mobile device display, typically developed by web developers using basic web technologies such as Hypertext Markup Language (HTML), HTML 5, Cascading Style Sheets (CSS), JavaScript and other popular web frameworks. This approach extremely cuts development costs for the companies as they can use less human resources and also do not need to develop separate applications for each platform. Popular web-based mobile applications frameworks are Electron, React native and Ionic (previously Apache Cordova [5] and PhoneGap) [19] [20] [21].

1.3.1 Electron

Electron is free and open-source framework for building web-based mobile applications. It was originally developed for Atom code editor by GitHub, which is subsidiary of Microsoft, and initially released July 2013. Most of Electron's APIs are written in C++ and Objective-C programming languages and then exposed directly to the application code through JavaScript bindings [22]. The principle is to build standard web application (with consideration of mobile devices display and resolution) in JavaScript, HTML and CSS, and embed in Chromium [6] rendering engine and Node.js runtime. This basically means that the application is a single-use web browser mobile application (Google Chrome) displaying just one website (this application). Because of this, these applications eat enormous amount of device's Random-access Memory (RAM) and their performance is much slower than performance of native applications.

1.3.2 React native

This is open-source UI framework developed and maintained by Facebook. It allows developers to use React JavaScript framework along with each platform's native APIs for building mobile multi-platform applications [23]. But again, same to previous technologies, all applications can be developed on any OS, but iOS applications ca be build and deployed only using Mac computers (directly or via network connection). It uses declarative approach of programming. Core source code of each application is written in React.js programming language and then the UI components are declared using this

[5] Previously PhoneGap, mobile application development framework developed by Nitobi

[6] Free and open-source web browser project, base of Google Chrome web browser

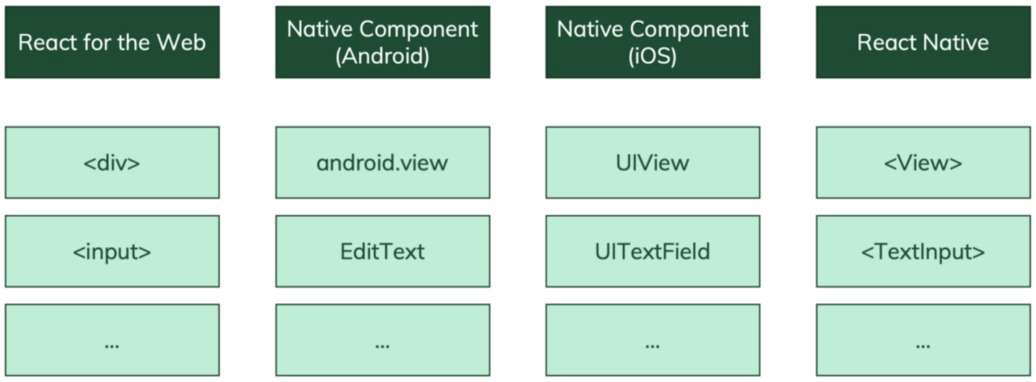


Fig. 1.5 Comparison of React and native UI elements
[24]

React native framework which provides correct translation of declared elements into each platform's native elements as seen in the figure 1.5. What is interesting is that the core source code of the application in JavaScript is not compiled into native code, but only the UI elements are compiled into native code – native elements [23]. Thanks to this React native applications' UI response is faster to the users' interaction than for example Electron applications.

1.3.3 Ionic

Ionic is open-source UI toolkit for building mobile and web applications again using HTML, CSS and JavaScript originally created by Drifty. Original version released in 2013 was built on top of AngularJS [7)] and Apache Cordova, nevertheless latest release was rebuild as a set of web components to support other web JavaScript-based frameworks such as Angular [8)], previously described React and Vue [9)] [25]. Alternatively, it can be used standalone without any frontend framework. But access to hardware interfaces through each platforms' provided API capabilities is still provided through Cordova. Development of Ionic applications was originally done in Ionic Studio IDE however it is no longer supported and developers can use Microsoft's Visual Studio Code with installed Ionic plugin instead.

[7)] Discontinued free and open-source JavaScript-based web framework developed by Google

[8)] TypeScript-based free and open-source web application framework developed by Google, descendant of AngularJS

[9)] Open-source MVVM frontend JavaScript framework developed by Evan You

2 Model-View-ViewModel

Model-View-ViewModel (MVVM) is a software architectural pattern first designed by John Gossman, Windows Presentation Foundation (WPF) 1) and Silverlight 2) software architect at Microsoft, in 2005 initially to use with WPF [26]. But usage of this architectural pattern quickly spread from Windows applications into any other applications like web and mobile and it eventually made its way into almost every UI-based framework.

The aim is to separate program logic from the UI controls – business layer from the presentation layer. It helps organize source code, separate it into modules and logical concerns to make continuous development easier and faster, and enhances simplicity and testability.

2.1 Application composition

The separation of source code is divided into Model, View and ViewModel components:

- Model

  It represents the whole data access layer - all the data and their set of properties and methods, application data as a class objects, retrieved from Object-Relational Mapping (ORM) or from the database. Model is responsible for managing these data and to ensure its consistency and validity. These data are retrieved by the ViewModel.

- View

  View is the client interface of the application. It houses collection of all the visible elements of the application and handles user's input. This includes UI, animations and all the text. The meaning is to make UI independent from the "code behind" and also platform-independent. View is the simplest component because it does not know about any other components and it does not control anything. It can not obtain any data directly from the Model because it is unaware of the Model and ViewModel, however ViewModel, despite it is generally unaware of the View, is aware of the View's needs and provides the desired data. One View is able to communicate and obtain data only from one singe ViewModel.

1) Free and open-source graphical subsystem (similar to Windows Forms) developed by Microsoft
2) Discontinued rich web applications framework developed by Microsoft

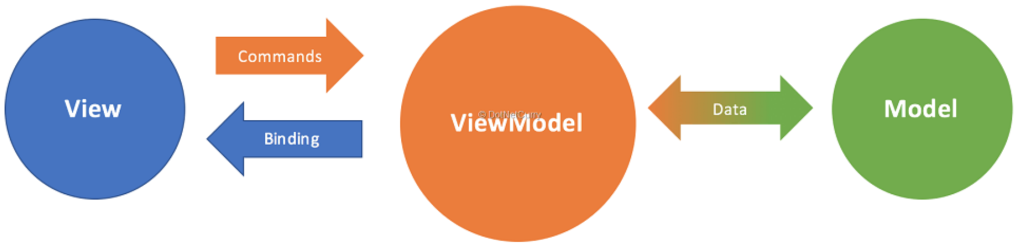


Fig. 2.1 Model-View-ViewModel schema
[27]

- ViewModel

  This is middle layer between Model and View, it has direct access to the Model, but is unaware of the View. One ViewModel can have access to multiple Models, as a one-to-many relation and encapsulates business logic and data for the View. Communication with the View is provided through data binding.

  Data binding is the key technology on which the whole MVVM architecture is based on. It provides automated two-way connection between the ViewModel and View. Additionally, a converter may be attached to the binder, for example to format currency float value into user-friendly readable string with corresponding globalization.

2.2 MvvmCross

MvvmCross is a cross-platform MVVM framework to create Xamarin.iOS, Xamarin.Android, Xamarin.Mac, Xamarin.Forms, UWP and WPF applications following MVVM architectural pattern [28]. To begin development of MvvmCross Xamarin application there is NuGet plugin for Visual Studio called MvxScaffolding developed by Jonathan Froom which is a customizable template used to scaffold a cross-platform MvvmCross applications [29].

Typical MvvmCross application consists of two parts – Core and UI. Core project contains all the ViewModels, Services, Models and business logic, and UI project contains Views and platform specific code for interacting with the Core [28]. For multiplatform applications, Core serves as a .NET Standard library and then there is (UI) project for each target platform. Optionally, the application can be split into more projects/assemblies for better readability and reusability of the source code.

2.2.1 Project composition

Composition of the MvvmCross application (as seen in figure 1.4):

- Core project
  - `App`
  - `AppStart`
  - Model + Services
  - ViewModels
- Each platform (UI) project
  - `Application`
  - `Setup`
  - Platform specific implementation of interfaces (declared in Core project), for example biometric authentication, networking or data storage
  - Resources (assets)
  - Views and ViewPresenter
  - UI declaration

App

This class inherits from `MvxApplication` and is responsible for registering custom objects on the Inversion of Control (IoC) container – mostly services, for registering main or root ViewModel and `IMvxAppStart` object. It should not be confused with `Application` and `ApplicationDelegate` classes. They are native classes for each platform and are part of each platform project [28].

AppStart

This is optional object which decides which ViewModel should be present as first.

ViewModels

In MvvmCross, all ViewModels have to inherit from `MvxViewModel`. They typically contain properties (model classes), commands, navigation to other ViewModels and injected dependencies.

Application

This handler object is responsible for native lifecycle events. For Android it is `MainActivity` or `MainApplication` and for iOS `AppDelegate` class. iOS `AppDelegate` class inherits from `MvxApplicationDelegate<MvxIosSetup<App>, App>` and obviously registers `Setup` class. `MainApplication` then inherits from `MvxAndroidApplication<MvxAndroidSetup<App>, App>`, but if the Android application uses Android Support packages, it must inherit from `MvxAppCompatApplication` and register `MvxAppCompatSetup` setup class instead [28].

Setup

`Setup` is responsible for bootstrapping MvvmCross and registering platform services. It inherits from `MvxIosSetup<Core.App, UI.App>` for iOS and `MvxAndroidSetup<Core.App, UI.App>` for Android.

Views and ViewPresenter

These are classes which house code behind the UI. ViewController classes in iOS are always extending `MvxViewController`. In Android these are represented by Fragments. ViewPresenters are singleton classes which act as a glue between Views and ViewModels, they provide clear separation between the ViewModel and the View layer [28].

2.2.2 Data binding

Data binding provides automated two-way connection between ViewModel and View. In MvvmCross optional converter – `MvxValueConverter` can be attached to the binder.

2.2.3 Navigation

Navigation between application screens is provided within ViewModels by injecting `IMvxNavigationService` into ViewModel. Then this navigation service is usually called from a command – `IMvxAsyncCommand` by the user click.

2.2.4 Inversion of Control

IoC is a programming principle essential in MvvmCross. It provides registration of services and singletons in one place and simply injecting them into ViewModels. Typical example is already mentioned navigation service, or any sorts of data service which provides data from the Model.

II. ANALYTICAL PART

3 MVVM implementation

3.1 MVVM in iOS

MVVM implementation in iOS is fully dependent on the chosen technology for UI implementation. There are three approaches:

- Storyboards and xib files

  Implementation of the View is happening strictly in the Xcode Interface Builder. Developers define which UI controls are used in the View, how they behave and the layout itself by drag and drop mechanism here, along with choosing of the various options in the Interface Builder.

  ViewModel then connects Models and Views (and may add some business logic related objects and functions) together. It must be implemented in the `UIViewController` classes since these are used for binding the Interface Builder defined UI elements to the code, thus creating the ViewModel.

- UIKit programmatically

  If the developers choose to define application's UI programmatically in the code while still using the UIKit framework, Views are class objects sub-classed from the UIKit's `View` class and conforming to the UIKit's prerequisites which are used by the OS to layout the application's UI. OS on that occasion calls overridden functions that are used to layout and set up the defined UI components to the display.

  ViewModels are created implementing `UIViewControllers` for the UI screens and components, however they contain `UIViews` directly and work by calling their functions, embedding them, and they also handle interaction with the user. Since the UI is defined solely in the code, there is a wider freedom in the possibility of splitting up the responsibilities of each `UIViewController` into multiple objects, thus the ViewModel can span multiple objects or ViewModels can be nested easily if needed.

- SwiftUI framework

  This is the most modern and clean approach outlining the View-ViewModel part of the MVVM architecture very clearly. This is also the way which was chosen for iOS example application for this thesis. Each screen must have its own ViewModel if it is supposed to have any kind of dynamic functionality and not used only to display some static content, because then obviously ViewModel is not needed. Each ViewModel can declare its own delegate (delegate is a protocol, which has similar purpose as interface in Kotlin) with functional requirements than have to be implemented by lower level objects:

```
protocol HomeViewModelDelegate: AnyObject {
    func requestAuthentication()
}

class HomeViewModel<DestinationView: View>
    : ObservableObject {

    weak var delegate: HomeViewModelDelegate?

    func authenticate() {
        delegate?.requestAuthentication()
    }
}
```

Functions prescribed in the delegate are then implemented in `AppCoordinator`:

```
extension IosAuthentificationAppCoordinator
    : HomeViewModelDelegate {

    func requestAuthentication() {
        .
        .
        .
    }
}
```

Simple UI elements might contain only state properties (for example label would contain just a string defining its state), whereas whole screen's ViewModel is a more complex structure containing many properties or even the screen logic. Navigation is usually implemented using `NavigationLink` which is a SwiftUI element, so it needs to contain hidden empty view, so it does not occupy any visual space on the screen. For example this `NavigationLink` which is attached to the `Log In` button is active only when the authentication was successful and then navigate user inside the application:

```
NavigationLink(
    isActive: $viewModel.wasAuthenticated,
    destination: viewModel
        .destinationViewAfterAuthentification
) { EmptyView().hidden() }
```

Main difference between the UIKit approach and SwiftUI is that in the UIKit implementation developers have to manually request to redraw the UI if the data were displayed before they were changed, causing the ViewModel implementation to be very complex. Thanks to the SwiftUI framework approach relying on the `@ObservedObject` mechanism along with the `@State` mechanism, each ViewModel change causes the SwiftUI framework to recalculate which part need to be updated and the framework automatically redraws only the relevant part of the UI, behaving in a very optimised manner.

3.2 MVVM in Android

Unfortunately, unlike Xamarin, there are dozens ways how to implement MVVM in Android and it is common that each company or each team handles this differently. Especially how ViewModels are used and responsible for. In the ideal world navigation between ViewModels is declared in the `nav_graph.xml` and used programmatically in the corresponding Views so ViewModels are free from navigation, and must be initialized in `MainActivity` class in `onCreate()` method. The following sample extracted from the `nav_graph.xml` shows how navigation is defined from `PersonsFragment.kt` into `PersonDetailFragment.kt`:

```xml
<fragment
        android:id="@+id/persons_destination"
        android:name="cz.verunka.example.views
          .persons.PersonsFragment"
        android:label="List of~persons"
        tools:layout="@layout/fragment_persons">

        <action
            android:id="@+id/action_open_person_detail"
            app:destination="@id/person_detail_destination" />
    </fragment>

    <fragment
        android:id="@+id/person_detail_destination"
        android:name="cz.verunka.example.views
          .persons.PersonDetailFragment"
        android:label="Person detail"
        tools:layout="@layout/fragment_person_detail" />
```

Here is shown how it is then initialized in `MainActivity` class in `onCreate()` method:

```kotlin
val bottomNavigationView = binding.bottomNavigation
// list of all navigation graphs
val navGraphIds = listOf(R.navigation.nav_graph)

// setup bottom navigation
val controller = bottomNavigationView.setupWithNavController(
    navGraphIds = navGraphIds,
    fragmentManager = supportFragmentManager,
    containerId = R.id.main_activity_fragment,
    intent = intent
)

// setup action bar
controller.observe(this, Observer { navController ->
    setupActionBarWithNavController(navController)
})
currentNavController = controller
```

And then it can be called from `PersonsFragment.kt` in `OnItemClick` method like this:

```
override fun onItemClick(person: Person) {
    findNavController()
    .navigate(R.id.person_detail_destination, bundle)
}
```

The most clear way of implementation is to have own ViewModel for each screen or item (again similarly to Xamarin) which is actually the original principle of MVVM. However, in real world, Android developers often use one ViewModel for many screens from the certain area to simplify the data flow. With just one ViewModel holding data for many screens, there is no problem as all Views (Fragments) have access to this ViewModel and its data. The caveat is that navigation must be fully independent from the ViewModel, which is opposite to Xamarin implementation of MVVM. Surprisingly developers overloads ViewModels with responsibilities which are not supposed to. Like instead of calling navigation in Fragments and binding data in XML files, they throw all this work into ViewModels. Then these ViewModels are huge and unreadable with hundreds of lines of source code. This sample of code show how is navigation called from the ViewModel:

```
fun onPersonClick(view: View) {
    view.findNavController().navigate(
        PersonsFragmentDirections.actionOpenPersonDetail()
    )
}
```

And then this method must be assigned to corresponding item in XML file like this:

```
android:onClick="@{viewModel::onPersonClick}"
```

3.3 MVVM in Xamarin

MVVM implementation in Xamarin is very easy and straightforward. Whole source code must be split into Model, ViewModels and Views, whereas Model and ViewModel are housed in Core project and Views in Xamarin.Forms UI project or in each platform (UI) project. Each screen has its own ViewModel where all the commands and navigation are implemented. The most popular technology for implementation is to use MvvmCross framework so it was chosen for this thesis as well. In pure Xamarin data binding is provided declaratively in XAML files using the keyword `Binding` in front of binded value. It the following example of code is shown binding text value `CurrentUser.Email` into text label, then `OpenEmailCommand` into tap gesture command with a command parameter `CurrentUser.Email`:

```
<Label Text="{Binding CurrentUser.Email}"
  TextDecorations="Underline">

    <Label.GestureRecognizers>

        <TapGestureRecognizer
          Command="{Binding OpenEmailCommand}"
          CommandParameter="{Binding CurrentUser.Email}"/>

    </Label.GestureRecognizers>

</Label>
```

In MvvmCross data binding is defined in the View classes programmatically for iOS and Android project and declaratively in XAML for Windows project. However, if the application is developed in Xamarin.Forms it is also defined declaratively in XAML files. The following sample of code extracted from `OnCreate()` method in Android project or from `ViewDidLoad()` method in iOS project shows how data binding is implemented:

```
var set = this.CreateBindingSet<ExampleView, ExampleViewModel>();

set.Bind(this.FindViewById<EditText>(Resource.Id.example_edit))
  .To(x => x.EditValue);

set.Bind(this.FindViewById<TextView>(Resource.Id.example_result))
  .To(x => x.ResultValue);

set.Apply();
```

Registration of services must be performed in `App` class in the `Initialize()` method in the Core project:

```
this.CreatableTypes()
    .EndingWith("Service")
    .AsInterfaces()
    .RegisterAsLazySingleton();
```

III. PRACTICAL PART

4 Example application design

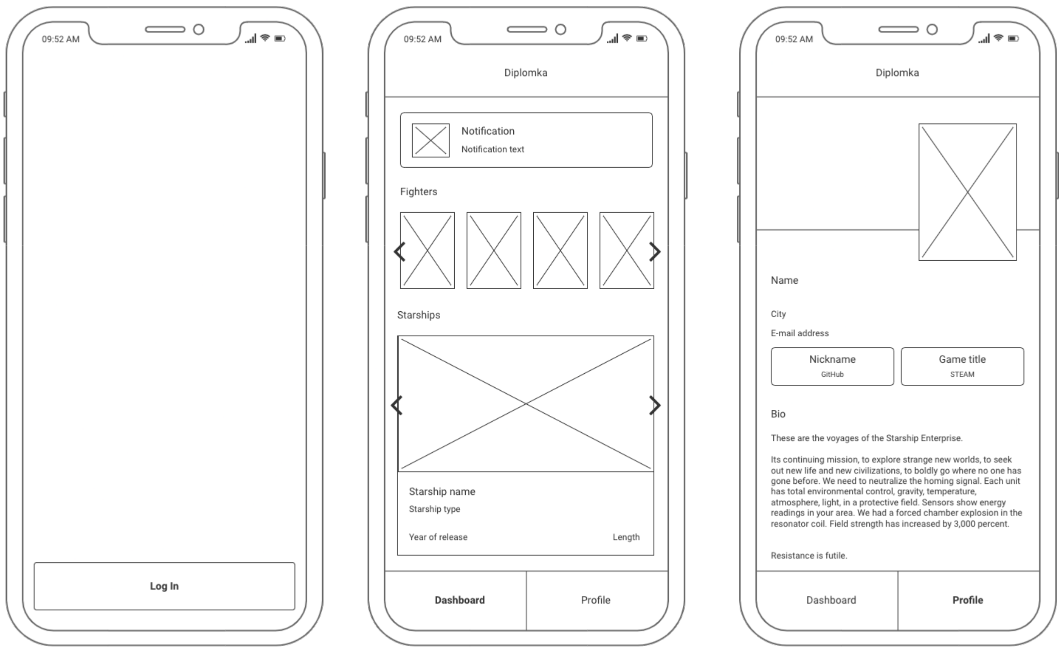
4.1 Components

For demonstration how MVVM is implemented across different platforms and for the simplicity of the example application, there were selected following components:

- Biometric authentication
- Bottom navigation
- Notification card view
- Horizontal scroll view with images
- Carousel horizontal scroll view with complex card views
- Buttons opening external application

4.2 Wireframe

Wireframe of this application is displayed in figure 4.1.



These are the voyages of the Starship Enterprise.

Its continuing mission, to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. We need to neutralize the homing signal. Each unit has total environmental control, gravity, temperature, atmosphere, light, in a protective field. Sensors show energy readings in your area. We had a forced chamber explosion in the resonator coil. Field strength has increased by 3,000 percent.

Resistance is futile.

Fig. 4.1 Wireframe of example application

4.3 Use-case diagram

Use-case of this application is displayed in figure 4.2.

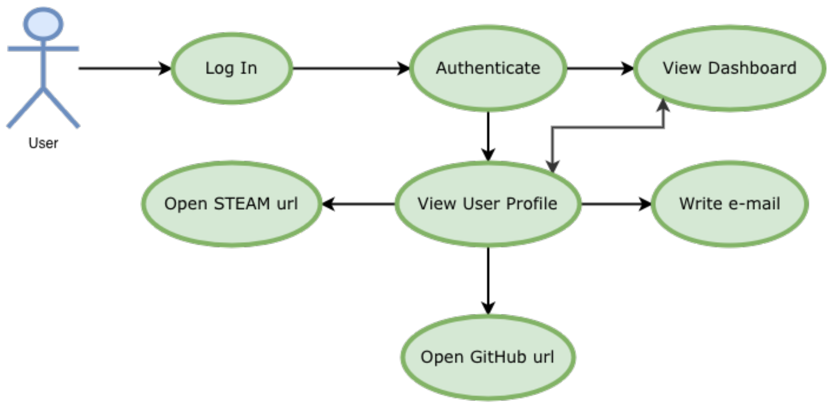


Fig. 4.2 Use-case diagram of example application

4.4 Activity diagram

Activity diagram of this application is displayed in figure 4.3.

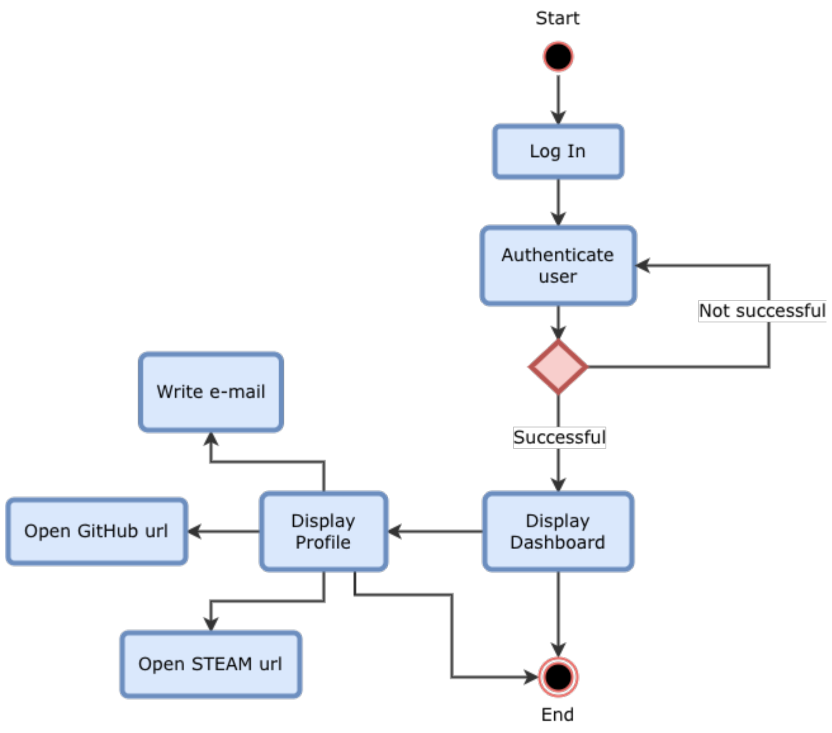


Fig. 4.3 Activity diagram of example application

5 iOS application

Example iOS native application was developed in XCode IDE using Swift programming language and SwiftUI framework for UI implementation. The main `App` structure class is `IosAuthenticationApp` which just sets the main coordinator and entry application's view:

```swift
@main
struct IosAuthenticationApp: App {

    let mainCoordinator = IosAuthentificationAppCoordinator()

    var body: some Scene {
        WindowGroup {
            mainCoordinator.homeView
        }
    }
}
```

Main coordinator – `IosAuthentificationAppCoordinator`, which is a final class, initializes all the services, ViewModels and main View. In this file are also implemented ViewModel delegates functions by their corresponding coordinators such as opening e-mail and URL, or biometric authentication. This shows how biometric authentication is implemented:

```swift
extension IosAuthentificationAppCoordinator:
    HomeViewModelDelegate {

    func requestAuthentication() {

        let localAuthenticationContext = LAContext()
        localAuthenticationContext
            .localizedFallbackTitle = "Use passcode"

        var authError: NSError?
        let reasonString = "We need Your biometric information
            for authentication."

        if localAuthenticationContext.canEvaluatePolicy(
            .deviceOwnerAuthentication,
            error: &authError
        ) {
            localAuthenticationContext.evaluatePolicy(
                .deviceOwnerAuthentication,
                localizedReason: reasonString
            ) { [weak self] success, evaluateError in

                if success {
                    self?.homeViewModel.wasAuthenticated = true
                } else {
                    guard let error = evaluateError else {
                        return
                    }
                    print("Biometric authentication not enabled:
                        \(error._code),
                        \(error.localizedDescription)"
                    )
                }
            }
        } else {
            guard let error = authError else {
                return
            }
            print("Biometric authentication failed:
                \(error._code), \(error.localizedDescription)"
            )
        }
    }
}
```

And this is how simple it is to open new e-mail or website in device's default e-mail and web browser applications:

```
func openEmail(_ email: String) {
    if let url = URL(string: "mailto:\(email)") {
        UIApplication.shared.open(url)
    }
}

func openUrl(_ url: URL) {
    UIApplication.shared.open(url)
}
```

Additionally there is `Info.plist` file with reasonable request for biometric authentication:

```
<key>NSFaceIDUsageDescription</key>
<string>
    We need Your biometric information for authentication.
</string>
```

Main source code is then divided into so-called groups: `Models`, `ViewModels`, `Views`, `Services`, and `Resources`.

5.1 Models

There are `User` and `Starship` model classes which are actually structs in Swift programming language. `Starship` struct must additionally implement `Identifiable` protocol so it can be later used in a list. Both need to have imported `Foundation` and `SwiftUI` libraries.

5.2 ViewModels

Usually each ViewModel obeys its own defined delegate which is a protocol, and have imported `SwiftUI` library. Exception is when ViewModel does not need any service or request for any special function which must be implemented in the coordinator. This iOS application consist of three ViewModels:

- `HomeViewModel`

  This is an entry ViewModel of this application and its call for biometric authentication request, which is prescribed in its `HomeViewModelDelegate` protocol, is implemented in the `IosAuthentificationAppCoordinator` already explained above.

- `UserViewModel`

  `UserViewModel` is defining its `UserViewModelDelegate` protocol with prescribed two functions: `openEmail()` and `openUrl()`. These functions are also implemented in the `IosAuthentificationAppCoordinator`. Furthermore this ViewModel is obtaining data about `User` from `ExampleDataService`. The following `openUrl()` function shows how a delegate function is called from a ViewModel function:

```
func openUrl(_ url: URL?) {
    guard let url = url else { return }
    delegate?.openUrl(url)
}
```

- `DashboardViewModel`

  This ViewModel does not house any functions so it does not need a delegate. There is only initialization where data (list of starships and fighters) are received from data service.

5.3 Views

Views are implemented programmatically using SwiftUI framework, and same as ViewModels, they are structs. Each View can implement `PreviewProvider` so the defined UI can be live previewed in the XCode IDE. Their corresponding ViewModel is annotated as an `@ObservedObject` for effective redrawing when some date change. There are three Views for this application plus `TabBar` container:

- `Home`

  `HomeView` contains just a `Log In` button which triggers biometric authentication and then navigate user inside the navigation. Because of this, there is `NavigationLink` attached to the button which takes destination View. This shows how `PreviewProvider` is implemented here:

```
struct HomeView_Previews: PreviewProvider {
    static var previews: some View {
        HomeView(viewModel: HomeViewModel<EmptyView>(
            destinationViewAfterAuthentification: {
                EmptyView()
            }
        ))
    }
}
```

- `TabBar`

  `TabbarView` is responsible for navigation bar at the top of the screen with application name, and for bottom bar navigation with tabs. Tabs are first set up in `TabbarViewFactory` class. `TabbarView` then contains `TabView` with `Dashboard` and `User` Views, together with setup of navigation bar. Because we do not want to be able to navigate back to the `HomeView` with the `Log In` button, `BackButton` is disabled in the navigation bar. Images rendering mode is set to template, so they can change color based on if the certain tab is selected or not:

```
var body: some View {
    TabView {
        dashboardView
            .tabItem {
                Image("home").renderingMode(.template)
                Text(Strings.dashboardView.rawValue)
            }

        userView
            .tabItem {
                    Image("person").renderingMode(.template)
                    Text(Strings.profileView.rawValue)
               }
     }
     .navigationBarBackButtonHidden(true)
     .navigationTitle(Text(Strings.appName.rawValue))
     .navigationBarTitleDisplayMode(.inline)

}
```

- `User`

  `UserView` displays information about example user. This sample of code shows how data binding is done in SwiftUI Views – for example how user's name is binded into `Text` element:

```
 Text(viewModel.user.name)
     .frame(maxWidth: .infinity, alignment: .leading)
     .font(.system(size: 24, weight: .regular))
```

  On e-mail (and GitHub and STEAM card views) click is attached `.onTapGesture` recognizer with a call to perform an action. In this case to open a new e-mail to example user's e-mail address:

```
.onTapGesture {
     viewModel.openEmail(viewModel.user.email)
}
```

- `Dashboard`

  `DashboardView` houses example notification card view, horizontal list of images, and carousel list of complex cards about starships. It is very simple in SwiftUI to define such lists, for example this is how starships list is defined and binded using `ForEach` function:

```
ScrollView(.horizontal, showsIndicators: false) {
     HStack {
          ForEach(viewModel.starships) { starship in

               // Starship card view.
               VStack(spacing: 8) {

                    Image(starship.photo)
                         .resizable()
                         .scaledToFill()
                         .frame(width: 340, height: 160)
                         .clipped()
                    .
                    .
                    .
```

5.4 Services

There is just one service – data service. It is prescribed in a `ExampleDataService` protocol which needs to have imported `Foundation` and also `SwiftUI` libraries. `ExampleDataServiceImpl` is then a class which implements this protocol. There are defined example data for this application. `User` and `Starship` models are defined within the same file but an `extension`. For example the following sample of code shows how a starship entity is created. What is interesting is how easy it is to set image from the `Assets.xcassets` catalogue, just by its name:

```
fileprivate extension Starship {
    static var enterprise: Starship {
        .init(
            name: "USS Enterprise",
            type: "NCC-1701 (Shuttlecraft)",
            year: 2245,
            length: "288.646 m",
            photo: "enterprise"
        )
    }
}
```

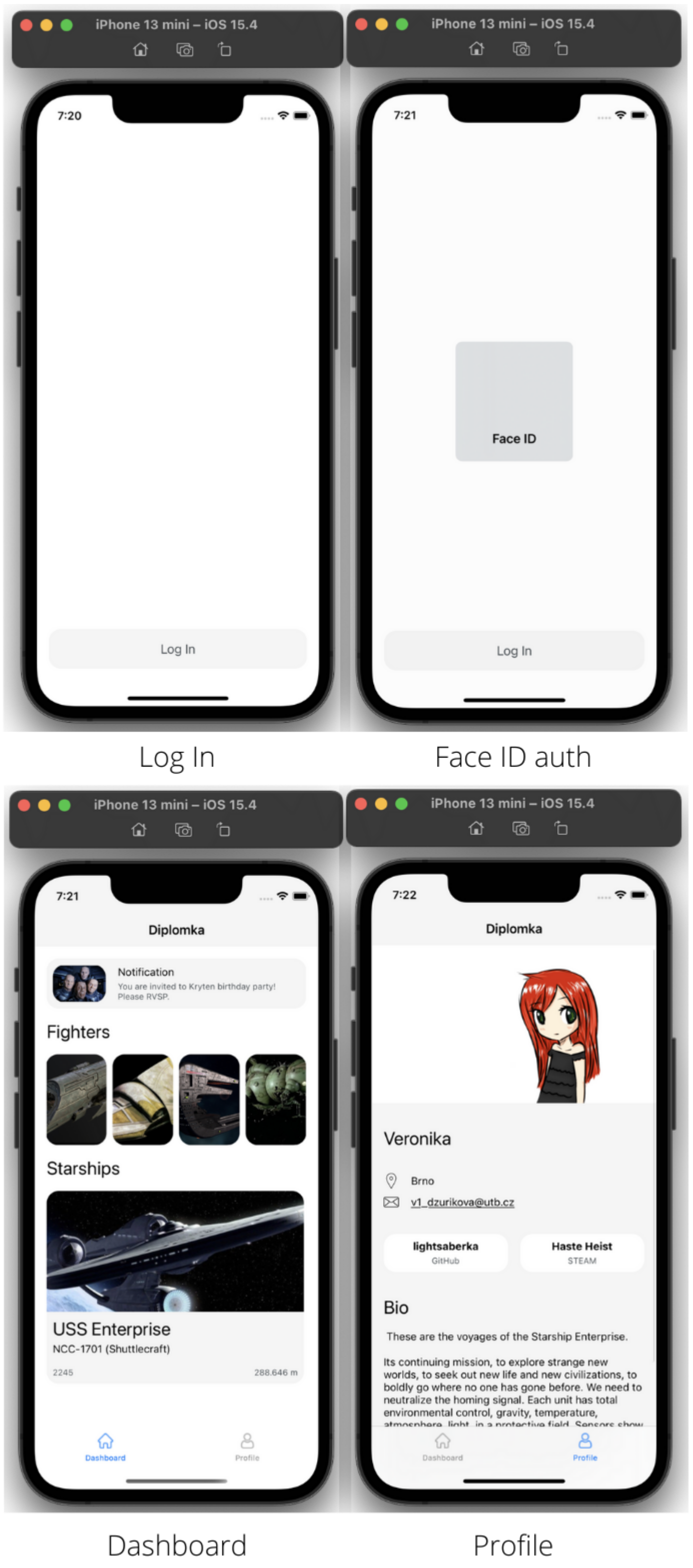
5.5 Resources

Just already mentioned `Assets.xcassets` catalogue contains all the images for the application. Colours are defined in the `Colors` enum and string resources in the `Strings` enum. Both these enums need to have imported `SwiftUI` library. For example this shows how custom colours are defined in SwiftUI:

```
enum Colors {
    static var dimGray: Color {
        Color(red: 0.30, green: 0.33, blue: 0.35)
    }

    static var whiteSmoke: Color {
        Color(red: 0.96, green: 0.96, blue: 0.96)
    }
}
```

Screenshots of the native iOS application can be seen in figure 5.1.



You are invited to Kryten birthday party! Please RVSP.

These are the voyages of the Starship Enterprise.

Its continuing mission, to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. We need to neutralize the homing signal. Each unit has total environmental control, gravity, temperature, atmosphere, light, in a protective field. Sensors show

Fig. 5.1 Native iOS application

6 Android application

Example Android native application was made in Android Studio IDE using its default wizard. Source code is implemented in Kotlin programming language and UI is defined in XML files. Main properties of the application are defined in `manifest` file – it was already prepared by the setup wizard, so only permission for biometric authentication request had to be defined. All the dependencies such as Material design, Biometric utility, Lifecycle, Navigation, et cetera, with required version are defined in `build.gradle` Gradle script file together with enabled data binding. The following sample of code shows how such a dependency is defined:

```
dependencies {

    implementation 'androidx.core:core-ktx:1.7.0'

    // biometric auth
    implementation "androidx.biometric:biometric:1.1.0"
    .
    .
    .
```

Main source code of the application is divided into four directories: obviously `model`, `viewModel`, `view` and `services`, and all the assets are located in the `res` (resources) directory.

6.1 Model

There are `User` and `Starship` data classes. When there is image resource property (for example photo of the starship), it must be annotated in constructor with `@DrawableRes` annotation. Setters and Getters are no needed thanks to Kotlin, so the whole class implementation is pretty succinct.

6.2 ViewModel

There are `Dashboard` and `Profile` ViewModels which both implement `androidx-.lifecycle.ViewModel`. They contain methods for providing data from repository for views (fragments).

6.3 View

`View` directory is composed of `MainActivity` and all the fragments. Fragments use their corresponding ViewModels as a lazy property obtained from `ViewModelProvider`.

- `MainActivity` and `MainFragment`

  This sample of code shows how `MainViewModel` is obtained in `MainFragment`:

```
private val viewModel: MainViewModel by lazy {
    ViewModelProvider(this)[MainViewModel::class.java]
}
```

  Then binding is set up in the `onCreateView()` method by inflating the corresponding XML fragment and then used to bind other properties, and launch biometric authentication on button click (`biometricPrompt` and `promptInfo` are also for simplicity implemented here in `onCreateView()` method, but usually in corporate life it is done in separate service or use case):

```
val binding: MainFragmentBinding = DataBindingUtil.inflate(
    inflater,
    R.layout.main_fragment,
    container,
    false
)

binding.lifecycleOwner = viewLifecycleOwner
binding.viewModel = viewModel

binding.mainButtonLogIn.setOnClickListener {
    biometricPrompt.authenticate(promptInfo)
}
```

  If the biometric authentication fails, there are allowed other methods – credentials (PIN code, password or pattern; depends on the certain device used). They are listed and then set into `promptInfo` with `.setAllowedAuthenticators(authenticators)` method:

```
val authenticators =
    BiometricManager.Authenticators.BIOMETRIC_STRONG or
    BiometricManager.Authenticators.BIOMETRIC_WEAK or
    BiometricManager.Authenticators.DEVICE_CREDENTIAL
```

- `TabsRootFragment`

  Next important fragment is `TabsRootFragment` which is responsible for bottom tab navigation when the user enters the application by successful authentication. There is auxiliary method for loading chosen fragment:

```
private fun loadFragment(fragment: Fragment) {
    val transaction = parentFragmentManager
        .beginTransaction()
    transaction.replace(
        R.id.tabs_layout_container,
        fragment
    )
    transaction.addToBackStack(null)
    transaction.commit()
}
```

  And these fragments are binded in the `onCreateView()` method:

```
binding.tabsNavigationView
    .setOnNavigationItemReselectedListener {
        when (it.itemId) {
            R.id.menu_dashboard -> {
                loadFragment(DashboardFragment())
                return@setOnNavigationItemReselectedListener
            }
            R.id.menu_profile -> {
                loadFragment(ProfileFragment())
                return@setOnNavigationItemReselectedListener
            }
        }
}
```

- `ProfileFragment`

  `ProfileFragment` does not contain anything special besides opening e-mail and web browser applications as an intent on e-mail address or buttons click (e-mail address needs to have `"mailto:"` prefix for working property as an input parameter for this method):

```
private fun openUrl(url: String) {
    val openURL = Intent(Intent.ACTION_VIEW)
    openURL.data = Uri.parse(url)
    startActivity(openURL)
}
```

- `DashboardFragment`

  `DashboardFragment` is housing two recycler views, one with pictures of fighter ships and second with more complex card views containing starship picture and information. Adapters are required for these items to later bind them in the fragment. Each adapter must implement `RecyclerView.Adapter` with corresponding holder implementing `RecyclerView.ViewHolder`. Following sample of code shows how `StarshipViewHolder` is implemented as a inner class inside the `StarshipAdapter`:

```
inner class StarshipViewHolder(
   private val viewBinding: StarshipCardViewBinding
) : RecyclerView.ViewHolder(viewBinding.root) {

    fun onBind(position: Int) {
        val item = starshipList[position]
        viewBinding.starship = item
    }
}
```

And then it can be used in the `StarshipAdapter` implementation:

```
class StarshipAdapter(val context: Context?) :
    RecyclerView.Adapter<
        StarshipAdapter.StarshipViewHolder
    >() {
    var starshipList: List<Starship> = ArrayList()

    override fun onCreateViewHolder(
       parent: ViewGroup, viewType: Int
    ): StarshipViewHolder {

        val viewBinding: StarshipCardViewBinding
            = DataBindingUtil.inflate(
                LayoutInflater.from(parent.context),
                R.layout.starship_card_view,
                parent,
                false
            )
        return StarshipViewHolder(viewBinding)
    }

    override fun getItemCount(): Int {
        return starshipList.size
    }

    override fun onBindViewHolder(
       holder: StarshipViewHolder,
       position: Int
    ) {
        holder.onBind(position)
    }

    .
    .
    .
```

DashboardFragment does not hold list of, for example, these starships, but just this adapter. And this sample of code shows how is this adapter then binded to fill recycler view with starships obtained from the repository:

```
starshipAdapter = StarshipAdapter(context)

binding.dashboardRecyclerStarships.layoutManager
    = LinearLayoutManager(
        activity,
        LinearLayoutManager.HORIZONTAL,
        false
    )

binding.dashboardRecyclerStarships.adapter = starshipAdapter

binding.dashboardRecyclerStarships
    .isNestedScrollingEnabled = false

starshipAdapter.setStarships(viewModel.starships)
```

And because we want the bottom recycler view with starship cards to act as a carousel view (otherwise called snapping), there is necessary to attach SnapHelper to recycler view:

```
val snapHelper: SnapHelper = PagerSnapHelper()
snapHelper.attachToRecyclerView(
    binding.dashboardRecyclerStarships
);
```

6.4 Services

So called repositories are used for proper data management in Android. Because this example application contains just little amount of data, there is only one repository. Usually there are data obtained from some API or database, but here just for demonstration purposes are data created locally. Repository moreover contains public methods such as `getUser()` or `getStarships()` which are called by ViewModels.

6.5 Resources

Resource directory in Android is called `res`. Here very important resource is navigation graph `main-_navigation.xml`. It contains list of fragments and their actions; where is it possible to navigate further within the application. For example `TabsRootFragment` has two actions: to open Dashboard and to open Profile fragment:

```
<fragment
    android:id="@+id/tabs_root_destination"
    android:name="cz.verunka.droid.authentication.view
       .TabsRootFragment"
    tools:layout="@layout/tabs_root_fragment">

    <action
        android:id="@+id/action_open_dashboard"
        app:destination="@id/dashboard_destination"/>

    <action
        android:id="@+id/action_open_profile"
        app:destination="@id/profile_destination"/>
</fragment>
```

Bottom bar navigation tabs are set up for the `TabsRootFragment` in the `bottom_nav-_menu.xml` file:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/menu_dashboard"
        android:icon="@drawable/home"
        android:title="@string/dashboard_view"/>
    <item
        android:id="@+id/menu_profile"
        android:icon="@drawable/person"
        android:title="@string/profile_view"/>
</menu>
```

Furthermore, there is `values` directory which houses themes, styles and colors of the application, and string resources. `drawable` directory is housing all the image assets (images for specific display resolutions may be placed in the corresponding resolutions sub directories, or if there are different images for dark mode, they are also placed in the specific night sub directories).

And last but not least is the `layout` directory where are all the fragments and another UI items located. For example `main_activity.xml` serves as a navigation host (container), so this XML is quite different than the others:

```
<androidx.fragment.app.FragmentContainerView
   xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/main_navigation"/>
```

The rest of the files are defining the UI. What may be interesting is inserting reference to ViewModel there, so data binding may be also executed from there, instead of programmatically in kotlin fragments. For example like this in `profile_fragment.xml`:

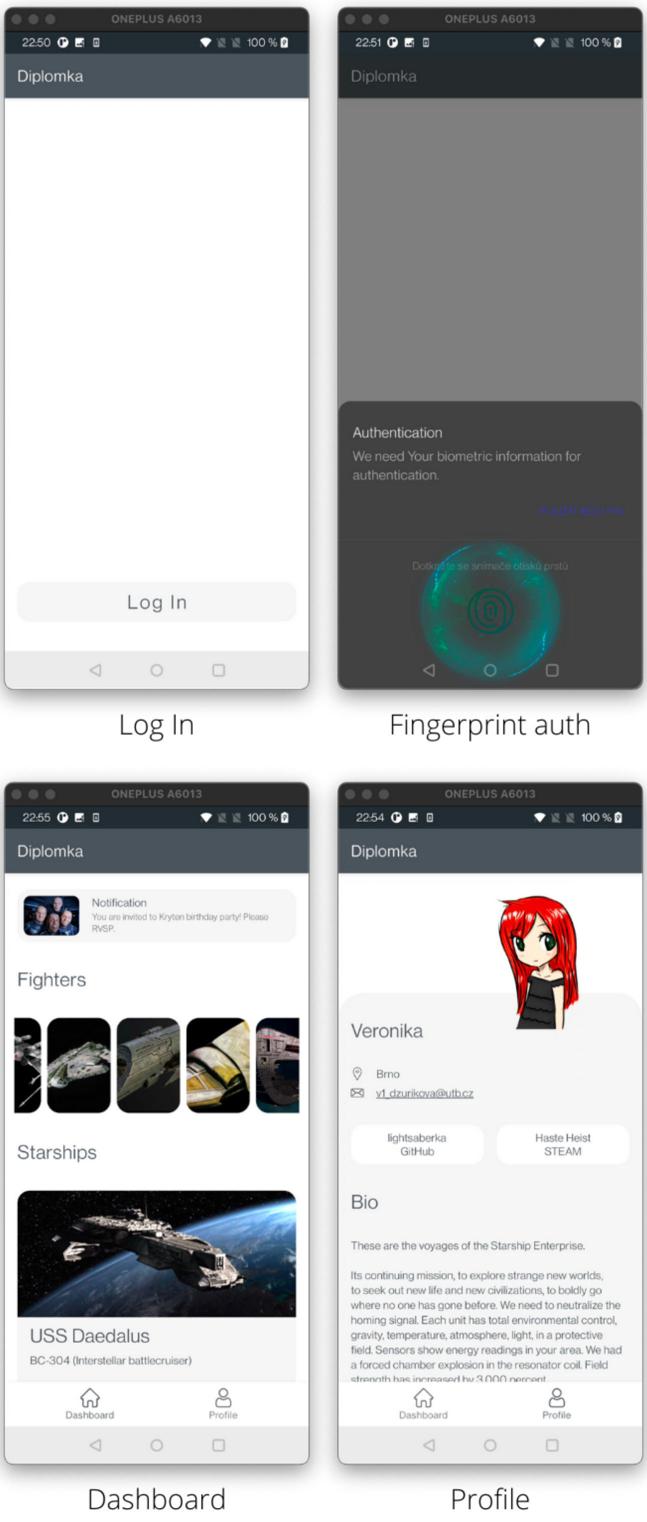
```
<data>
    <variable
        name="viewModel"
        type="cz.verunka.droid.authentication
           .viewModel.ProfileViewModel"/>
</data>
```

Also primitive data types or own Model classes may be inserted, like here in `starship-_card_view.xml`, and then used for example for binding image resource:

```
<data>
    <variable
        name="starship"
        type="cz.verunka.droid.authentication.model.Starship"/>
</data>
```

```
<ImageView
    android:id="@+id/starship_image"
    app:imageResource="@{starship.photo}"
    .
    .
    .
    />
```

Screenshots of the native Android application can be seen in figure 6.1.



We need Your biometric information for authentication.

You are invited to Kryten birthday party! Please RVSP.

These are the voyages of the Starship Enterprise.

Its continuing mission, to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. We need to neutralize the homing signal. Each unit has total environmental control, gravity, temperature, atmosphere, light, in a protective field. Sensors show energy readings in your area. We had a forced chamber explosion in the resonator coil. Field strength has increased by 3 000 percent

Fig. 6.1 Native Android application

7 Xamarin.Forms application

Example application in Xamarin.Forms was set using MvxScaffolding plugin in Visual Studio to set up basic MvvmCross source code structure and components. The structure can be already seen in figure 1.4 in the theoretical part of this thesis.

7.1 Core project

The Core project contains code base which is common for all platforms. There is `App` class where IoC, services and the starting ViewModel is initialised plus addition experimental feature – `CarouselView` used in the UI:

```
public override void Initialize()
{
    this.CreatableTypes()
        .EndingWith("Service")
        .AsInterfaces()
        .RegisterAsLazySingleton();

    this.RegisterAppStart<HomeViewModel>();

    Device.SetFlags(new string[] {"CarouselView_Experimental"});
}
```

`EndingWith("Service").AsInterfaces().RegisterAsLazySingleton()` means that it will look for all the services in the Core project and then registers them as singletons. Then these services can be injected and used in ViewModels.

7.1.1 Services

There are two custom services in this example application. Authentication service which handles biometric authentication for this application and for both, iOS and Android. If the device does not have these utilities, it ask for device's PIN code, if device has fingerprint utility, it asks for user's fingerprint and if device has face ID, it scans the user's face. If these biometric authentication attempts fail, it is of course possible to unlock with PIN code. `Plugin.Fingerprint.CrossFingerprint` plugin was used for this utility. This sample of code shows how this plugin is used and configured in this application:

```
public async Task<bool> Authenticate()
{
    var isAuthenticationAvailable = await CrossFingerprint
      .Current.IsAvailableAsync(true);

    if (!isAuthenticationAvailable) {
        Debug.WriteLine("Error: Biometric authentication
          is not available or is not configured.");
        return false;
    }

    // configure authentication prompt
    var conf = new AuthenticationRequestConfiguration(
      "Authentication",
      "We need Your biometric information for authentication."
    );

    // allow PIN/pattern authentication
    // if biometric is unsuccesfull
    conf.AllowAlternativeAuthentication = true;

    var authResult = await CrossFingerprint
      .Current.AuthenticateAsync(conf);

    if (authResult.Authenticated) {
        Debug.WriteLine("Success: Authentication succeeded");
        return true;
    } else {
        Debug.WriteLine("Error: Authentication failed");
        return false;
    }
}
```

Authentication method is asynchronous so it does not block the main thread of this application and returns Boolean value if the authentication was successful or not. Only successfully authenticated users can enter the application. First is check if the authentication is possible on the device and then there is request for its usage. Furthermore is configured favor which allows to use alternative form of authentication if there is lack of biometric sensors or they are currently not working. Then the actual authentication takes place and returns result if it was successful or not.

The second service is Data service which provides data to the ViewModel which is responsible for their representation in the View. Data service can obtain data from a database, JavaScript Object Notation (JSON) file or network (for example through API). For the simplicity there are just local example data in this application which live in the device's primary storage only until the application is destroyed or killed by the system.

Every application also needs navigation service. This is already implemented in MvvmCross framework and can be freely used for straightforward navigation between View-

Models, and there is no additional setup needed.

7.1.2 Model

Then there is Model package which houses all the Model classes which represents the data. In this example application there are two Model classes: `Starship(string name, string type, int year, string length, string photo)` and `User(string name, string hometown, string photo)`.

7.1.3 ViewModel

ViewModel is the largest and the most important part of the Core project. There are five groups of ViewModels:

- `!Base`

  There is `BaseViewModel` which inherits from `MvxViewModel`, and its extensions `.Param`, `.ResultAndParam` and `BaseViewModelResult`. It serves as parent for the other ViewModels.

- `Home`

  `HomeViewModel` is the entry point of this application and there is above mentioned biometric authentication service run through the command which is called when the user presses `Log In` button. If the user is successfully authenticated, navigation service navigates user to `TabsRootViewModel`.

```
public IMvxAsyncCommand AuthenticateCommand
{
    get {
        if (this._authenticateCommand == null) {
            this._authenticateCommand = new MvxAsyncCommand(
                async () => {
                    var wasAuthenticated = false;

                    using (UserDialogs.Instance.Loading()) {
                        wasAuthenticated = await this
                          ._authService.Authenticate();
                    }

                    if (wasAuthenticated) {
                        await this._navigationService
                          .Navigate<TabsRootViewModel>();
                    }
                    }
               );
          }
          return this._authenticateCommand;
     }
}
```

- `Tabs`

  `TabsRootViewModel` houses bottom navigation menu of this application. Because of this responsibility, it inherits from special MvvmCross navigation ViewModel called `MvxNavigationViewModel`. This sample of code shows how the bottom menu is configured. It houses list of ViewModels which represent tabs and sets navigation for each of them:

```
this.AllTabs = new List<Type>
{
     typeof(DashboardViewModel),
     typeof(UserViewModel)
};

this.ShowTabsCommand = new MvxAsyncCommand(
     this.InitializeTabs
);
```

```
public Task InitializeTabs()
{
     var tasks = new List<Task>();

     foreach (var tab in this.AllTabs) {
          tasks.Add(this._navigationService.Navigate(tab));
     }
     return Task.WhenAll(tasks);
}
```

- `Dashboard`

  `DashboardViewModel` is the first (and default) tab. There is displayed example of static notification card view, small horizontal list of images (space fighters) and carousel list of cards (spaceships) containing detailed information about them.

- `User`

  `UserViewModel` is the second tab. There are displayed user information, example lorem impsum text and buttons. These buttons can open other application – web browser with their corresponding Uniform Resource Locator (URL) address open.

When the user clicks on the e-mail address it initializes new e-mail in default mail application. These are the methods which handle this functionality:

```
public IMvxCommand OpenEmailCommand
    => new MvxCommand<string>(
        async (email) => await Launcher
            .OpenAsync($"mailto:{email}")
    );

public IMvxCommand OpenUrlCommand
    => new MvxCommand<string>(
        async (url) => await Launcher
            .OpenAsync(url)
    );
```

7.2 UI project

Because this application is made in Xamarin.Forms there is only one UI declaration for both (iOS and Android) platforms. And it is done so in XAML files where each of them has code-behind partial C# class. These code-behind partial classes have extension `xaml.cs` and initially are auto-generated but may need some attention. They usually inherit from `MvxContentPage` with defined corresponding ViewModel and have special annotations. `DashboardPage` and `UserPage` pages need such annotation:

```
[MvxTabbedPagePresentation(
    WrapInNavigationPage = false,
    Title = "Dashboard"
)]
public partial class DashboardPage
    : MvxContentPage<DashboardViewModel>
```

However `TabsRootPage` page needs different one, and inherits from `MvxTabbedPage`:

```
[MvxTabbedPagePresentation(
    TabbedPosition.Root,
    NoHistory = true
)]
public partial class TabsRootPage
    : MvxTabbedPage<TabsRootViewModel>
```

And additionally needs extra setup for Android application:

```
// set tabs to the bottom on Android
this.On<Xamarin.Forms.PlatformConfiguration.Android>()
    .SetToolbarPlacement(ToolbarPlacement.Bottom);
```

```

// disable swiping tabs on Android
this.On<Xamarin.Forms.PlatformConfiguration.Android>()
    .SetIsSwipePagingEnabled(false);
```

Data binding, as it was already explained in Analytical part of this thesis, is performed in XAML files within the UI declaration. This sample of code shows how user's e-mail (`CurrentUser.Email`) is binded into text label, and tap gesture recognizer attached with binded command (`OpenEmailCommand`) together with command parameter (`CurrentUser.Email`) to recognize user's click:

```
<Label Margin="16, 0, 16, 0" FontSize="16" VerticalOptions="End"
  Text="{Binding CurrentUser.Email}" TextDecorations="Underline">

    <Label.GestureRecognizers>

        <TapGestureRecognizer
          Command="{Binding OpenEmailCommand}"
          CommandParameter="{Binding CurrentUser.Email}"/>

    </Label.GestureRecognizers>
</Label>
```

7.3 iOS project

For iOS there must be Xamarin.Forms and PancakeView initialized in the `AppDelegate` class in the `FinishedLaunching()` method:

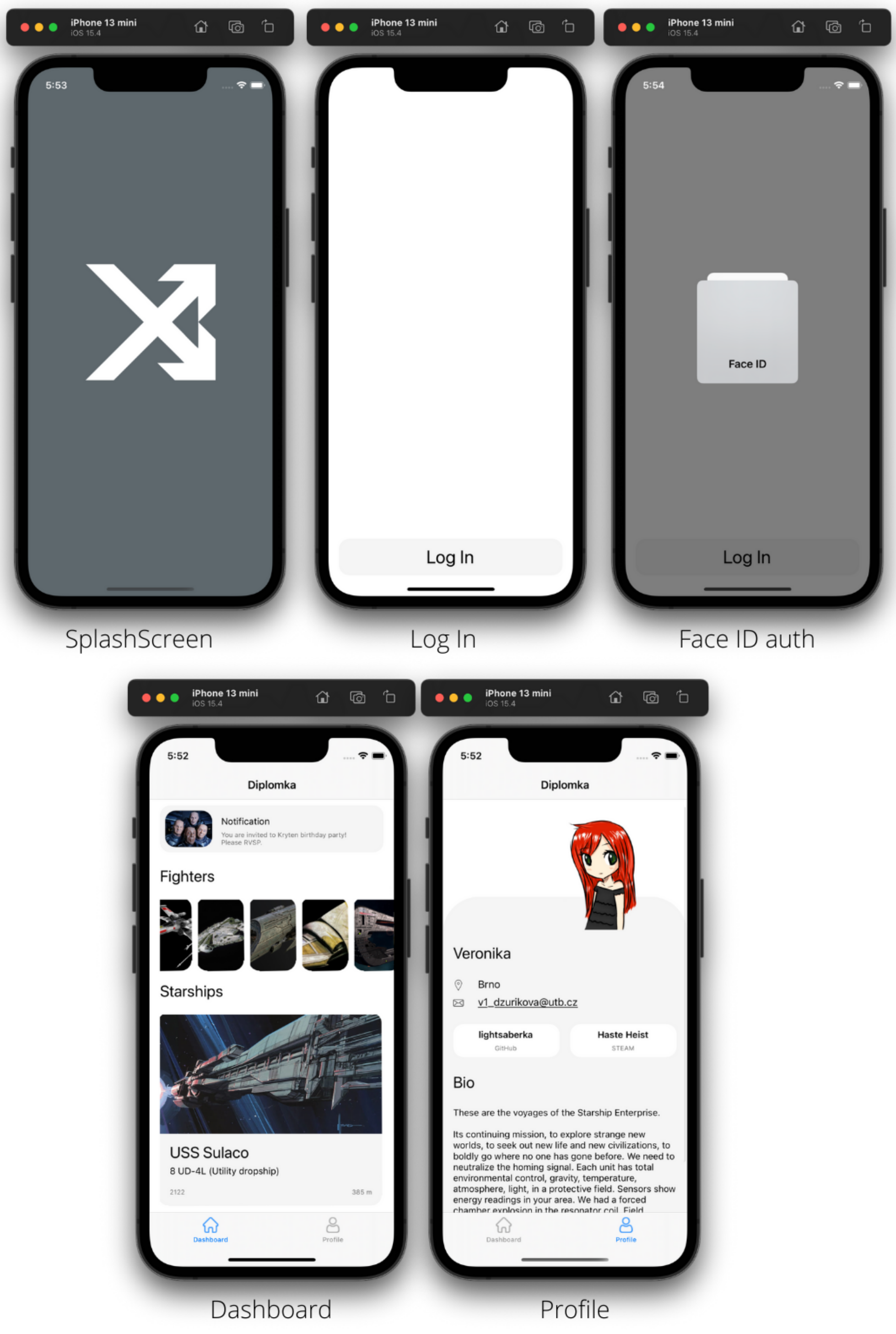
```
global::Xamarin.Forms.Forms.Init();
Xamarin.Forms.PancakeView.iOS.PancakeViewRenderer.Init();

LoadApplication(new App());
```

Then image resources (assets) must be included in `Media.xcassets` package file, and request for the mentioned biometric authentication in `Info.plist` file:

```
<key>NSFaceIDUsageDescription</key>
<string>
    We need Your biometric information for authentication.
</string>
```

Screenshots of the Xamarin.iOS application can be seen in figure 7.1.



You are invited to Kryten birthday party! Please RVSP.

These are the voyages of the Starship Enterprise.

Its continuing mission, to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. We need to neutralize the homing signal. Each unit has total environmental control, gravity, temperature, atmosphere, light, in a protective field. Sensors show energy readings in your area. We had a forced chamber explosion in the resonator coil. Field

Fig. 7.1 Xamarin.Forms iOS application

7.4 Android project

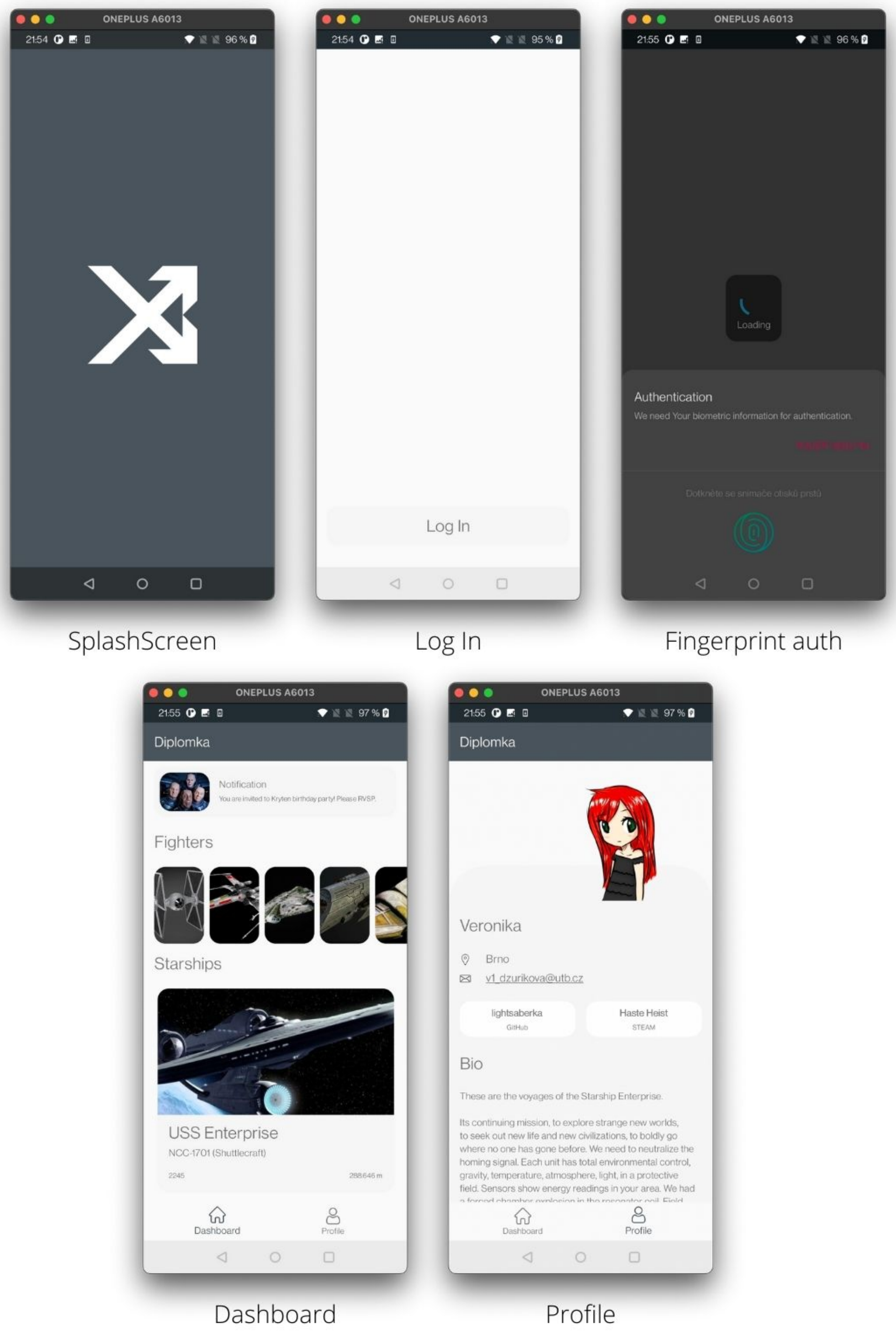
For Android, there is also almost nothing needed because application is made in Xamarin.Forms and biometric authentication is handled in the Core project as well. It just needs to have images included in the resources, and declared request for biometric authentication in `AndroidManifest.xml` file:

```
<uses-permission
    android:name="android.permission.USE_FINGERPRINT" />
<uses-permission
    android:name="android.permission.USE_BIOMETRIC" />
```

Additionally, this biometric authentication must be initialized in `MainActivity` class in `OnCreate() method` together with usage of dialogs:

```
CrossCurrentActivity.Current.Init(this, bundle);
CrossFingerprint.SetCurrentActivityResolver(
    () => CrossCurrentActivity.Current.Activity
);
UserDialogs.Init(this);
```

Screenshots of the Xamarin.Android applications can be seen in figure 7.2.



Authentication

We need Your biometric information for authentication.

Dotkněte se snímače otisků prstů

Notification

You are invited to Kryten birthday party! Please RVSP.

USS Enterprise

NCC-1701 (Shuttlecraft)

2245 288.646 m

Veronika

Brno

v1_dzurikova@utb.cz

lightsaberka GitHub

Haste Heist STEAM

Bio

These are the voyages of the Starship Enterprise.

Its continuing mission, to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. We need to neutralize the homing signal. Each unit has total environmental control, gravity, temperature, atmosphere, light, in a protective field. Sensors show energy readings in your area. We had

Fig. 7.2 Xamarin.Forms Android application

8 Evaluation and Comparison

To decide which technology and frameworks to use for mobile applications development for both iOS and Android platforms, consideration of complexity and robustness of the back-end must take place together with setting of the requirements for the proposed application. It is also important how much business logic will need to be implemented directly in the application or if the application will serve predominantly as a presentation layer for the back-end system.

When there is talk about simple presentational applications, web-based multiplatform technologies such as Electron, Ionic or React native are more than enough, especially when the application does not need to connect to any hardware interfaces through provided platforms API capabilities, or there is not such strong emphasis on design.

Native multiplatform technologies are suitable when the proposed application already requires some sort of business logic, work with hardware or platform's peripherals, but is still in a middle-range size. Such technologies are for example KMM, Xamarin or Flutter. Developers can choose if they define UI for each platform separately to keep the application look truly native, or they choose single conjunct UI definition. Typical middle-range suitable applications may be for travel agencies, shopping centres, or warehouse employees.

Very complex and robust applications where there is big consideration of security and hardware or platform's peripherals should be developed fully native. Typical example are smartbanking applications. This approach may be (at least) twice as expensive, but often necessary. Native UI implementation provides the best UX, and developers have all up-to-date platform libraries and tools at their disposal immediately.

8.1 Comparison based on implementation complexity

In terms of MVVM architecture implementation across the platforms, Xamarin (especially MvvmCross framework for Xamarin) is the most clean and straightforward way. There is clear guideline and process of the project composition and architecture implementation. Also development time is abbreviated thanks to multiplatform approach, additionally when also UI is created in Xamarin.Forms for all the platforms.

MVVM implementation in native iOS has a little looser guidelines, mainly in terms of ViewModels responsibilities, but it is still quite clear and easy to implement. SwiftUI toolkit facilitates the whole process providing its features.

Android implementation is the most difficult because there are dozens of ways how to do it. Every development team or company interprets this architecture differently and there are no any strict rules. Some follow original Microsoft's way – single ViewModel per each View, and some share one ViewModel for more Views. Data binding may be performed in Views (fragment classes in Kotlin) of in XML fragment files. Methods may be called from Views or from ViewModels, and so on.

8.2 Comparison based on implementation time

In consideration of amount of code and time to implement pure MVVM architecture, iOS native implementation is the complete winner, then is Android and the worst in this case is Xamarin because of its greatest abstraction of ViewModels implementation. However it must not be forgotten that currently released new technologies such as Jetpack Compose for Android, KMM for mutliplatform development in Kotlin, or SwiftUI for iOS already consider MVVM architecture in their core design and bring new paradigms – especially declarative and functional programming, which streamlines the development significantly. In older technologies and frameworks such as Xamarin or UWP this architecture could be only achieved with much of code around and abstraction.

8.3 Comparison based on application size

- Native iOS – 3.1 MB
- Native Android – 18.45 MB
- Xamarin.iOS – 65.5 MB
- Xamarin.Android – 31.71 MB

Xamarin applications are much larger in size than native iOS and Android applications because of all the .NET libraries indispensable in Xamarin framework. This may cause slower initialization when the application starts. Xamarin.iOS application's double size than Xamarin.Android is because iOS does not support Just-In-Time Compilation (JIT) so the system has to compile all the .NET Intermediate Language (IL) code into machine code ahead (so-called Ahead-of-Time Compilation (AOT)) [14]. Android supports JIT so Android application's compiled machine code is more compact. Native Swift application for iOS is using SwiftUI framework which significantly optimizes rendering of the UI. Native Android application must be including all the necessary `androidx` libraries and these add in size. Another aspect is how many OS versions is each application targeting, because as the libraries and languages are continuously updating by their maintainers, the application needs to contain them all for each version, and it may cause significant growth in application's size. Additionally when the OS does not support JIT, like iOS.

8.4 Evaluation of advantages and disadvantages

Native applications development has advantage of receiving immediate platform's libraries updates, higher security by not depending on third party libraries and frameworks which also favors in a smaller application size, or better native UI and UX. Disadvantage is double of development time spent for two separate applications which results in higher development cost, and need of developers with programming skills for each platform. iOS, unlike Android, has much cleaner OS architecture and targets less variety of devices so applications for iOS devices are also cleaner and nicer, with less amount of code. Android implementation of APIs for hardware interfaces and libraries often changes with every version, so developers have to implement couple variations of methods to satisfy all supporting OS versions.

Multiplatform development is advancing in less amount of source code because of shared codebase between platforms, so the development process takes less time. This also means fewer developers and with just one required technology skills needed. Also, in consideration of Xamarin.Forms framework, multiplatform development is very easy for beginners or when companies need to come with a prototype of their new product very quickly. However, these multiplatorm applications tend to be larger in size with slower performance, may not provide such native UX, or may cause complications for developers if some new platform's update is not yet available in the chosen multiplatform development technology.

CONCLUSION

Theoretical part of this thesis contains description of current available methods in mobile applications development – native, native multiplatform and web-based multiplatform. Furthermore, it contains explanation of the Model-View-ViewModel (MVVM) architecture pattern development for iOS and Android, and MvvmCross framework for multiplatform development in Xamarin.Forms. In analytical part was performed research of the ways how this architecture pattern can be implemented for each platform. In the practical part example application was designed to be developed for each platform natively and in Xamarin.Forms to demonstrate MVVM implementation in the best possible way. Three applications were developed and then their implementation described in detail – native iOS application in Swift, native Android application in Kotlin and multiplatform application for both these platforms in Xamarin.Forms in C# programming language. Lastly, development of these applications and MVVM implementation were evaluated and compared.

REFERENCES

[1] StatCounter: Mobile Operating System Market Share Worldwide. 2021-11-15.
`https://gs.statcounter.com/os-market-share/mobile/worldwide`

[2] Apple: The Darwin Kernel. 2021-2-8.
`https://github.com/apple/darwin-xnu`

[3] John C. Mitchell: *Concepts in Programming Languages*. Cambridge University Press, 2002, ISBN 9780511804175.

[4] Apple: Actors. 2021-2-8.
`https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md`

[5] Apple: Xcode. 2021-2-8.
`https://developer.apple.com/documentation/xcode`

[6] Apple: Apple Developer Documentation. 2021-2-8.
`https://developer.apple.com/documentation`

[7] Ekramul Hoque: iOS View Controller Life Cycle. 2021-4-23.
`https://medium.com/good-morning-swift/ios-view-controller-life-cycle-2a0f02e74ff5`

[8] Ehab Yosry Amer: iOS Storyboards: Getting Started. 2021-4-23.
`https://www.raywenderlich.com/5055364-ios-storyboards-getting-started`

[9] Joel Joseph: Throwback Tech Thursday: World's First Android Phone Revisited. 2021-1-17.
`https://www.gizmochina.com/2019/06/06/worlds-first-android-phone-htc-dream-g1-revisited`

[10] Google Developers: Android for Developers. 2021-12-10.
`https://developer.android.com`

[11] Steve Pomeroy: The complete Android activity/fragment lifecycle. 2021-1-17.
`https://github.com/xxv/android-lifecycle`

[12] Google Developers: Jetpack Compose Tutorial. 2022-1-16.
`https://developer.android.com/jetpack/compose/tutorial`

[13] Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, ISBN 978-0132350884.

[14] Microsoft: Xamarin documentation. 2021-11-30.
`https://docs.microsoft.com/en-us/xamarin`

[15] Microsoft: What is .NET MAUI? 2021-11-30.
`https://docs.microsoft.com/en-us/dotnet/maui/what-is-maui`

[16] JetBrains: Kotlin for Android. 2022-1-16.
`https://kotlinlang.org/docs/android-overview.html`

[17] Flutter: Flutter documentation. 2022-3-4.
`https://docs.flutter.dev/`

[18] Google: Dart documentation. 2022-3-4.
`https://dart.dev/guides`

[19] Max Lynch: The Easiest Way for Web Developers to Build Mobile Apps. 2022-3-4.
`https://dev.to/ionic/the-easiest-way-for-web-developers-to-build-mobile-apps-1ih8`

[20] InApp: Top 5 Cross-Platform Mobile App Development Frameworks in 2021. 2022-3-4.
`https://inapp-inc.medium.com/top-5-cross-platform-mobile-app-development-frameworks-in-2021-ef9e74ab1b14`

[21] Sophia Martin: 7 Popular Cross-Platform App Development Tools That Will Rule in 2021. 2022-3-4.
`https://medium.datadriveninvestor.com/7-popular-cross-platform-app-development-tools-that-will-rule-in-2020-349c80fb51`

[22] Shelley Vohr: From native to JavaScript in Electron. 2022-3-4.
`https://www.electronjs.org/blog/from-native-to-js`

[23] Meta Platforms: React Native. 2022-3-4.
`https://reactnative.dev`

[24] ITnetwork: Lekce 1 - React Native - Základy React Native. 2022-3-4.
`https://www.itnetwork.cz/javascript/react/native/react-native-zaklady-react-native`

[25] Ionic: Ionic Framework. 2022-3-4.
`https://ionicframework.com`

[26] Josh Smith: Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. 2021-11-30.

https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[27] Gerald Versluis: Using MVVM in your Xamarin.Forms app. 2021-11-30.
https://www.dotnetcurry.com/xamarin/1382/mvvm-in-xamarin-forms

[28] MvvmCross: MvvmCross documentation. 2021-11-30.
https://www.mvvmcross.com/documentation

[29] Plac3Hold3r: MvxScaffolding. 2021-11-30.
https://marketplace.visualstudio.com/items?itemName=Plac3Hold3r.MvxScaffolding

LIST OF ABBREVIATIONS

AOT Ahead-of-Time Compilation. 69

API Application Programming Interface. 12, 20–22, 27, 28, 55, 60, 68, 70

BSD Berkeley Software Distribution. 12, 13

CRUD Create, Read, Update and Delete. 18

CSRG Computer Systems Research Group. 12

CSS Cascading Style Sheets. 27, 28

DLL Dynamic-link Library. 20

DSL Domain-specific Language. 16

GPU Graphics Processing Unit. 14

HTML Hypertext Markup Language. 27, 28

IDE Integrated Development Environment. 14, 17, 20, 24, 26, 28, 44, 47, 51

IL Intermediate Language. 69

IoC Inversion of Control. 31, 33, 59

JIT Just-In-Time Compilation. 26, 69, 70

JSON JavaScript Object Notation. 60

JVM Java Virtual Machine. 25

KMM Kotlin Multiplatform Mobile. 24, 25, 68, 69

LLVM Low Level Virtual Machine. 13, 25

MVVM Model-View-ViewModel. 10, 28–30, 35, 37–39, 42, 68, 69, 71, 77

OHA Open Handset Alliance. 17

ORM Object-Relational Mapping. 29

OS Operating System. 12–14, 17, 20, 26, 27, 35, 70

PDF Portable Document Format. 78

RAM Random-access Memory. 27

SDK Software Development Kit. 12–14, 24, 26

UI User Interface. 12–18, 20–22, 24, 27–32, 35–37, 39, 44, 47, 51, 56, 57, 59, 63, 64, 68, 70, 77

URL Uniform Resource Locator. 44, 62

UWP Universal Windows Platform. 20, 22, 30, 69

UX User Experience. 12, 22, 68, 70

VM Virtual Machine. 26

WPF Windows Presentation Foundation. 29, 30

XAML Extensible Application Markup Language. 22, 39, 63, 64

XML Extensible Markup Language. 15, 18, 22, 38, 51, 52, 56, 69

LIST OF FIGURES

LIST OF APPENDICES