Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj aplikace pro mobilní operační systém iOS

David Aldorf

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. David Aldorf

Informatika

Název práce

Vývoj aplikace pro mobilní operační systém iOS

Název anglicky

App development for iOS mobile operating system

Cíle práce

Diplomová práce je tematicky zaměřena na problematiku vývoje aplikace pro platformu iOS. Hlavním cílem práce je návrh, vyvinutí a otestování reálné aplikace pomocí reaktivního programování.

Dílčí cíle práce jsou:
- představit reaktivní programování
- komparace architektury MVVM s možnými alternativami
- komparace vyvinuté aplikace s možnými alternativními aplikacemi

Metodika

Diplomová práce sestává ze dvou částí – popisu teoretických východisek a praktického řešení vybraného problému.

Metodika zpracování teoretické části práce je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

Praktická část práce spočívá ve vytvoření návrhu architektury, návrhu uživatelského rozhraní a následné implementaci iOS aplikace za použití programovacího jazyka Swift. Proces tvorby aplikace bude popsán a budou shrnuty zkušenosti z využití reaktivního programování.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Mobilní aplikace, iOS, iPhone, Reaktivní programování, Swift, RxSwift, MVVM

Doporučené zdroje informací

LEE, Valentino., Heather. SCHNEIDER a Robbie. SCHELL. Mobile applications: architecture, design, and development [online]. Upper Saddle River, N.J.: Pearson Education, c2004 [cit. 2018-03-19]. ISBN 01-311-7263-8.

MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. London, England: Prentice Hall, 2018. ISBN 0134494164.

PILLET, Florent, Junior BONTOGNALI, Marin TODOROV a Scott GARDENER. RxSwift: Reactive Programming with Swift, Second Edition [online]. raywenderlich.com, c2017 [cit. 2018-03-19]. ISBN 9781942878469.

The Swift Programming Language: Swift 3.1 [online]. Swift Programming Series. Apple Inc., 2014. Dostupné také z: https://itunes.apple.com/cz/book/the-swift-programming-language-swift-3-1/id881256329?l=cs&mt=11

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 28. 3. 2018

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 3. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 30. 03. 2018

Čestné prohlášení

Prohlašuji, že svou diplomovou práci " Vývoj aplikace pro mobilní operační systém iOS " jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2018 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Ing. Jiřímu Brožkovi, Ph.D. za vedení diplomové práce.

Vývoj aplikace pro mobilní operační systém iOS

Abstrakt

Tato diplomová práce se zabývá vývojem mobilní aplikace pro operační systém iOS pomocí reaktivního programování. Vytvoření aplikace bylo provedeno na základě poznatků získaných v teoretické části diplomové práce. Byla provedena komparace jednotlivých softwarových architektur používaných pro vývoj iOS aplikací a na základě bodovací metody byla vybrána architektura MVVM (Model View ViewModel). Dalším krokem bylo vytvoření logického návrhu, který poskytuje rozvržení a funkcionality jednotlivých komponent. Na základě logického návrhu byl vytvořen grafický návrh, využívající nativní komponenty, tak aby bylo vyhověno pokynům „Human Interface Guidelines“ od společnosti Apple.

Pro vývoj aplikace bylo použito vývojové prostředí Xcode od společnosti Apple. K implementaci kódu byl použit moderní objektově orientovaný programovací jazyk Swift, společně s jeho rozšířením RxSwift, určeným pro reaktivní programování. Aplikace poskytuje uživateli možnost přihlášení pomocí sociální sítě Facebooknebo prostřednictvím emailu. Data jsou ukládána v cloudové databázi, díky čemuž má uživatel ke svým datům přístup z více zařízení s operačním systémem iOS. Pro serverovou část aplikace, databázi a přihlašování byla použita platforma Firebase od společnosti Google.

Klíčová slova: iOS, Firebase, Xcode, RxSwift, Swift, mobilní aplikace, databáze, Apple

App development for iOS mobile operating system

Abstract

This thesis deals with the development of a mobile application for iOS operating system. Development of the application was based on theoretical knowledge gained in the theoretic part of the thesis. Individual software architectures used in iOS application's development were compared and MVVM (Model View ViewModel) architecture was chosen based on scoring method. Then a wireframe that contains the layout and describes the functionality of the components used in the application was created. Based on the wireframe a graphic design was created using native components to conform Apple's "Human Interface Guidelines".

Xcode by Apple was used as the Integrated Development Environment. Modern object-oriented programming language Swift with RxSwift framework, designed for reactive programming, was used for the implementation of the code. The application allows user to authenticate via Facebook social network or via email. The data is stored in a cloud database, so the user can access his or her data from multiple iOS devices. The Firebase platform by Google was used for the server-side of the application, including authentication or real-time database.

Keywords: iOS, Firebase, Xcode, RxSwift, Swift, Mobile Application, Database, Apple

Obsah

1 Úvod 13

2 Cíle práce a metodika 14

2.1 Cíl práce 14

2.2 Metodika 14

3 Teoretická východiska 16

3.1 Co je iOS 16

3.2 Historický vývoj iOS 16

3.2.1 iPhone OS 1 16

3.2.2 iPhone OS 2 16

3.2.3 iPhone OS 3 16

3.2.4 iOS 4 17

3.2.5 iOS 5 17

3.2.6 iOS 6 17

3.2.7 iOS 7 17

3.2.8 iOS 8 17

3.2.9 iOS 9 18

3.2.10 iOS 10 18

3.3 Architektura iOS 18

3.3.1 iOS SDK 19

3.3.2 iOS Frameworky 19

3.3.3 Developer library 19

3.4 Cocoa Touch 20

3.4.1 App Extensions 20

3.4.2 HandOff 20

3.4.3 Document Picker 20

3.4.4 AirDrop 20

3.4.5 TextKit 21

3.4.6 UIKit Dynamics 21

3.4.7 Multitasking 21

3.5 Media vrstva 21

3.5.1 Grafické technologie 21

3.5.2 Technologie pro zvuk 22

3.5.3 Technologie pro video 22

3.6 Core services vrstva 22

3.6.1 Služba Peer-to-Peer 22

3.6.2 Uložiště iCloud 23
3.6.3 Ochrana Dat 23
3.6.4 Nákup In-App 23
3.6.5 SQLite 24
3.6.6 Podpora XML 24
3.6.7 Accounts Framework 24
3.6.8 Adressbook Framework 24
3.6.9 Ad support Framework 24
3.6.10 CFNetwork Framework 24
3.6.11 CloudKit Framework 25
3.6.12 CoreData Framework 25
3.6.13 Core Fundation Framework 25
3.6.14 Core Location Framework 25
3.6.15 CoreMedia Framework 26
3.6.16 CoreMotion Framework 26
3.7 Core OS vrstva 26
3.7.1 Accelerate Framework 26
3.7.2 CoreBluetooth Framework 26
3.7.3 External Accessory Framework 27
3.7.4 Local Authentication Framework 27
3.7.5 Network Extension Framework 27
3.7.6 Security Framework 27
3.7.7 Systém 27
3.7.8 64-bit podpora 28
3.8 SW architektura iOS Aplikace 28
3.8.1 Dobrá architektura iOS Aplikace 28
3.8.1.1 Definice dobré architektury 29
3.8.2 SW architektury využívané pro vývoj iOS aplikací 29
3.8.3 MVC – Model View Controller 30
3.8.3.1 Tradiční MVC 30
3.8.3.2 Apple’s MVC (Cocoa MVC) 31
3.8.4 MVP – Model View Presenter 33
3.8.5 MVVM – Model View ViewModel 34
3.8.5.1 Vazby (Bindings) 35
3.8.6 VIPER – View Interactor Presenter Entity Router 36
3.9 Vývoj pro platformu iOS 39
3.9.1 3.14.2 Xcode 40
3.9.2 Objective-C 40

3.9.3 Swift ... 40
3.9.4 Reaktivní programování ... 41
3.9.5 RxSwift ... 43
3.9.5.1 Observable ... 43
3.9.5.2 Subscribing ... 44
3.9.5.3 DisposeBag ... 45
3.9.5.4 Subject ... 47
3.9.5.5 Operátory ... 47

4 Vlastní práce ... 50
4.1 Komparace architektury iOS aplikací ... 50
4.1.1 Rozdělení odpovědností ... 50
4.1.2 Testovatelnost ... 50
4.1.3 Snadné použití a nízké náklady na udržovatelnost ... 50
4.1.4 MVC – Model View Controller ... 51
4.1.5 MVP – Model View Presenter ... 51
4.1.6 MVVM – Model View ViewModel ... 51
4.1.7 VIPER – View Interactor Presenter Entity Router ... 52
4.1.8 Shrnutí komparace architektur ... 53
4.2 Logický návrh ... 54
4.2.1 Příprava ... 54
4.2.2 Drátěný model (Wireframe) ... 56
4.2.2.1 Přihlášení uživatele ... 56
4.2.2.2 Lednice ... 56
4.2.2.3 Přidání a úprava produktu ... 57
4.2.2.4 Recepty ... 57
4.2.2.5 Detail receptu ... 58
4.3 Grafický návrh ... 58
4.4 Implementace ... 59
4.4.1 Seznámení s IDE Xcode ... 59
4.4.2 Návrh architektury ... 59
4.4.3 Dependency injection ... 60
4.4.4 Navigator ... 62
4.4.5 Model ... 63
4.4.5.1 Entity ... 63
4.4.5.2 Databáze Firebase ... 64
4.4.5.3 Přihlášení pomocí Facebook ... 64

4.4.5.4 Přihlášení přes email ... 66
4.4.5.5 Lednice a recepty ... 67
4.4.6 ViewModel ... 69
4.4.6.1 Driver ... 71
4.4.6.2 Error handling ... 71
4.4.7 View ... 71
4.4.7.1 Storyboard ... 72
4.4.7.2 ViewController ... 72
4.5 Testování aplikace ... 73
4.5.1 Unit testy ... 73
4.6 Komparace vyvinuté aplikace s možnými alternativami ... 75
5 Výsledky a diskuse ... 76
5.1 Komparace architektury ... 76
5.2 Logický a grafický návrh aplikace ... 77
5.3 Implementace ... 78
5.4 Testování ... 79
5.5 Zamyšlení a zhodnocení aplikace ... 80
6 Závěr ... 81
7 Seznam použitých zdrojů ... 82
8 Přílohy ... 84

Seznam obrázků

Obrázek 1 - Vrstvy iOS ... 19
Obrázek 2 - Tradiční MVC ... 30
Obrázek 3 – Vize MVC od společnosti Apple ... 31
Obrázek 4 - Cocoa MVC ... 31
Obrázek 5 - MVP ... 33
Obrázek 6 – MVVM ... 35
Obrázek 7 – VIPER ... 37
Obrázek 8 - Marble Diagram ... 42
Obrázek 9 - Observable sekvence pomocí Marble Diagramu ... 43
Obrázek 10 - Diagram rozdělení funkcionalit ... 60

Seznam tabulek

Tabulka 1 - Bodovací tabulka komparace iOS architektur ... 54

Seznam použitých zkratek

MVC – Model View Controller
MVP – Model View Presenter
MVVM – Model View ViewModel
VIPER – View Interactor Presenter Entity Router
DI – Dependency Injection

1 Úvod

Časy mobilních telefonů, které poskytovaly pouze možnost volání a psaní SMS zpráv, jsou pryč. Nároky lidí na techniku se postupně zvyšují a společně s nimi se vyvíjí i mobilní telefony. V dnešní době jsou mobilní telefony na velkém technologickém vzestupu a stávají se nedílnou součástí života velké části populace. Ať si to připouštíme nebo ne, jsou pro nás velkým usnadněním běžného života.

Aplikace v chytrých telefonech nám pomohou v komunikaci s lidmi na delší vzdálenost v podstatě kdekoliv, kde máme možnost mobilního internetu či wifi. Samozřejmě je mnoho aplikací a funkcí chytrých telefonů, které se využívají v offline módu, tzn. bez připojení na internet. Velkou výhodou chytrých telefonů jsou jejich kompaktní rozměry, při kterých si v této době ponechávají výpočetní výkon téměř na úrovni stolních počítačů či notebooků. Další využití chytrých telefonů se nachází ve využití fotoaparátukterý v této době ve značné míře nahradil klasické fotoaparáty běžných uživatelů a umožňuje pořídit kvalitní fotografii bez toho, aby bylo potřeba s sebou nosit další zařízení.

Vývoji mobilních aplikací pro platformu iOS se věnuji již od posledního ročníku bakalářského studia, kdy jsem si ve své bakalářské práci dal za úkol seznámit se s vývojem mobilních aplikací a vytvořit aplikaci, která by mně i dalším lidem se zájmem o zdravou stravu pomohla. V diplomové práci bych rád dále poukázal na výhody a nevýhody reaktivního programovaní iOS aplikací.

Jelikož žijeme ve velice konzumní společnosti, kde se veliké množství jídla vyhodí či znehodnotí, rád bych pomocí své aplikace navedl uživatele, aby jídlo, co mají doma, efektivněji využili.

Cílem této diplomové práce je tedy vytvoření aplikace, která by pomohla navrhnout uživateli, jaké jídlo by si mohl připravit z potravin, které má aktuálně doma. Aplikace také upozorní na potraviny, kterým v brzké době vyprší expirační doba.

2 Cíle práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na problematiku vývoje aplikace pro platformu iOS. Hlavním cílem práce je návrh, vyvinutí a otestování reálné aplikace pomocí reaktivního programování. Dílčími cíli práce jsou: Představení reaktivního programování, komparace architektury MVVM s možnými alternativami a komparace vyvinuté aplikace s možnými alternativami.

2.2 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Z poznatků založených na teoretických východiscích diplomové práce byla provedena komparace softwarových architektur používaných pro vývoj iOS aplikací. Komparace softwarových architektur byla provedena na základě ohodnocení zvolených kritérií u jednotlivých architektur, která mají přímý vliv na vývoj aplikace samotné.

Při sestavení logického návrhu aplikace byla použita metoda User experience, ve které byly vytvořeny drátěné modely aplikace pro každou její stranu. Tyto modely byly otestovány za pomoci pěti zástupců cílové skupiny. Na základě poznatků získaných z testování byl vytvořen konečný logický návrh, ze kterého byl následně vytvořen návrh grafický. Jeho tvorba byla řízena souborem pravidel „Human Interface Guidelines“ pro tvorbu mobilní aplikace od společnosti Apple, s důrazem na použití nativních komponent iOS aplikací. Pro vytvoření uživatelského rozhraní bylo použito vývojové prostředí Xcode, pomocí metod UIKit bylo vytvořeno na základě grafického návrhu.

Pro tvorbu uživatelského rozhraní byla využita funkcionalita Storyboard vývojového prostředí Xcode. Implementace kódu pro uživatelské rozhraní je rozdělena do instancí ViewControlleru aplikace v prezentační vrstvě architektury View. Pro první stranu aplikace určenou pro přihlašování a registraci uživatelů do databáze byly využity frameworky Firebase a Facebook SDK. Pro přístup k databázi Firebase byly použity metodické postupy založené na využití Firebase framework, který zajišťuje zapsání a čtení obsahu v databázi. Dále byla vytvořena aplikace pro přístup k Facebooku SDK na stránce developers.facebook.com, pomocí které je zajištěný přístup aplikace k API Facebooku. V

kódu aplikace byla vytvořena funkcionalita pro přihlášení uživatele a jeho zapsání do databáze. Další obrazovky aplikace využívají především práci s databází Firebase, jedná se o ukládání, úpravu, odstranění a získání dat. Pro skenování čárového kódu byl využit framework BarcodeScanner, který využívá nativní funkce pro skenování barometrických kódů v knihovně iOS. Na základě syntézy teoretických poznatků a výsledků praktické části byly formulovány závěry bakalářské práce.

3 Teoretická východiska

3.1 Co je iOS

IOS je operační systém, který je základem mobilních zařízení společnosti Apple. Mezi zařízení podporující iOS patří iPad, iPhone a iPod. Operační systém spravuje Hardware zařízení a poskytuje technologie pro implementaci nativních aplikací. IOS je mobilní verzí operačního systému OS X určené pro desktopové zařízení iMac a Macbook společnosti Apple. Operační systém iOS je na zařízení dodáván společně s různými systémovými aplikacemi, jako je například aplikace telefon, mail a safari, které poskytují standartní systémové služby pro uživatele. [1] [2]

3.2 Historický vývoj iOS

3.2.1 iPhone OS 1

V roce 2007 byla zveřejněna první verze operačního systému od společnosti Apple Inc. Operační systém byl v tu dobu určen pouze pro první generaci mobilního telefonu iPhone. Do roku 2008, kdy byl vydán iPhone software development kit, nebyl oficiálně pojmenován. Od tohoto roku nesl operační systém název iPhone OS. V první verzi nebylo možné instalovat aplikace, ale i přesto společnost Apple Inc. udala jasný směr v uživatelském prostředí, kterým se výrobci mobilních zařízení v dalších letech dali. [4]

3.2.2 iPhone OS 2

V roce 2008 společně s vydáním iPhone 3G, přichází druhá generace iPhone OS. Tato
generace přichází s revolucí v aplikacích a tím se stal konkrétně App Store. Uživatelé již nebyli nuceni instalovat pomocí svého počítače a hledat je z různých internetových zdrojů. Protože Apple vymyslel centrální místo, odkud byly aplikace dostupné všem uživatelům přímo z jejich mobilních zařízení. [4]

3.2.3 iPhone OS 3

O rok později přichází společnost s další verzí iPhone OS 3. Ve třetí verzi jsou přidány pouze některé funkce, navíc např. centrum notifikací, funkce pro kopírování a vkládání textu a aplikaci Find my iPhone (bezpečností aplikaci pro nalezení ztraceného či

odcizeného iPhone). [4]

3.2.4 **iOS 4**

V roce 2010 byla zveřejněna 4. generace operačního systému, poprvé nesoucí název iOS. Tato verze jako první nepodporuje všechna zařízení, konkrétně iPhone první generace. Nová verze obsahuje multitasking, tj. práce s více spuštěnými aplikacemi najednou, provozování videohovorů mezi zařízeními iOS, za pomocí aplikace FaceTime, vytváření složek s aplikacemi, vzdálené přehrávání multimédií AirPlay a vzdálený tisk pomocí AirPrint. [4]

3.2.5 **iOS 5**

V roce 2011 byla zveřejněna 5. generace operačního systému iOS, která s sebou přináší hlasového klienta Siri - aplikaci rozpoznávající dotazy, na základě nichž vyhledává požadované informace. Dále poskytuje službu iMessage, textová komunikace mezi zařízeními iOS a cloudové uložiště iCloud, pro zálohování zařízení. [4]

3.2.6 **iOS 6**

V roce 2012 společnost Apple uvedla 6. generaci iOS. Přináší novou aplikaci pro mapy, která umožňuje hlasovou navigaci na pozadí, předinstalovanou sociální síť Facebook a funkci pro sdílení fotografií. [4]

3.2.7 **iOS 7**

V roce 2013 byla vydána 7. generace iOS. Největší devizou byl vzhled systému, ve kterém jsou použity bílé odstíny barev, průhledné efekty, nový vzhled ikon aplikací a ovládací centrum. Rozšíření se týká také notifikačního centra, které místo jedné záložky obsahuje tři záložky a aplikaci AirDrop, která slouží pro sdílení obsahu mezi iOS zařízeními. [4]

3.2.8 **iOS 8**

V roce 2014 Apple představil iOS 8. generace. Zařízení podporující iOS 8 jsou iPhone 4S, iPad 2, iPod Touch (5. generace) a novější. IOS 8 přinesla velké změny převážně pro vývojáře aplikací. Bylo zpřístupněno velké množství API a byl představen programovací jazyk Swift. V nové verzi systému je použito provázání aplikací. Kontinuita

zařízení, tj. propojení zařízení iOS, OSX a jejich funkcí. Jsou přidány také interaktivní notifikace a další rozšíření aplikací iOS. [4]

3.2.9 **iOS 9**

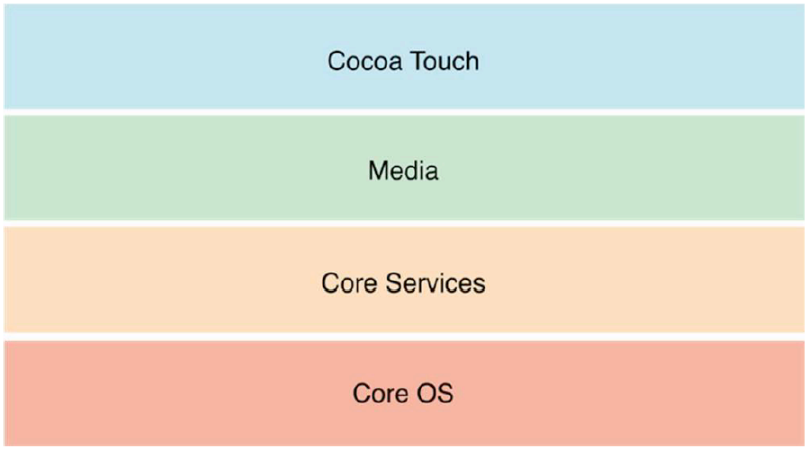
V roce 2015 Apple představil iOS 9. Jedná se o aktuální verzi systému od společnosti Apple. iOS 9 přináší plnohodnotný multitasking, tj. SlideOver pro iPady, uložiště iCloud Drive, aplikace pro nahrávání dokumentů, videí, fotografií a jejich sdílení a je upravena aplikace pro poznámky. Inovace se týkají dále aplikací News, Siri a byl přidán úsporný režim pro delší výdrž baterie zařízení. [4]

3.2.10 **iOS 10**

V iOS 10 se objevily další velké změny, například chatovací aplikace iMessage získala vylepšení v podobě Samolepek. Dále se App Store stal plnohodnotnou platformou a plně se otevřel vývojářům třetích stran. Dokonce i Siri dokáže komunikovat s aplikacemi, které nejsou od Applu. Světlo světa spatřila i aplikace Home pro ovládání inteligentní domácnosti. A viditelných změn se dočkala oblast notifikací, která nyní zobrazuje i přímo fotky či videa. [4]

3.3 **Architektura iOS**

iOS je odlehčenou verzí operačního systému OS X, který je používaný u desktopových a přenosných počítačů MAC. IOS je operační systém UNIXového typu určený pro mobilní zařízení, proto není potřeba, aby obsahoval veškerou funkcionalitu OS X. Na druhou stranu, je zde oproti OS X přidána podpora dotykového ovládání. Operační systém iOS se dělí na čtyři základní vrstvy Cocoa Touch, Media, Core Services a Core OS. Každá z vrstev obsahuje balíček frameworků a rozhraní, pomocí kterých mohou být při vývoji aplikace použity. [2]



Obrázek 1 - Vrstvy iOS
Zdroj: [2]

3.3.1 iOS SDK

IOS Software Development Kit (dále jen iOS SDK) obsahuje nástroje a rozhraní potřebné pro vývoj, instalaci, spuštění a testování nativní aplikace, které jsou zprostředkovány uživateli na domovské obrazovce zařízení. Nativní aplikace jsou vytvořeny za použití iOS systémových frameworků a programovacího jazyku Objective-C, aplikace jsou spustitelné přímo na iOS. Nativní aplikace jsou na rozdíl od aplikací webových nainstalovány přímo na zařízení, a proto je jejich obsah dostupný i v režimu Offline, tzn. bez přístupu k internetu. Nativní aplikace jsou umístěny vedle dalších systémových aplikací a jejich uživatelská data jsou synchronizována do desktopového zařízení pomocí iTunes. [2]

3.3.2 iOS Frameworky

Frameworky jsou speciální balíčky pomocí kterých Apple, poskytuje většinu svých systémových rozhraní. Framework je adresář, který obsahuje dynamicky sdílené knihovny a zdroje, např. hlavičkové soubory, obrázky a pomocné aplikace, potřebné k podpoře knihovny. [2] [5]

3.3.3 Developer library

Developer library iOS je důležitým zdrojem informací k použití během vývoje aplikací. Obsahuje reference všech systémových API, programovací příručky, technické poznámky, ukázkový kód a mnoho dalších zdrojů nabízejících tipy a rady při vytváření aplikace přímo od vývojářů společnosti Apple. [2] [5]

3.4 Cocoa Touch

Vrstva Cocoa Touch je Framework (knihovna) uživatelského rozhraní od společnosti Apple pro platformu iOS. Je založen na systému OS X jako mobilní verze Cocoa. Uživatelské prostředí, využívá vysokých programovacích jazyků pro nativní vývoj tzn. Objective-C a nově také Swift. Použití Cocoa Touch frameworku v první řadě přináší zjednodušení implementovaného kódu. Pomocí uživatelského rozhraní je možné přistupovat k funkcím iOS jako je např. Multitasking, jenž zabezpečuje práci a přepínání s dvěma a více spuštěných aplikací. [2] [5]

3.4.1 App Extensions

Operační systém iOS poskytuje rozšíření vybrané oblasti systému s použitím předem nakonfigurovaných App Extensions. App Extensions podporují sdílení obsahu se sociálními sítěmi např. Facebook a Twitter, provést jednoduchý úkol s aktuálním obsahem, poskytnout rychlou aktualizaci a zobrazení úkolu v oznamovacím centru zařízení, provádět úpravu fotografií a videa v aplikaci Fotky a poskytují místo pro ukládání dokumentů, pro které je vytvořen přístup i z ostatních aplikací. [2] [5]

3.4.2 HandOff

HandOff je funkce v operačních systémech OS X a iOS. Umožňuje uživatelům zahájit činnost v aplikaci na jednom zařízení od společnosti Apple a dokončit jej na druhém. Při předávání aktivit se přiřazují aplikacím API v oblasti Foundation. Aktivita musí být reprezentována v aplikaci jako objekt uživatelské činnosti (User aktivity object). [2] [5]

3.4.3 Document Picker

Document Picker, neboli document picker controller, poskytuje uživatelům přístup k souborům mimo sandbox aplikace. Jedná se o sdílení dokumentů mezi aplikacemi. Vývojáři třetích stran mohou poskytnout dokumenty ke sdílení pomocí Storage Provider. [2] [5]

3.4.4 AirDrop

AirDrop umožňuje uživatelům sdílet fotografie, dokumenty a další data s blízkými zařízeními iOS. [2] [5]

3.4.5 TextKit

TextKit je sada tříd pro manipulaci s textem a typografií. Požití TextKit poskytne aplikaci rozvržení textu do odstavců, sloupců, nebo obtékání grafického objektu textem. TextKit je integrován se všemi textovými ovládacími prvky UIKit. [2] [5]

3.4.6 UIKit Dynamics

UIKit Dynamics určí aplikaci dynamické chování pro objekty, které jsou podporovány protokolem UIDynamicItem, tyto objekty se nazývají dynamické položky. Dynamické položky navodí uživateli příjemnější požitek z používání aplikace. Objekty se stanou dynamicky aktivní, po přidání UIDynamicAnimator třídy. [2] [5]

3.4.7 Multitasking

Multitasking poskytuje přepínání mezi aplikacemi, při kterém dochází ke zneaktivnění první aplikace a následně zaktivnění aplikace druhé. Při zneaktivnění, aplikaci je umožněno požádat o konečné množství času ke zpracování úkolu, např. přehrávání hudby. [2] [5]

3.5 Media vrstva

Umožňuje vytvářet graficky a zvukově propracované aplikace a také plynulé přehrávání animací, zvuků a videí. [2] [5]

3.5.1 Grafické technologie

Ve většině případů lze využít grafické rozhraní definované samotným systémem. Pro ostatní případy je možné použít následující technologie.

- Core graphics (Quartz) - 2D vektory a renderované obrázky
- Core animation - Pokročilé animace obrázků
- OpenGL ES - HW akcelerované vykreslování 2D/3D obrázků
- Core Text - Sofistikovaný engine pro vykreslování textu
- Image I/O - Čtení a zápis grafických formátů
- The Assets library framework - Přístup k obrázkové knihovně uživatele

[2] [5]

3.5.2 Technologie pro zvuk

Umožňuje přehrávání videozáznamů a používání vibrací. Podporuje následující frameworky.

- The Media Player framework - Umožňuje přístup k iTunes knihovně
- AV Foundation - Rozhraní pro správu a přehrávání záznamu zvuku
- OpenAL - Multiplatformní pozicování zvuku (3D)
- Core Audio framework - Poskytuje rozhraní pro přehrávání a záznam zvuku

[2] [5]

3.5.3 Technologie pro video

Umožňuje přehrávání a záznam videozáznamu a pracovat s nimi v aplikacích. Podporovány jsou následující frameworky.

- Media Player framework - Umožňuje přehrávání videí
- AV foundation - Rozhraní pro záznam a přehrávání videa
- Core media - Popisuje nízkoúrovňové typy a rozhraní používané ve vysokoúrovňových frameworcích

[2] [5]

3.6 Core services vrstva

Obsahuje základní systémové služby pro aplikace. Klíčové jsou služby Core Foundation a Foundation frameworks, které definují základní typy pro používání všech aplikací. Tato vrstva podporuje funkce např. lokace, iCloud, sociální sítě a síťové připojení.

[6]

3.6.1 Služba Peer-to-Peer

Rámec Multipeer Conectivity poskytuje peer-to-peer připojení pomocí technologie bluetooth. Zařízení umožňuje zahájení komunikace po síti peer-to-peer s dalšími blízkými zařízeními. Multipeer Conectivity se používá především ve hrách a podobných typech aplikací. [6]

3.6.2 **Uložiště iCloud**

Umožňuje aplikaci zápis dat a uživatelských dokumentů do centrálního umístění. Do uložiště iCloud uživatelé mají přístup ze všech počítačů a zařízení iOS. Úprava a zobrazení souborů je poskytnuta bez nutnosti synchronizace nebo samostatného přenosu souborů.

iCloud document storage se používá pro ukládání a zobrazení dokumentů a dat na iCloud účtu uživatele. iCloud key-value data storage tato funkce slouží ke sdílení dat mezi instancemi aplikace.

CloudKit storage, tato funkce se používá pro vytvoření veřejně sdíleného obsahu, nebo při řízení přenosu dat vývojáři.

[6]

3.11.2 Block Objects

Je konstrukt programovacího jazyka C, tzv. C-Level, který je možné začlenit do kódu společně s C a Objective-C. Blokové objekty jsou nejčastěji používány pro nahrazení delegátů a delegovacích metod, jako náhrada za rekurzivní funkce a pro usnadnění provádění úlohy pro všechny body v kolekci.

[6]

3.6.3 **Ochrana Dat**

Umožňuje aplikacím práci s citlivými daty uživatele při použití integrovaného šifrování. Pokud je zařízení zamknuté, obsah souborů je nepřístupný. Při odemčení zařízení je vytvořen dešifrovací klíč, který přístup k aplikacím a souborům zpřístupní. Implementace ochrany dat vyžaduje vytvoření a následnou správu chráněných dat.

[6]

3.6.4 **Nákup In-App**

Poskytuje aplikace možnost koupi obsahu uvnitř aplikace, prostřednictvím AppStore nebo iTunes. Tato funkce je implementována pomocí frameworku StoreKit, ten poskytuje infrastrukturu potřebnou pro zpracování finančních transakcí pomocí účtu iTunes.

[6]

3.6.5 SQLite

Zprostředkovává vložení lehké SQL databáze do aplikace bez použití samostatného vzdáleného databázového serveru. Knihovna je navržena a optimalizována pro rychlý přístup k záznamům databáze.
[6]

3.6.6 Podpora XML

Foundation Framework poskytuje NSXML parser, třída pro načítání prvků ze souboru XML. V iOS_SDK jsou umístěny knihovny pro zápis dat pomocí XML souboru a jeho transformace do jazyka HTML.
[6]

3.6.7 Accounts Framework

Poskytuje single sign-on model pro uživatelské účty. Single sign-on zlepšuje uživatelský dojem z používání aplikace, eliminuje potřebu přístupu uživatele k více účtům. Tento rámec lze použít společně s rámcem Social Framework.
[6]

3.6.8 Adressbook Framework

Umožňuje aplikace přístup ke kontaktům uživatele, jejich pro jejich zobrazení, ale i úpravu. Přístup ke kontaktům vyžaduje povolení ze strany uživatele. Proto by měla být aplikace připravena pro zamítnutí uživatele k přístupu. V souboru info.plist musí být uvedeno, z jakého důvodu aplikace o přístup ke kontaktům žádá.
[6]

3.6.9 Ad support Framework

Rámec Ad support umožňuje přístup k identifikátoru, který aplikace využívají k reklamním účelům. Rámec kontroluje odhlášení uživatele ze sledování reklamy.
[6]

3.6.10 CFNetwork Framework

Je sada výkonných rozhraní, které využívají objektově orientovanou abstrakci pro práci se síťovými protokoly a kteréjsou založené na bázi programovacího jazyka C. Tento

rámec se používá pro zjednodušení komunikace s FTP a http servery, nebo jako řešení DNS hostitele. Umožňuje použití BSD socketů, vytvoření šifrovaného připojení pomocí SSL a TSL. Publikovat a procházet služby Bonjour.
[6]

3.6.11 **CloudKit Framework**

Poskytuje kanál pro přesun dat mezi aplikací iCloud a aplikací. Rámec je možné použít pro správu všech typů dat. Aplikace kde je přímo použit CloudKit mohou ukládat data do složky sdílené všemi uživateli aplikace. Tato veřejná uložiště jsou zpřístupněna i na zařízeních bez registrovaného účtu iCloud.
[6]

3.6.12 **CoreData Framework**

Rámec pro správu datového modelu ModelViewController. Rámec CoreData je určen pro použití aplikací, ve kterých je model dat vysoce strukturovaný. Namísto definování datových struktur implementací kódu, je možné použít nástroje v IDE Xcode a vytvořit schéma datového modelu graficky.
Ukládání objektových dat do databáze SQLite , správa funkce dopředu/zpět nad rámec základní editace textu, podpora šíření změn při zachování konzistentního vztahu mezi objekty.
[6]

3.6.13 **Core Fundation Framework**

Podpora následujících typů pole, sady, řady, uzly, String řetězce, Datum a čas, Raw data block, preference, URL a stream, threads a run loops, komunikace Port a socket. Rámec je úzce spjat s Foundation Framework.
[6]

3.6.14 **Core Location Framework**

Poskytuje aplikaci informace o umístění. V případě informací o poloze rámec využívá GPS, wifi a mobilní připojení k nalezení zeměpisné šířky a délky umístění zařízení. Tato technologie se využívá v aplikacích. Další funkce jsou přístup ke kompasu iOS zařízení obsahující magnetometr, podpora oblasti monitorování založené na zeměpisné

poloze, podpora využití mobilní sítě s nízkým výkonem a spolupráce s MapKit za účelem zlepšení kvality lokalizačních údajů.
[6]

3.6.15 **CoreMedia Framework**

Stanovuje nízkoúrovňové typy médií používané ve AVFoundation Framework.
[6]

3.6.16 **CoreMotion Framework**

Poskytuje jedinou sadu rozhraní pro přístup k dispozici všechna data o pohybu dostupná na
zařízení, pomocí akcelerometru za použití nové sady Block-based rozhraní. U zařízení s vestavěným Gyroskopem, načítá gyroskopické data. Rámec nachází časté použití ve hrách a v dalších aplikací používající pohyb, např. sportovní aplikace.
[6]

3.7 Core OS vrstva

Poskytuje nízkoúrovňové funkce ostatním technologiím, které jsou na ní postaveny.
[7]

3.7.1 **Accelerate Framework**

Accelerate Framework obsahuje rozhraní pro zpracování digitálního signálu DSP, lineární algebry a výpočty pro zpracování obrazu. Výhodou rámce je optimalizace pro všechna iOS zařízení.
[7]

3.7.2 **CoreBluetooth Framework**

CoreBluetooth Framework zajišťuje aplikaci interakci s nízkoenergetickými bluetooth zařízeními. Využívá se pro hledání ostatních bluetooth zařízení a jejich připojení, či odpojení, předávání informací pomocí iBeacon mezi zařízeními iOS a pro získání informací o dostupnosti periferních bluetooth zařízení.
[7]

3.7.3 External Accessory Framework

External Accessory Framework poskytuje podporu pro komunikaci s hardwarovými doplňky připojených k iOS zařízení. Příslušenství je možné připojit pomocí dokovacího konektoru nebo bezdrátově pomocí bluetooth.
[7]

3.7.4 Local Authentication Framework

Local Authentication Framework umožňuje použití Touch ID k ověření uživatele. Aplikace mohou vyžadovat ověření uživatele pro přístup k celému obsahu aplikace nebo k jeho části. Uživatel musí být řádně informován o důvodu použití ověření. Pokud je ověření úspěšné aplikace povolí přístup.
[7]

3.7.5 Network Extension Framework

Network Extension Framework poskytuje podporu pro konfiguraci a řízení virtuální privátní sítě, tzv. VPN. Aplikace může VPN spustit manuálně nebo může vydávat pravidla pro vyžádání k jejímu spuštění.
[7]

3.7.6 Security Framework

Security Framework poskytuje rozhraní pro správu certifikátů, veřejných a soukromích klíčů a zásad důvěryhodnosti. Podporuje generování náhodných šifrovaných čísel. Také podporuje uložiště certifikátů a kryptografických klíčů v klíčovém řetězci, využívané pro zabezpečení citlivých uživatelských dat v aplikaci. Umožňuje sdílení klíčových řetězců mezi více aplikacemi, díky tomu není uživatel nucen do všech aplikací zvlášť zadávat své heslo, vyžaduje konfiguraci aplikací v IDE Xcode pomocí zásad důvěryhodnosti.
[7]

3.7.7 Systém

Systém zahrnuje kernelové prostředí, ovladače a nízkoúrovňové UNIX rozhraní operačního systému. Kernel je zodpovědný za každý aspekt operačního systému. Systém spravuje virtuální paměť, souborový systém a komunikaci uvnitř procesů. Ovladače

poskytují rozhraní mezi dostupným hardwarem systémovými frameworky.
Z bezpečnostních důvodů je přístup ke kernelu a ovladačům omezen sadou frameworků a aplikacemi. Využití je v přístupu k souborovému systému, ke standardu I/O, k alokaci paměti a k matematickým výpočtům.
[7]

3.7.8 **64-bit podpora**

Operační systém iOS byl původně navržen pro podporu binárních souborů ve 32-Bitové
architektuře. Od operačního systému iOS 7 byla představena kompilace a ladění binárních souborů v 64-Bitové architektuře. Všechny systémové knihovny a frameworky jsou 64-bit ready, což znamená, že mohou být použity jak v 32-bitové, tak v 64-Bitové architektuře.
[7]

3.8 SW architektura iOS Aplikace

3.8.1 **Dobrá architektura iOS Aplikace**

Pokud vývojář nebude dbát na vybrání architektury, tak se jednoho dne oprava chyb v aplikaci a vůbec jejich nalezení, či přidání nových funkcionalit stane skoro nemožným. Pokud funkcionality aplikace budou záviset například na jediné třídě, kód v ní bude obrovský časem nejspíše i neuspořádaným a hlavně velice málo přehledným. Pokud jde o příklad iOS aplikace případy špatné architektury mohou být tyto:

- Tato třída je podtřída UIViewController
- Data jsou uloženy a zpracovány přímo v této třídě
- UIView nemá skoro žádnou funkcionalitu
- Modelová struktura neobsahuje žádnou logiku
- Unit testy nic nepokrývají

To se může stát i přes skutečnost, že používáte pokyny společnosti Apple a implementujete architektonický vzor MVC (Model View Controller) společnosti Apple, který používá ve veškerých ukázkových kódech a dokumentaci. Tato architektura má svá úskalí, na která v této diplomové práci poukážu v dalších částech této kapitoly.
[8][9]

3.8.1.1 Definice dobré architektury

- Vyvážené rozdělení odpovědností mezi subjekty, které mají přesně definované role
- Testovatelnost obvykle pochází již z první funkcionality
- Snadné použití a nízké náklady na udržovatelnost

[8][9]

3.8.2 **SW architektury využívané pro vývoj iOS aplikací**

V dnešní době máme mnoho rozlišných možností použití SW architektury (dále jen architektury) pro vyvíjenou iOS aplikaci. Je potřeba si stanovit nejdříve co si lze představit pod pojmem „Dobrá architektura aplikace“.

Dobrá architektura pro SW by měla mít tyto body:

- Každý objekt má jasně danou roli
- Schopnost snadno sledovat tok dat
- Nezávisí na žádném konkrétním frameworku
- Omezené a jasně dané závislosti
- Flexibilita díky jednoduchosti a nikoliv složité abstrakci

V dnešní době máme mnoho možností, pokud jde o architektonické návrhy:

- MVC – Model View Controller
- MVP – Model View Presenter
- MVVM – Model View ViewModel
- VIPER – View Interactor Presenter Entity Router

První tři z nich předpokládají uvedení entit aplikace do jedné ze tří kategorií:

- Model - Je zodpovědný za doménové data nebo vrstvy přistupující k manipulaci s daty. Tzn. Doménové data např. „Person“ nebo třídy poskytující a upravující data „PersonDataProvider“
- View – Je zodpovědný za prezentační vrstvu, neboli grafické uživatelské rozhraní aplikace (GUI), pro příklad iOS aplikací se lze bavit o všechny třídy s předponou „UI“

- Controller/Presenter/ViewModel - Prostředník, nebo pojidlo mezi modelovou a prezentační vrstvou. Obecně je zodpovědný za změnu modelové vrstvy reakcí na uživatelské změny na prezentační vrstvě a zároveň aktualizuje informace prezentační vrstvy na základě vrstvy modelové.

Rozdělení jednotlivých entit nám umožňuje:

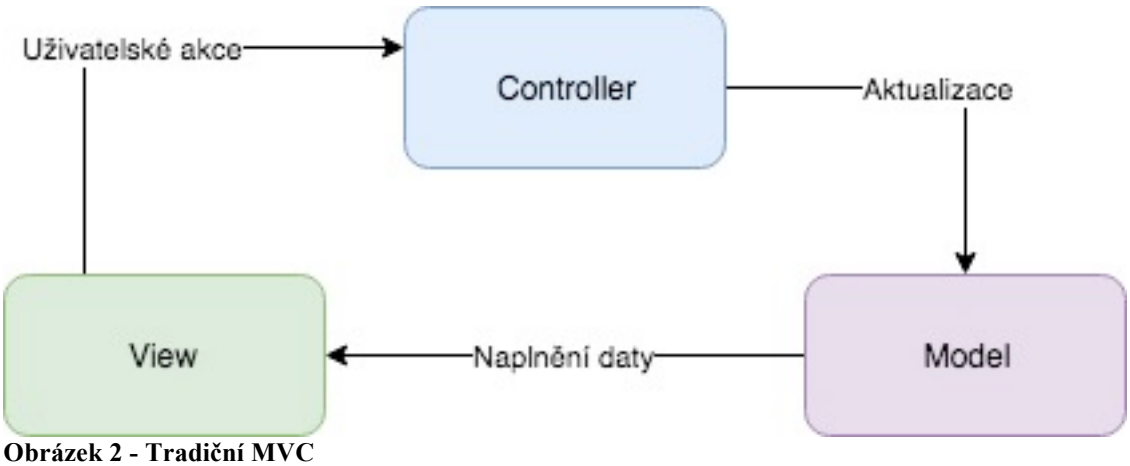
- Lépe jim porozumět
- Znovu použitelnost (nejčastěji aplikovatelné na „View“ a „Model“)
- Nezávislá testovatelnost

[8][9]

3.8.3 MVC – Model View Controller

Architektura MVC dělí aplikaci na 3 logické části tak, aby je šlo upravovat samostatně a dopad změn byl na ostatní části co nejmenší. Tyto tři části jsou Model, View a Controller. Model reprezentuje data a business logiku aplikace, View zobrazuje uživatelské rozhraní a Controller má na starosti tok událostí v aplikaci a obecně aplikační logiku. [8][9]

3.8.3.1 Tradiční MVC

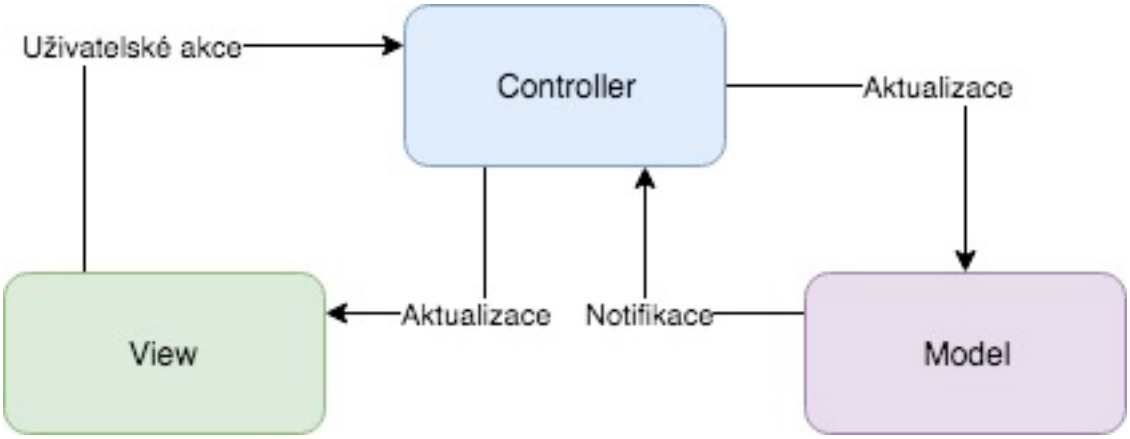


Obrázek 2 - Tradiční MVC
[Vlastní zpracování]

V tomto případě je prezentační vrstva view bezstavová. Je překreslená pomocí Controlleru vždy, když je modelová vrstva změněna. Např. webová stránka je kompletně překreslena, pokud uživatel změní aktuální záložku a je přesměrován na stránku s jiným obsahem.

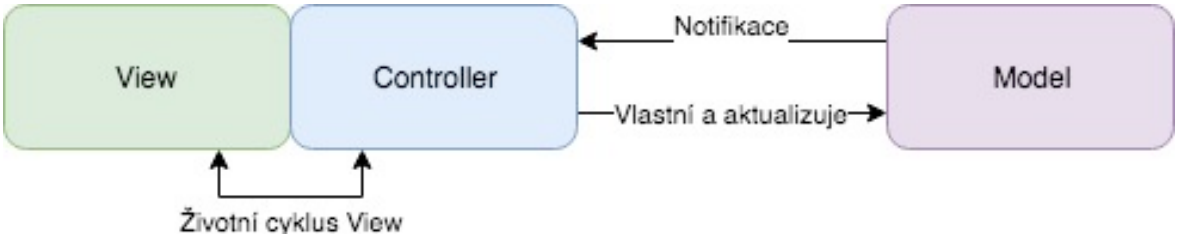
Přestože je možné tradiční MVC architekturu implementovat v iOS aplikaci, nastává problém při separaci jednotlivých vrstev. Ty jsou úzce propojeny a závisí na sobě, tedy např. prezentační vrstva ví o řídící vrstvě (Controller) a zároveň o vrstvě modelové. Toto se zásadně podepíše na možnosti znovu použitelnosti, kterékoliv z entit.
[8][9]

3.8.3.2 Apple's MVC (Cocoa MVC)



Obrázek 3 – Vize MVC od společnosti Apple
[Vlastní zpracování]

Očekávání MVC architektury od Apple byla následující. Řídící vrstva „Controller", který zastupuje roli prostředníka mezi prezentační vrstvou „View" a datovou vrstvou „Model". Tím pádem by byla zaručena značná separace jednotlivých vrstev, alespoň z pohledu prezentační a datové vrstvy, které o sobě navzájem nemají žádnou informaci, obě vrstvy přistupují k sobě přes řídící vrstvu. Tímto způsobem by byla zaručena znovu použitelnost entit datové a prezentační vrstvy. Bohužel toto očekávání se nenaplnilo.



Obrázek 4 - Cocoa MVC
[Vlastní zpracování]

Reálné Cocoa MVC od společnosti Apple podporuje psaní masivních řídících vrstev (Controller) a to díky tomu, že Apple pozici řídící vrstvy úzce spojil se životním cyklem prezentační vrstvy tak, že lze jen těžce tvrdit, že jsou třídy od sebe oddělené. Ve

skutečnosti se problém zdá být ještě rozsáhlejší. Controller také slouží jako delegát a zdroj dat pro prezentační vrstvu a na druhou stranu je obvykle zodpovědný za odesílání a rušení requestů na síť. Můžeme zde říci, že řídící vrstva je tou nejdůležitější částí v architektuře a kvůli její komplexnosti se v rozsáhlých aplikacích stává téměř netestovatelnou.
Příklad:

```
let cell = tableView.dequeueReusableCell(
            withIdentifier: cellIdentifier,
            for: indexPath
            ) as? WeatherCell
cell.configureCell(forecast: forecast!)
```

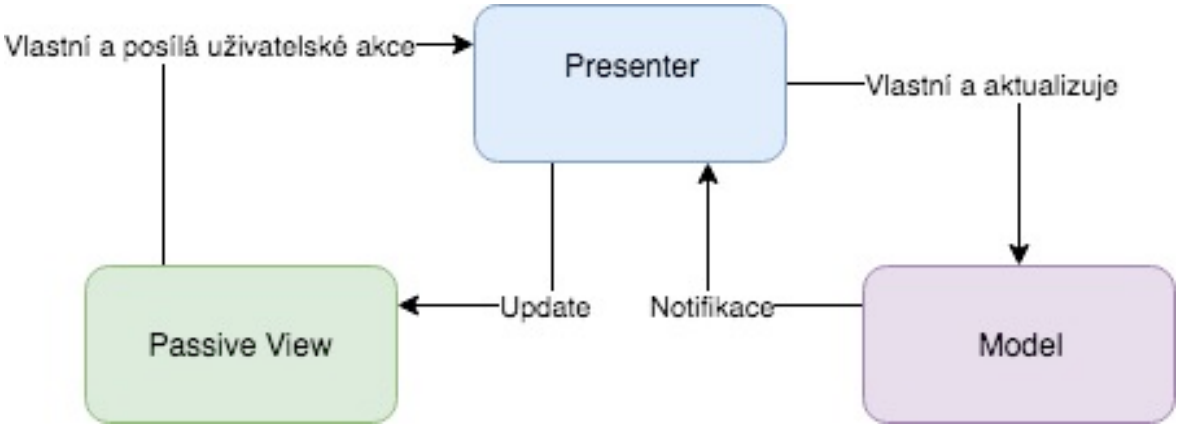
Buňka tabulky (UITableViewCell) je nakonfigurována přímo pomocí datové vrstvy. Tato inicializace buňky je prováděna na řídící vrstvě. Tedy pokyny tradičního MVC jsou porušeny a toto se děje v Cocoa MVC ve všech případech. Pokud bychom se drželi pokynů MVC architektury, je potřeba zamezit přístupu datové vrstvy k prezentační, tedy je v řídící vrstvě striktně separovat, to bohužel mnohem více rozšíří řídící vrstvu.
Díky tomuto počinu se Cocoa MVC v nezkrácené verzi nazývá „Massive View Controller“, tedy Masivní řídící vrstva. Tento problém není evidentní, dokud nepřijde na řadu testování, nebo také oprava chyb. První překážkou v testování je úzké propojení prezentační a řídící vrstvy. Vývojář je nucen mockovat data (nahrazovat produkční data, za data testovací) na řídící vrstvě pro prezentační vrstvu a její životní cyklus. Musí se tak při psaní kódu řídící vrstvy dávat pozor na výrazné oddělení business logiky od prezentační vrstvy.

```
class GreetingViewController : UIViewController { // View + Controller
    var person: Person!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:",
forControlEvents: .touchUpInside)
    }
    func didTapButton(button: UIButton) {
        let greeting = "Hello \(person.firstName) \(person.lastName)"
        self.greetingLabel.text = greeting
    }
}
```

Příklad uvedený výše poukazuje na obtížnou testovatelnost řídící vrstvy. Zde můžeme přesunout „greeting“ do nové datové třídy např. „GreetingModel“ a testovat jej

zvlášť, bohužel nám to zamezuje testování prezentační logiky, naštěstí ne v tomto příkladu, v řídící vrstvě která přímo volá metody „UIView“ (viewDidLoad a DidTapButton), tyto metody mohou způsobit načtení všech tříd prezentační vrstvy. Tento jev je v testování velice nežádoucí.
[8][9]

3.8.4 MVP – Model View Presenter



Obrázek 5 - MVP
[Vlastní zpracování]

MVP neboli „Passive View Variant“ je architektonický vzor vytvořený na základě vylepšení Cocoa MVC na očekávanou architekturu. Hlavním rozdílem oproti Cocoa MVC separování prostředníka mezi datovou a prezentační vrstvou. Tento prostředník „Presenter“ nemá již co do činění například se životním cyklem prezentační vrstvy a není s ní úzce spojen. Díky tomu může být prezentační vrstva, neboli „Passive View“ jednoduše namockována (nahrazena testovacími daty). Řídící vrstva je zde zodpovědná za překreslování prezentační vrstvy na základě dat a stavu. V MVP jsou podtřídy UIViewControlleru, obsaženy v prezentační vrstvě.

```
class GreetingPresenter {
    unowned let view: GreetingView
    let person: Person
    required init(view: GreetingView, person: Person) {
        self.view = view
        self.person = person
    }
    func showGreeting() {
        let greeting = "Hello \(person.firstName) \(person.lastName)"
        self.view.setGreeting(greeting)
    }
}
```

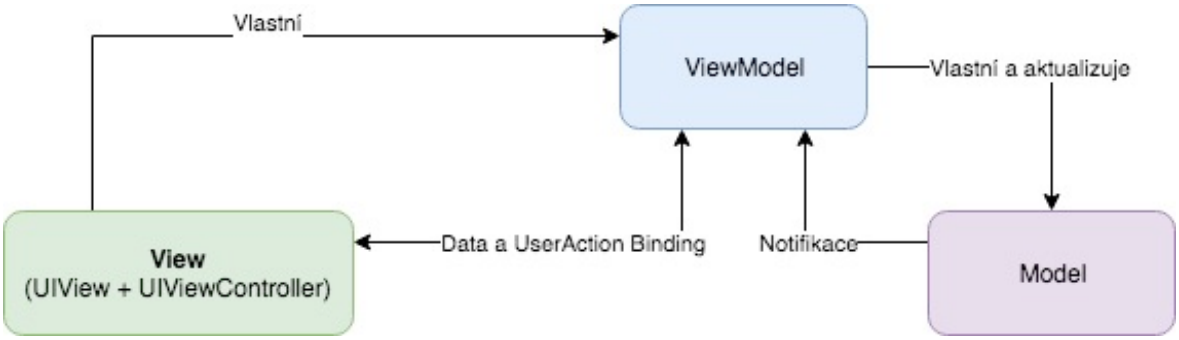
Výše uvedený „GreetingPresenter“ je inicializován s modelovou třídou „Person“ a prezentační třídou „GreetingView“. Zde je názorná ukázka funkce zprostředkovatele, kde pomocí funkce „showGreeting“ předá data prezentační vrstvě.

```
class GreetingViewController : UIViewController, GreetingView {
    var presenter: GreetingViewPresenter!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:",
forControlEvents: .TouchUpInside)
    }
    func didTapButton(button: UIButton) {
        self.presenter.showGreeting()
    }
    func setGreeting(greeting: String) {
        self.greetingLabel.text = greeting
    }
}
```

Výše uvedený „GreetingViewController“ ve svém životním cyklu a ve funkcích UIView() pouze používá data předané presenterem.
MVP je prvním vzorem, který odkrývá problematiku separovaných vrstev. U iOS aplikací je potřeba na některém místě v aplikaci určit, jaká entita prezentační vrstvy má být aktuálně zobrazena. Pro tento případ je potřeba v aplikaci vytvořit jakýsi „Router“, který bude zodpovědný za navigaci v aplikaci a také bude inicializovat jednotlivé instance a předávat jim potřebné závislosti. Tento router bude muset být dostupný napříč celou aplikací, abychom zajistili View-to-View prezentaci. Tento krok bude muset být použit i v následujících architekturách.
[8][9]

3.8.5 **MVVM – Model View ViewModel**

Teoreticky Model-View-ViewModel vypadá velmi dobře. Prezentační a datová vrstva jsou velice podobné například architektuře MVP, jediným rozdílem na první pohled je prostředník mezi prezentační a datovou vrstvou, v tomto případě se jedná o tzv. ViewModel.



Obrázek 6 – MVVM
[Vlastní zpracování]

Podobnost s MVP je shodná i na úrovni nezávislosti prezentační a datové vrstvy. Prezentační vrstva obsahuje kromě pasivních UIView také UIViewController. MVVM využívá pro komunikaci mezi prezentační vrstvou a ViewModelem vazby. Pomocí těchto vazeb jsou zprostředkovány uživatelské akce a práce s daty. Např. Uživatel stiskne tlačítko pro přihlášení. Tato akce je napojená na ViewModel, který jí poslouchá a na základě vyvolání akce aktualizuje datovou vrstvu, která vyvolá request pro přihlášení.

ViewModel je v podstatě je UIKit nezávislé zastoupení View a jeho stavu. ViewModel vyvolá změny v modelu a aktualizuje se s aktualizovaným modelem a protože máme vazbu mezi ViewModel a View, tak je View odpovídajícím způsobem aktualizováno také.

3.8.5.1 Vazby (Bindings)

Vazby pochází z balíčku OS X vývoje. Tedy desktopových aplikací určených pro operační systém OS X společnosti Apple. Bohužel nejsou obsaženy v balíčcích pro vývoj iOS. Jedinou náhradou je možnost napsat vlastní vazby, jako nadstavbu pro KVO observer spojené s notifikacemi. To už bohužel není tak pohodlné jako při vývoji OS X aplikací.

Kvůli této situaci vznikají reaktivní frameworky určené pro vývoj iOS, které absenci vazeb doplňují. Reaktivní frameworky poskytují vývojáři velmi silný nástroj co se týče vazeb a asynchronního volání jednotlivých funkcích. Značnou nevýhodou reaktivního programování bude debuggování chyb v aplikaci. To vyžaduje často zkušenějšího vývojáře, nežli je tomu v předchozích zmíněných architekturách.

```
class ViewModel {
    let person: Person
    var greeting: String? {
        didSet {
```

```
            self.greetingDidChange?(self)
        }
    }
    var greetingDidChange: ((GreetingViewModelProtocol) -> ())?
    required init(person: Person) {
        self.person = person
    }
    func showGreeting() {
        self.greeting = "Hello \(person.firstName) \(person.lastName)"
    }
}
```

Příklad ViewModelu, který využívá v jednoduchosti CallBack funkci, která se inicializuje na ViewControlleru a pomocí showGreeting() funkce, která je navázaná na uživatelskou akci po stisknutí tlačítka showGreetingButton.

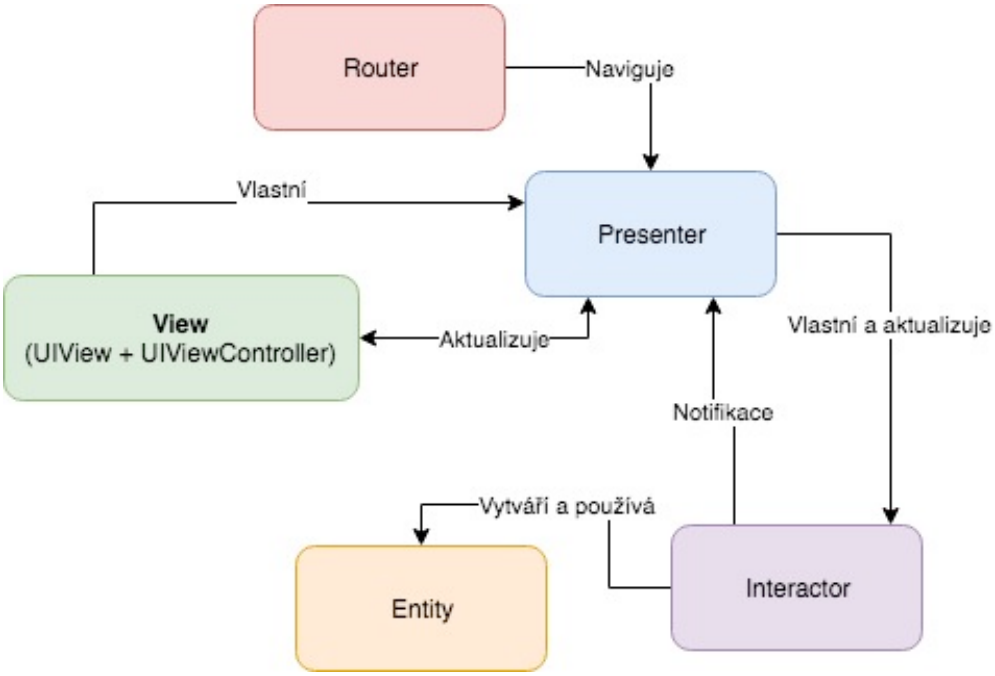
```
class ViewController : UIViewController {
    var viewModel: ViewModel! {
        didSet {
            self.viewModel.greetingDidChange = { [unowned self]
viewModel in
                self.greetingLabel.text = viewModel.greeting
            }
        }
    }
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self.viewModel, action:
"showGreeting", forControlEvents: .TouchUpInside)
    }
}
```

V tomto jednoduchém příkladu je použití KVO a notifikací zbytečné, místo toho je vhodnější explicitně žádat ViewModel aby se aktualizoval použitím showGreeting() funkce.

[8][9]

3.8.6 VIPER – View Interactor Presenter Entity Router

Zkušenosti s budováním stavebnice Lego přenesené do architektury iOS aplikace. VIPER je posledním kandidátem ve srovnání a zároveň jako jediný nepatří do kategorie MV(X), tedy Model, View a prostředník mezi nimi. VIPER separuje jednotlivé odpovědnosti mezi další iteraci a vzniká tak pět vrstev architektury



Obrázek 7 – VIPER
[Vlastní zpracování]

View – zodpovídá za uživatelské rozhraní aplikace. Obsahuje jak funkce UIView tak UIViewController, tak jako tomu je u View v MVVM.

```
class GreetingViewController : UIViewController {
    var eventHandler: GreetingViewEventHandler!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:",
forControlEvents: .TouchUpInside)
    }

    func didTapButton(button: UIButton) {
        self.eventHandler.didTapShowGreetingButton()
    }

    func setGreeting(greeting: String) {
        self.greetingLabel.text = greeting
    }
}
```

Interactor - obsahuje business logiku související s daty (entitami) nebo sítí, jako je vytváření nových instancí entit nebo jejich načítání ze serveru. Pro tyto účely se používají některé služby a manažery, které nejsou považovány za součást modulu VIPER, ale spíše jako externí závislosti.

```
protocol GreetingProvider {
    func provideGreetingData()
}
class GreetingInteractor : GreetingProvider {
    weak var output: GreetingOutput!

    func provideGreetingData() {
        let person = Person(firstName: "David", lastName: "Blaine")
let subject = person.firstName + " " + person.lastName
        let greeting = GreetingData(greeting: "Hello", subject: subject)
        self.output.receiveGreetingData(greeting)
    }
}
```

Presenter - obsahuje business logiku související s uživatelským rozhraním (je nezávislý na UIKit), vyvolává metody na Interactoru.

```
protocol GreetingOutput: class {
    func receiveGreetingData(greetingData: GreetingData)
}
protocol GreetingViewEventHandler {
    func didTapShowGreetingButton()
}
class GreetingPresenter : GreetingOutput, GreetingViewEventHandler {
    weak var view: GreetingView!
    var greetingProvider: GreetingProvider!

    func didTapShowGreetingButton() {
        self.greetingProvider.provideGreetingData()
    }

    func receiveGreetingData(greetingData: GreetingData) {
        let greeting = "\(greetingData.greeting)
\(greetingData.subject)"
        self.view.setGreeting(greeting)
    }
}
```

Entities – jedná se pouze o objekty reprezentující data, nikoliv datovou vrstvu zodpovědnou za přístup k datům

```
struct Person {
    let firstName: String
    let lastName: String
}

struct GreetingData {
    let greeting: String
    let subject: String
}
```

Router - zodpovědný za přechody mezi VIPER moduly a navigaci jednotlivých Presenterů.

Modul VIPER může být v podstatě jedna obrazovka nebo celý uživatelský příběh vaší aplikace – např. autentizace, která může být jedna obrazovka nebo několik souvisejících. Velikost a funkcionalitu modulů určuje vývojář.
Porovnáme-li VIPER s typem MV (X), uvidíme několik rozdílů v rozdělení povinností:
Logika Modelu (interakce dat) se přesunula do Interactoru s Entitami jako jednoduchými datovými strukturami. Do funkce Presenter se přesunuly pouze funkce reprezentace uživatelského rozhraní Controller / Prezenter / ViewModel, nikoliv však funkce pro změnu dat. VIPER je první architektonický vzor, který výslovně řeší odpovědnost za navigaci, pomocí Routeru. Správný způsob, jak provádět navigaci, je výzvou pro aplikace iOS., architektury MV (X) prostě tento problém neřeší a je potřeba ho do architektury přidat.
Během používání architektury VIPER v aplikaci se mnoho vývojářů může cítit, jakoby stavěli Pražský hrad ze stavebnice lego. Mnoho projektů díky použití této architektury čelila problémům s náklady na vývoj a udržovatelnost. Je potřeba říci, že tato architektura je určena spíše pro rozsáhlé projekty s velkým počtem funkcionalit.

Během používání služby VIPER se můžete cítit jako budování The Empire State Building z bloků LEGO a to je signál, že máte problém. Možná je příliš brzy na přijetí VIPER pro vaši aplikaci a měli byste zvážit něco jednoduššího. Někteří vývojáři to ignorují a pokračují v ní domnívajíc se, že pro jejich aplikace bude přínosem přinejmenším v budoucnu, i když nyní náklady na údržbu jsou nepřiměřeně vysoké. Pokud si myslíte totéž, doporučil bych vám zkusit „Generamba“, jedná se o nástroj pro generování koster architektury VIPER, který může urychlit psaní rozhraní a tříd.
[8][9]

3.9 Vývoj pro platformu iOS

S rozvíjejícím se trendem v oblasti mobilních zařízení se postupně vyvíjí i možnosti vývoje aplikací určených pro operační systém iOS. V iOS je možné vyvíjet aplikace napsané v jazyku C, v jeho nadstavbě Objective-C a nebo v jazyce Swift. Jedná se o nativní vývoj pro tuto platformu.
Dlouhou dobu bylo možné vyvíjet aplikace pouze v aplikaci XCode, což je vývojové prostředí od firmy Apple, nabízené a dostupné zdarma. Toto vývojové prostředí, neboli IDE, je však dostupné pouze pro operační systém Mac OS X, tudíž vývoj v ostatních operačních systémech např. ve Windows či Linuxu nebyl možný. Tento problém se

pokusilo vyřešit několik projektů, které se snažily kompilovat programy napsané v jiných programovacích jazycích do nativního kódu Objective-C, nebo Swift. Asi největším počinem v této oblasti je krok společnosti Adobe, která v nové verzi svého nástroje pro vývoj aplikací Flash umožňuje kompilovat právě do programu určeného pro iOS. Tento a jemu podobné nástroje však byly zakázány v licenčním ujednání, ale po velké nevoli ze strany vývojářů byly opět povoleny. Dnes existuje nepřeberné množství vývojových prostředí ať už nativně pro iOS nebo hybridně (crossplatform, vývoj pro více mobilních platforem zároveň) např. Xamarin, který úzce souvisí s vývojovým prostředím Visual studio od společnosti Microsoft, nebo frameworky založené na HTML 5 a javascriptu např. ReactNative, Ionic a jiné, které se stávají mezi vývojáři velice populární.
[8][9]

3.9.1 **3.14.2 Xcode**

Xcode je IDE společnosti Apple, které obsahuje balíček profesionálních vývojářských nástrojů pro vývoj softwarových aplikací na platformy iOS a OS X. Nejnovější dostupná oficiální verze je 9.2, ta umožňuje vyvíjet aplikace na nejnovější verze operačních systémů Apple iOS 11 a macOS high Sierra 10.3. Apple ho nabízí volně ke stažení z App Store, ale pouze pro operační systémy OS X (macOS). V Xcode je možné implementovat kód v jazyce C, Objective-C a v jazyce Swift. Pro uživatelské rozhraní jsou zde použity frameworky (knihovny) Cocoa, pro OS X a Cocoa Touch pro iOS aplikace.
[9]

3.9.2 **Objective-C**

Objective-C, často nazývaný ObjC, je objektově orientovaný programovací jazyk implementovaný jako rozšíření jazyka C, do kterého byl přidán systém zasílání zpráv z jazyka Smalltalk. V současné době je používán v operačních systémech Mac OS X, iOS a v GNU projektu GNUstep. Obě prostředí jsou založena na standardu OpenStep.
Překladač tohoto jazyka je součástí GCC. Ovšem nejpoužívanějším překladačem v současné době je clang, díky jeho použití firmou Apple v Xcode.
[3]

3.9.3 **Swift**

Swift je kompilovaný programovací jazyk od společnosti Apple určený pro vývoj

na platformách Mac OS X a iOS. Vývoj tohoto programovacího jazyka začal v roce 2010 společností Apple. Oficiálně byl představen v roce 2014 na Apple WWDC 2014. Je považován za alternativu Objective-C a je bezpečnější než jeho předchůdce, tzn. nedovolí programátorovi tolik chyb. Zápis kódu je zjednodušený a umožní nám kratší zápis. Umí spolupracovat s existujícími frameworky Cocoa a Cocoa Touch. Swift je kompilován pomocí LLVM a ve stejném programu může být spolu s kódem v jazycích C, Objective-C a C++. Dalšími výhodami Swiftu jsou: možnost kratšího zápisu kódu, oproti Objective-C a C a dále rychlost operací - např. řazení velkého počtu dat umožní o 50 % rychleji.

Ukázka zápisu kódu v Objective-C a Swiftu

```
NSString *str = @"hello,";
str = [str stringByAppendingString:@" world"];
```

kód v Objective-C

```
var str = "hello,"
str += " world"
```

kód ve Swiftu

[3]

3.9.4 **Reaktivní programování**

Reaktivní programování je programování s asynchronními datovými toky. Práce s asynchronními datovými toky je již známá ve vývoji iOS aplikací pro poslouchání událostí vyvolaných uživatelskými akcemi. Sběrnicové události nebo uživatelské akce např. kliknutí na tlačítko, jsou skutečně asynchronní datové toky, které lze sledovat a na základě sledování vyvolat určitou událost.

Oproti tomu je myšlenka reaktivního programování rozšířená. Lze vytvořit asynchronní datový tok všeho, a ne pouze jen z uživatelských akcí. Datové toky nejsou náročné na výpočet a jsou všudypřítomné. V podstatě cokoliv může být datovým proudem a to např. proměnné, uživatelské vstupy, vlastnosti, mezipaměť (cache), datové struktury apod. Na datový tok lze poslouchat a na základě toho odpovídajícím způsobem reagovat. Například si můžeme představit, že váš zdroj Twitter bude datovým tokem stejným způsobem jako události kliknutí.

Kromě toho reaktivní programování společně s použitím reaktivních frameworků např. RxSwift nebo ReactiveSwift, poskytuje rozsáhlou sadu nástrojů pro práci s datovými

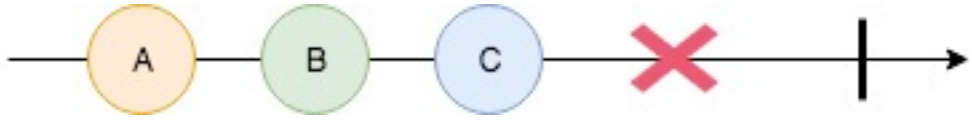
toky, napomáhajícím ke kombinování, vytváření, filtrování a změně datových proudů. Proud může být použit jako vstup do jiného. Dokonce i více streamů lze použít jako vstupy do jiného streamu. Můžete sloučit dva proudy. Můžete filtrovat proud a získat jiný, který má pouze ty události, které vás zajímají. Můžete namapovat hodnoty dat z jednoho streamu na jiný nový. Základem reaktivního programování je programování funkcionální.

Datový tok je sekvence průběžných událostí seřazených v čase. Může vysílat tři různé věci: hodnotu (nějakého typu), chybu nebo signál "dokončený". Datový tok může vypadat následovně. Pokud aktuální okno obsahuje tlačítko, je datový tok dokončený. Datový tok v tomto případě poslouchá aktuální zobrazené okno, a pokud je v něm obsaženo tlačítko, vyšle signál „Dokončený“.

Vysílané události se zachycují asynchronně, definováním funkce, která bude provedena, jakmile datový tok zachytí hodnotu. Jiná funkce bude prováděna, pokud datový tok zachytí chybu (error). Poslední funkce je definována pro zachycení dokončeného stavu „Completed“. V některých případech mohou být poslední dvě funkce vynechány a definována pouze funkce pro hodnotu. Poslouchání na datový tok se nazývá „Subscribing“. Funkce, které definujeme, se nazývají „Observer“ a datový tok, na který se poslouchá, se nazývá „Observable“. Základ reaktivního programování vyplívá z návrhového vzoru Observer.

Návrhový vzor Observer umožňuje objektu „Observer“ získávat informace o pozorovaném objektu pokaždé, když se stav pozorovaného objektu změní. Na základě této změny může vyvolat akci. Předmět pozorování si interně udržuje seznam objektů, kteří ho pozorují a automaticky jim posílá zprávy (volá jejich metody) pokaždé, když se jeho stav změní. Observer má možnost tyto události filtrovat a vybrat si tak jen určité z nich, které ho zajímají.

Na následujícím příkladu jsou graficky znázorněny datové toky pomocí časové osy, barevná kolečka značí hodnoty v datovém toku v určitém čase. Svislá čára značí signál „Completed“, tedy dokončený. Toto grafické znázornění se nazývá „Marble diagram“. [11][12][13]



Obrázek 8 - Marble Diagram

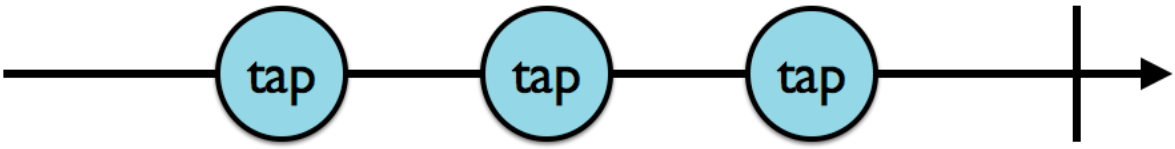
[Vlastní zpracování]

3.9.5 **RxSwift**

RxSwift a RxCocoa jsou součástí sady nástrojů ReactiveX (obvykle zkráceně "Rx"), které spadají do několika programovacích jazyků a platforem. Zatímco ReactiveX začal jako součást ekosystému .NET / C #, stal se extrémně populární u Ruby, JavaScriptorů a zejména Java a Android vývojářů. RxSwift je framework pro interakci s programovacím jazykem Swift, zatímco RxCocoa je framework, který usnadňuje používání Cocoa API používaných v systémech iOS a OS X pomocí reaktivních technik. Frameworky ReactiveX jsou navrženy tak, aby poskytovaly společnou slovní zásobu pro určité úkoly, které se opakovaně používají v různých programovacích jazycích. Toto (teoreticky) usnadňuje soustředění se na syntaxi samotného jazyka, spíše než ztrácet čas, když zjistí, jak vyřešit společný úkol do každého nového jazyka.
[11][12][13]

3.9.5.1 Observable

Observable je datový tok, neboli sekvence, využívaná v reaktivním programování. Mezi její důležité vlastnosti tak jak již bylo řečeno je asynchronost. Tato sekvence je vždy ohraničená v čase. Sekvence produkuje takzvané eventy, procesu této produkce se říká „Emitting“. Eventy mohou nabývat číselných hodnot, instancí a také vlastních typů, které jsou pro sekvenci vytvořeny, nebo také rozeznaná gesta, například kliknutí na tlačítko.



Obrázek 9 - Observable sekvence pomocí Marble Diagramu
[13]

Na výše uvedeném diagramu, si znázorníme životní cyklus sekvence. Tato sekvence emittuje tři eventy. Po každém úspěšném emittu eventu se volá funkce „next()“, která čeká na následující event v sekvenci dokud nenastane „Complete“, nebo „Error“. Po

dosažení události complete a nebo error, sekvence terminuje a nadále již nemůže emitovat eventy. (viz příloha č. 2)

Vytvoření Observable v jazyce Swift:

```
let one = 1
let two = 2
let three = 3
let observable = Observable<Int>.just(one)
```

Uvedený Observable je typu „Int“, neboli integer, celočíselná hodnota. V tomto případě to znamená, že Observable může emitovat eventy pouze pro hodnoty typu Int. Můžeme také inicializovat proměnnou bez implicitního typování, jelikož programovací jazyk Swift, při kompilaci typy sám rozezná.

```
let observable2 = Observable.of(one, two, three)
```

[11][12][13]

3.9.5.2 Subscribing

Subscribing, tedy systém poslouchání notifikací a Observers, je velice podobný přihlašování se na notifikace NotificationCenter, který se využívá v iOS a macOS aplikacích. Příklad návrhového vzoru Observer využívaný v iOS:

```
let observer = NotificationCenter.default.addObserver(
    forName: .UIKeyboardDidChangeFrame,
    object: nil,
    queue: nil
) { notification in
    // Co se má stát s notifikací
}
```

Příklad návrhového vzoru Subscribing v RxSwift:

```
let one = 1
let two = 2
let three = 3
let observable2 = Observable.of(one, two, three)
observable.subscribe { print($0)}
```

Výsledek z konzole:

```
next(1)
```

```
next(2)
next(3)
completed
```

Při Subscribing Observables v RxSwift je potřeba zavolat jednu z metod na poslouchání (observe) Observables. Místo metody addObserver(), která se používá v jazyce swift, se používá metoda subscribe() a její mutace. Dalším rozdílem je při použití NotificationCenter se často využívá „.default“, což je v tomto případě instance návrhového vzoru Jedináček (Singleton). V Rx každá Observable je rozdílná a návrhový vzor Jedináček se zde nevyužívá.

Nejdůležitějším bodem při Subscribing Observables je použití tzv. Subscriber. Pokud není použit Subscriber pomocí metody „subscribe()“, nebo dalším alternativám. Sekvence není poslouchána a zmíněné eventy nikdy nebudou emitovány. Subscriber je v podstatě shodný s funkcí „next()“ na iterátoru, ve standartní knihovně jazyka Swift.

```
let sequence = 0..<3
var iterator = sequence.makeIterator()
while let n = iterator.next() {
  print(n)
}
```

Výsledek z konzole:

```
0
1
2
```

[11][12][13]

3.9.5.3 DisposeBag

Kromě již zmíněného návrhového vzoru Observer je nutné vysvětlit i další nástroj pro reaktivní frameworky a tím je „DisposeBag“, který pomáhá řešit správu automatického počítání referencí (dále jen „ARC“).

a paměti. Jedná se o virtuální " tašku" objektů Observer, které jsou odstraněny, když je jejich nadřazený objekt přerušený (událostí „Complete“, nebo chybou). Když se na objekt, který má DisposeBag jako vlastnost, zavolá deinicializace, taška je "vyprázdněna" a každý jednorázový pozorovatel se automaticky odhlásí z toho, co pozoruje. To umožňuje uvést

ARC do původního stavu. Bez DisposeBagu bychom získali jeden ze dvou výsledků. Observer by vytvořil zadržující cyklus, který by visel na tom, co pozoruje po nekonečně dlouhou dobu, dokud by nedošlo k přetečení zásobníku, nebo by se mohl dealokovat ze stávajícího objektu a způsobit pád aplikace. Proto je potřeba všechny Observable objekty přidávat do DisposeBagu. Pomocí DisposeBagu můžeme ukončit subscribing na Observable, např. po dosažení požadované hodnoty.

Příklad DisposeBagu v RxSwift:

```
let disposeBag = DisposeBag()
Observable.of("A", "B", "C")
    .subscribe {
        print($0)
    }
    .disposed(by: disposeBag)
```

Častěji pro ukončení Observable sequence a pro zachycení možných událostí se využívá funkce create(), která umožňuje implementaci manipulace s chybami a výsledky.

```
let observable = Observable<String>.create { observer in
    observer.onNext("1")
    observer.onCompleted()
    observer.onNext("?")
    return Disposables.create()
}
observable.subscribe(
    onNext: { print($0) },
    onError: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Disposed") }
    )
    .disposed(by: disposeBag)
```

Výsledek z konzole:

```
1
Completed
Disposed
```

[11][12][13]

3.9.5.4 Subject

Subjekt v RxSwift působí jako Observable i Observer. Funkce subjektů je založená na obdržení „.next“ eventů. Pokaždé, když obdrží event, vyšlou jej na jejich Subscribera. Známe čtyři druhy subjektů:

- PublishSubject: Spustí se prázdná a pouze předává nové hodnoty Subscriberům
- BehaviorSubject: Začíná s počáteční hodnotou a předá ji, nebo nejnovější hodnotu, novým Subscriberům.
- ReplaySubject: Inicializuje velikost bufferu a udržuje počet prvků až do této velikosti, celý buffer předá novým Subscriberům.
- Proměnná: Obalí BehaviorSubject, zachovává jeho aktuální hodnotu jako stav a předává jí novým Subscriberům.

Příklad subjektu v RxSwift:

```
subject.on(.next("1"))
```

Výsledek z konzole: `1`
[11][12][13]

3.9.5.5 Operátory

Operátory jsou stavebními kameny Rx, které můžete použít k transformaci, zpracování a reakci na události vysílané Observables. Stejně jako lze kombinovat jednoduché aritmetické operátory jako +, -, a / pro vytváření komplexních matematických výrazů, můžete řetězit a sestavit dohromady jednoduché operátory Rx pro vyjádření komplexní logiky aplikací.
Prvním typem operátorů je filtrování. Filtrování umožňuje zpracovat jen některé eventy a ostatní ignorovat, dle zadaných kritérií, neboli jsou to operátory, které selektivně emitují hodnoty z Observable.
Zástupci filtrovacích operátorů:

- Debounce – vrátí hodnotu ze sekvence, pokud uplynula určitá doba, bez emittu další hodnoty
- Distinct - vyřazuje duplicitní hodnoty v sekvenci

- ElementAt – vrací pouze hodnotu „n“ ze sekvence
- Filter - vrátí pouze ty hodnoty z pozorovatelného objektu, které projdou predikátovým testem
- First - vrátí pouze první položku nebo první položku, která splňuje podmínku, z pozorovatelného objektu
- IgnoreElements - nevrací žádné položky ze sekvence, pouze zrcadlí oznámení o ukončení
- Last – vrací pouze poslední hodnotu nebo poslední hodnotu, která splňuje podmínku, z pozorovatelného objektu
- Sample - vrací nejnovější hodnotu v sekvenci v pravidelných časových intervalech
- Skip - vyřazení prvních n položek v sekvenci
- SkipLast - vyřazení poslední položky „n“ v sekvenci
- Take - vrátí pouze první hodnoty „n“ v sekvenci
- TakeLast - vrátí pouze poslední hodnoty „n“ v sekvenci

Druhým typem operátorů je transformování. Transformování umožňuje sekvenci změnit na sekvenci např. jiného typu.

Zástupci transformačních operátorů:

- Buffer - pravidelně shromažďují hodnoty ze sekvence do svazků, které vrací
- FlatMap – transformuje hodnoty v sekvenci, vyřazuje prázdné hodnoty a vrací sekvenci novou
- GroupBy – rozdělí sekvenci do sady sekvencí, kde každá sekvence obsahuje odlišnou množinu hodnot, řazených podle klíče
- Map - transformace hodnot sekvence vysílaných použitím transformační funkce
- Scan – použitím predikátové funkce, postupně vrací po sobě jdoucí hodnoty

Posledním typem operátorů je kombinování. Kombinování umožňuje spojit více sekvencí do jedné.

Zástupci kombinačních operátorů:

- And/Then/When — kombinuje sadu hodnot ze dvou a více sekvencí
- CombineLatest — kombinuje položky ze dvou sekvencí, vrací nejnovější hodnotu z obou sekvencí, kombinace je prováděna na základě definované funkce
- Merge — kombinuje více sekvencí, spojením jejich emitovaných hodnot

- StartWith — kombinuje sekvenci počátečních hodnot
- Switch — převede vice sekvencí do jedné a ta emittuje nejnovější hodnoty.
- Zip — kombinuje více sekvencí dohromady přes stanovenou funkci a vrací jednotlivé hodnoty pro každou kombinaci na základě výsledků této funkce

[11][12][13]

4 Vlastní práce

4.1 Komparace architektury iOS aplikací

Komparace architektur byla provedena na základě teoretických poznatků získaných z kapitoly [3.5] SW architektura iOS. Byly zvoleny parametry pro hodnocení pomocí bodovací metody.

4.1.1 Rozdělení odpovědností

Rozdělení odpovědností pomáhá k lepší představě o tom, jak dané funkcionality pracují.

Je potřeba říci, že čím déle vývojář danou aplikaci vyvíjí, tím lépe se přizpůsobí na její komplexnost. Bohužel tato schopnost se roste pouze lineárně a její růst se zastaví poměrně rychle. Proto nejjednodušší cestou jak se postavit komplexnosti aplikace, je rozdělit její funkcionality, datové struktury atd. do logických celků s jasně definovanými odpovědnostmi, které následují jednotný odpovědnostní princip. To napomůže v orientaci v kódu a zlepší jeho udržovatelnost.

4.1.2 Testovatelnost

O testovatelnosti aplikace a vůbec o použití např. unit testů v aplikaci je obecně pro programátory velké téma. Bohužel v praxi je velký tlak na programátory a aplikace jsou často velmi silně produktově orientované, což ve skutečnosti znamená, že firmy pokládají testovatelnost aplikace za nadbytečnou. Testy v aplikaci, pokud opravdu testují funkcionality a práci s daty apod., velmi přispívají k rychlejší distribuci kvalitního produktu a ve většině případů by se nemělo stát, že v produkční verzi aplikace se nachází chyby v datech, funkcionality nepracují tak jak by měli, či nedochází k pádům aplikace. Pokud se chyba stane u koncového uživatele tak její oprava může trvat i týdny.

4.1.3 Snadné použití a nízké náklady na udržovatelnost

Nejlepší kód je takový kód, který nikdy nebyl napsán. Neboli zde často platí pravidlo, čím méně kódu aplikace obsahuje, tím je v něm lepší orientace a méně chyb. Samozřejmě je potřeba se nad danou problematikou zamyslet a vždy upřednostnit co možno nejchytřejší řešení, tak aby kód byl dobře udržovatelný a nejlépe i znovu

použitelný. Zde velice přispívá i logické nazývání jmenných prostorů, tak aby vývojáři podle názvu bylo srozumitelné co daná funkce, či třída vykonává.

4.1.4 MVC – Model View Controller

- Rozdělení odpovědností - prezentační a modelová vrstva jsou od sebe odděleny, ale prezentační a řídící vrstva jsou spolu velmi úzce propojeny.
- Testovatelnost - díky špatnému rozdělení odpovědností, je velice obtížné otestovat řídící vrstvu, pravděpodobně testování bude probíhat pouze na vrstvě datové
- Snadné použití a náklady na udržovatelnost – Cocoa MVC dosáhne pravděpodobně k nejmenšímu množství kódu mezi ostatními vzory. Další nedílnou výhodou je, že všichni iOS vývojáři jsou s ní obeznámeni, to vede k možnosti udržování kódu i méně zkušenějšími vývojáři. Cocoa MVC je také nejlepším architektonickým vzorem, pokud se bavíme o rychlosti vývoje. Bohužel architektura je spojena s vyššími náklady na údržbu kódu. Je velice vhodný pro méně rozsáhlé projekty, případně pro začínající vývojáře.

4.1.5 MVP – Model View Presenter

- Rozdělení odpovědností – Většina odpovědností v MVP je rozdělena mezi řídící a datovou vrstvu, prezentační vrstva je zde bez jakékoliv logiky a slouží pouze pro grafické rozhraní.
- Testovatelnost – Lze dosáhnout velice dobré testovatelnosti aplikace, díky možnosti oddělení jednotlivých vrstev, které lze namockovat (včetně prezentační vrstvy)
- Snadné použití a náklady na udržovatelnost – Ve výše uvedeném příkladu, můžeme vidět, že kód oproti Cocoa MVC je v téměř dvojnásobný, na druhou stranu myšlenka MVP je velmi smyslu plná a velice dobře se v ní orientuje, tím pádem udržovatelnost kódu je oproti Cocoa MVC znatelně lepší.

4.1.6 MVVM – Model View ViewModel

- Rozdělení odpovědností – Oproti MVP má rozdělení odpovědností rozprostřené mezi všechny tři vrstvy. Prezentační vrstva zde zodpovídá nastavení vazeb a

pomocí vazeb aktualizuje svůj stav. V MVVM je ViewModel prostředník který na základě akcí z prezentační vrstvy aktualizuje vrstvu datovou.

- Testovatelnost - díky separaci jednotlivých vrstev je testování Modelu a ViewModelu velice snadné, dokonce je přínosné otestovat vazby View na ViewModelu, které je snadné namockovat. Jediným problémem testování View, jakožto prezentační vrstvy, tím že využívá funkce životního cyklu UIView a UIViewControlleru a pouze prezentuje jednotlivá data, při použití frameworku RxCocoa nebo ReactiveCocoa jsou tyto funkce deklarovány reaktivně, ve většině případů k testování prezentační vrstvy kromě UI testů nedochází.
- Snadné použití a náklady na udržovatelnost – MVVM v projektu vede k stejnému či podobnému množství kódu jako je tomu u MVP, pokud se použijí vazby na prezentační vrstvě, kód bude znatelně kratší. Udržovatelnost závisí na použitém přístupu programování. Pokud se bude jednat o reaktivní programování a použití frameworku RxSwift a nebo ReactiveSwift, bude udržovatelnost ve veliké míře záviset na zkušenostech vývojáře, jelikož reaktivní programování je alternativní a velice odlišný přístup od přístupu společnosti Apple.

4.1.7 **VIPER – View Interactor Presenter Entity Router**

- Rozdělení odpovědností - VIPER aplikuje nejlepší rozdělení odpovědností napříč zmíněnými architekturami, díky tomu že řeší navigaci a architektura je rozdělena do pěti vrstev.
- Testovatelnost - díky dobrému rozdělení odpovědností, je velice snadné otestovat vrstvy odděleně a pokrýt tak veškeré funkcionality aplikace.
- Snadné použití a náklady na udržovatelnost – použití VIPER architektury vyžaduje zkušené vývojáře, tak aby správně určily jednotlivé moduly a jejich funkcionality a rozdělili tak aplikaci do logických celků. Dále je potřeba napsat velké množství rozhraní pro třídy s velmi malými odpovědnostmi a to se projeví na větším množství kódu nežli je tomu v MV(X) architekturách. Na druhou stranu díky rozdělení odpovědností, se chyby v aplikaci hledají snadněji.

4.1.8 Shrnutí komparace architektur

Nejprve je nutné říci, že žádná zázračná architektura, která by splnila všechna kritéria, není. Architektura doporučovaná společností Apple, tedy modifikace modelu architektury MVC, Cocoa MVC má svá úskalí, o kterých Apple ví a tak i ve své dokumentaci uvádí, že se jedná o ilustrativní příklady a použití jejich architektury není nijak vynucováno. Úskalí Cocoa MVC jsme si představili v kapitole MVC. Proto velké množství vývojářů vede snahu o výběr ideálnější architektury pro jejich aplikaci. Cocoa MVC nevyhovuje zejména v ohledu rozdělení odpovědností a testovatelnosti, díky velmi rozsáhlé prezentační vrstvě, která v aplikaci bere zodpovědnost za business logiku i navigaci mezi okny. Její značnou výhodou je snadné použití, to je také jeden z hlavních důvodů, použití společností Apple. Pro začínající vývojáře je velice snadné použít tuto architekturu, díky rozsáhlému množství materiálů, ať už od společnosti Apple, tak od komunity. Čím rozsáhlejší aplikace tím horší bude udržovatelnost.

Architektura MVP se snaží docílit původní představy společnosti Apple pro architekturu aplikací iOS. Rozdělení odpovědností, je znatelně lepší, bohužel odpovědnosti jsou nesouměrně rozdělené pouze pro řídící a datovou vrstvu. Dosahuje lepší testovatelnosti, díky možné separaci jednotlivých vrstev. Napříč těmto skutečnostem jsou náklady na udržovatelnost vyšší. Datová a řídící vrstva obsahují veškerou logiku a funkcionality, tím pádem při rozsáhlejší aplikaci, bude hledání chyb velice obtížné podobně jako u Cocoa MVC.

Architektura MVVM je architektura používaná v .NET aplikacích dlouhou dobu. Zajišťuje souměrnější separaci vrstev. Je také druhou nejpoužívanější architekturou v iOS aplikacích hned po Cocoa MVC. MVVM nabylo na popularitě, díky rozdělení vrstev a také možností, použití reaktivního programování, které se v této architektuře nejčastěji používá. Komunita okolo reaktivního programování a vývoje iOS aplikací má pro architekturu MVVM již velké množství podkladů a tutoriálů, které mohou využít i začínající vývojáři. Testovatelnost díky dobrému rozdělení odpovědností je velice dobrá i s použitím reaktivních frameworků. Udržovatelnost aplikace závisí na zkušenostech vývojáře, pokud se bavíme o použití reaktivního programování, které vyžaduje poněkud rozdílný přístup k vývoji, než je tomu u objektového přístupu.

Architektura VIPER je posledním adeptem. Tato architektura je velmi rozsáhlá. Separace jednotlivých vrstev je oproti ostatním adeptům nejlepší. Testovatelnost díky

velmi dobré separaci je také nejlepší z uvedených architektur. Jedinou zábranou proč architekturu nevyužít, je její složitost. Architektura vyžaduje zkušeného vývojáře, tak aby se náklady na vývoj aplikace příliš nezvýšili. Pokud se dodržují v aplikaci všechny pravidla této architektury, je velice dobrým kandidátem pro rozsáhlé aplikace, které vyžadují dobrou orientaci v kódu, pro nově příchozí vývojáře a zároveň se snížili náklady na udržovatelnost aplikace, při hledání chyb.

Pro aplikaci diplomové práce jsem se rozhodl použít architekturu MVVM, která umožňuje použití reaktivního programování, tak aby se využili jeho výhody. Největším soupeřem pro ni byla architektura VIPER. Po zvážení časových nákladů na vývoj a rozsáhlost aplikace, bylo rozhodnuto pro MVVM, se kterou již mám zkušenosti.

Tabulka 1 - Bodovací tabulka komparace iOS architektur

	Cocoa MVC	MVP	MVVM	VIPER
Rozdělení odpovědností	2	3	4	5
Testovatelnost	1	4	4	5
Snadné použití	5	4	3	1
Udržovatelnost	2	3	4	4
Celkem:	10	14	15	15

Zdroj: Vlastní zpracování

4.2 Logický návrh

4.2.1 Příprava

Před grafickým návrhem je potřeba zamyslet se nad funkcionalitou aplikace a rozmístěním prvků, tak aby bylo zajištěno snadné uživatelské používání a tím uživatelský zážitek (UX – user experience). Je nutné si nejdříve představit, že aplikace určená pro děti v předškolním věku bude mít jinou koncepci, než aplikace určená pro vrcholné manažery velkých společností. Díky využití metod UX je možné předejít případným nedostatkům v

přehlednosti aplikace a usnadnit budoucím uživatelům její používání. Hlavním kritériem UX aplikace diplomové práce je použití nativních komponent a design by měl následovat „Human Interface Guidelines“ od společnosti Apple. Pokud vývojář následuje pokyny společnosti Apple pro tvoření uživatelského rozhraní aplikace, vyhne se tak možnému zamítnutí přijetí aplikace do obchodu AppStore. Použití nativních komponent tuto část velice zjednodušuje a zaručuje i to, že uživatel iOS zařízení je s nimi již obeznámen, díky širokému využití v ostatních aplikacích ať už společnosti Apple, či vývojářů.

Aplikace a její funkcionality jsou rozděleny zhruba do čtyř logických celků. Prvním z nich je přihlášení uživatele. Dlouho jsem přemýšlel, jestli bude vůbec přihlášení potřebné a došel jsem k závěru, že přihlášení do aplikace zařadím a to z důvodu ukládání dat na serveru, uživatel díky tomu bude schopen aplikaci se svými daty využívat napříč zařízeními. Přihlášení a autentizace uživatele bude probíhat pomocí emailu a hesla, nebo pomocí sociální sítě Facebook. Autentizace bude prováděna pomocí služeb aplikací třetích stran a to z důvodu vyšší bezpečnosti s využitím bezpečnostního protokolu pro autentizaci standartu OAuth 2.0.

Druhým logickým celkem je seznam potravin, které uživatel naskenuje. Jedná se o přehled všech potravin se základními údaji o expiraci a typu potraviny. Tak aby uživatel měl na jedné obrazovce přehled o všech potravinách.

Třetím logickým celkem je přidání potraviny. Musí být koncepčně postavená tak aby uživateli usnadnila přidání potravin bez značné ztráty času a neodradila ho od používání aplikace. Přidání potraviny bude založeno na skenování čárového kódu, ke kterému uživatel zadá expirační dobu a název. Pokud již produkt jednou přidal, uloží se naskenovaný čárový kód a druhé naskenování již bude spojeno se zadáním expirační doby. Prováděl jsem průzkum po databázi produktů podle čárových kódů, ale jedná se pouze o záležitost určenou pro velké supermarkety, které volně tuto databázi nesdílí. Proto jsem se rozhodl, že databáze čárových kódů se bude tvořit komunitou. Čárové kódy a informace o produktu budou uloženy na serveru a sdíleny mezi uživateli.

Čtvrtým logickým celkem je databáze receptů. Uživateli na základě produktů, které má přidané v aplikaci, budou zobrazeny recepty, jenž je možné s těmito potravinami připravit. Tato klíčová funkcionalita by měla napomoct problému velkého množství nespotřebovaných potravin.

4.2.2 **Drátěný model (Wireframe)**

K návrhu drátěných modelů byla použita webová aplikace Ninjamock. Aplikace je určena pro vytváření návrhů mobilních aplikací. Umožňuje nám vybrat si mezi mobilními platformami Android, iOS a windows phone a surface, nebo www stránkami. Po výběru platformy ještě lze vybrat v pravé části aplikace, konkrétní typ zařízení (zde byl použit iPhone 8). Při vytváření návrhu aplikace pracuje s grafickými prvky a elementy uživatelského rozhraní, které jsou umístěny v levé části aplikace v rozbalovacím menu a jednoduchým tažením myší se dají přesunout na plátno. Komponenty lze upravovat, měnit jim vlastnosti např. velikost, barvu, u textu barvu písma atd. vše po vzoru uživatelského rozhraní a to vše v pravém panelu webové aplikace. Aplikace si zachovává rozpracovaný projekt - pokud by došlo k zavření webového prohlížeče, k projektu je možné se znovu vrátit v bodu, kde byl uzavřen. Po vytvoření uživatelského účtu, lze projekty ukládat do souboru nebo případně sdílet na sociálních sítích a přes email. (viz příloha č. 3 – č. 9)

4.2.2.1 Přihlášení uživatele

Obrazovka přihlášení uživatele byla navržena na základě porovnání s ostatními aplikacemi konkurence. Na stránce v horní části je logo aplikace zarovnané horizontálně na střed. Pod logem aplikace následují dva text boxy, první je určený pro vyplnění emailu a druhý pro vyplnění hesla. Pod text boxy se nachází tlačítko pro přihlášení přes email. Pokud jsou oba texty správně vyplněny a uživatel stiskne toto tlačítko, dostává se na hlavní stranu aplikace. Pod ním následuje oddělující text „nebo“, který dává na výběr další možnost. Pod ním je umístěno tlačítko pro přihlášení přes sociální síť Facebook. Po jeho stisknutí se uživatel přesměruje, pomocí „Facebook Authentication“ pro iOS, na sociální síť. Po úspěšném dokončení přihlášení se uživatel dostane na hlavní stranu aplikace.

4.2.2.2 Lednice

Obrazovka Seznam produktů, neboli „Lednice“, slouží jako hlavní obrazovka aplikace. Hlavní logické celky aplikace „Lednice“ a „Recepty“, jsou zobrazovány pomocí TabBar, neboli je to lišta se záložkami, pomocí které se uživatel přesouvá mezi jednotlivými obrazovkami. Aby byl splněn požadavek společnosti Apple TabBar by měl oddělovat jednotlivé sekce aplikace.

Na této obrazovce je seznam přidanným produktů. Každý produkt obsahuje název, datum expirace a kategorii v podobě obrázku kategorie. Produkty jsou řazené podle data expirace. Po kliknutí uživatele na produkt se dostane na „Úpravu produktu“, ve kterém je možné upravit jeho množství atd.

Na obrazovce v navigační liště v horní části, se nachází tlačítko pro přidání produktu. Po jeho stisknutí se uživatel dostane na obrazovku „Přidání produktu“.

4.2.2.3 Přidání a úprava produktu

Obrazovky pro přidání a úpravu produktů, budou využívat jednotný design s mírnými odlišnostmi. Tato obrazovka Obsahuje v horní části ikonu pro zvolenou kategorii potravin a tlačítko skenovat. Ikona kategorie se změní vždy po změně text boxu určeného pro kategorii. Pomocí tlačítka skenovat se uživatel dostane na samostatnou obrazovku, která v sobě obsahuje funkcionalitu kamery, pomocí které detekuje čárový kód na produktu. Po úspěšné detekci se vrátí zpět na obrazovku „Přidání produktu“. Kde v text boxu čárový kód bude vyplněný detekovaný kód. Uživatel musí vyplnit název produktu, expirační dobu, množství a měrné jednotky. Pokud je v databázi čárový kód již obsažen, název, jednotky a kategorie se vyplní společně s čárovým kódem. Po vyplnění, všech text boxů má uživatel možnost přidat produkt pomocí tlačítka „Přidat“. Po úspěšném přidání produktu je uživatel vrácen na předchozí obrazovku „Lednice“. Rozdílem „Úpravy produktu“ je absence tlačítka skenovat a tlačítko ve spodní části sloužící pro uložení má popis „Upravit“.

4.2.2.4 Recepty

Obrazovka „Recepty“ slouží pro zobrazení seznamu receptů, dostupných s aktuálním stavem potravin v lednici. Na tuto obrazovku se uživatel dostane po klinutí na záložku „Recepty“ v TabBaru. Každý recept v seznamu obsahuje obrázek receptu, název, kategorii a dobu přípravy. Byla zvolena větší výška řádku oproti „Lednici“ a to z důvodu upoutání uživatelské pozornosti na obrázku a popisu receptu, které se zlepší znatelnějším oddělením buněk od sebe pomocí obrázku po celé délce buňky. Po kliknutí na buňku receptu se uživatel dostane na obrazovku „Detail receptu“.
Do budoucna je zamýšlena implementace filtrování receptů podle dostupnosti, kategorií či potravin.

4.2.2.5 Detail receptu

Obrazovka detail receptu zobrazí uživateli veškeré informace o receptu. Název receptu je umístěn v navigační liště. Pod ním následuje obrázek sloužící jako ilustrace receptu. Následuje seznam surovin, tedy produktů potřebných v receptu. Položky v seznamu surovin budou obsahovat vždy množství a název produktu. Pod seznamem produktů je již doba přípravy a popis postupu receptu. Pomocí navigační lišty lze jít na předchozí obrazovku. Postup i seznam produktů jsou skrolovatelné, je tedy možnost si zobrazit větší množství informací.

4.3 Grafický návrh

Grafický návrh je vytvořen na základě „Human Interface Guidelines“ od společnosti Apple. Výraznou barvou v aplikaci, která slouží pro aktivní prvky aplikace a pozadí na přihlašovací obrazovce, byla zvolena barva modrá, konkrétně RGB (0,150,255) (#0096FF v hexadecimálním zápisu). Rozhodováno bylo mezi zelenou a modrou barvou. Pro modrou barvu bylo rozhodnuto na základě porovnání designu testovací skupinou, čítající 7 lidí. Použité ikony a obrázky v aplikaci (assety) podléhají open source licenci. Jsou tedy volně šiřitelné a vhodné i ke komerčnímu užití. Samozřejmě některé ikony byly upraveny pro použití v aplikaci.

První obrazovkou, kterou uživatel v aplikacích iOS uvidí hned po instalaci, je tzv. Launch screen. Launch screen je statická obrazovka, která slouží pro časový interval spuštění aplikace a načtením jejích dat. Pro Launch screen v diplomové aplikaci byla zvolena obrazovka s modrým pozadím a bílým logem aplikace, které se využívá i na ikoně aplikace. Logo je tvořeno kuchařskou čepicí a knírkem pod ní. Logo má znázorňovat šéf kuchaře, konkrétně uživatele přivést na myšlenku, že aplikace je určena pro vaření, které je hlavní myšlenkou aplikace. Aplikace nese název „Don’t Waste“ jedná se o slovní spojení v angličtině s významem „Neplýtvej“. Má v uživateli vzbudit úvahu nad plýtváním potravin.

Další obrazovky již odpovídají rozložením prvků i logikou „Logickému návrhu“ z kapitoly „Logický návrh“. Použité prvky grafiky jsou nativní komponenty iOS aplikací z Cocoa vrstvy iOS architektury. Prvky grafiky obsahují sadu tří barev. Již zmíněná modrá, šedá, bílá a černá. Šedá barva je použita pro popisky jednotlivých komponent a neaktivní prvky v aplikaci. Bílá barva je určena pro pozadí v hlavních částech obrazovky a také

pozadí jednotlivých komponent, kromě navigační lišty a přihlašovací obrazovky. Černá barva slouží pro textové pole a boxy v aplikaci. (viz příloha č. 10 – č. 14)

4.4 Implementace

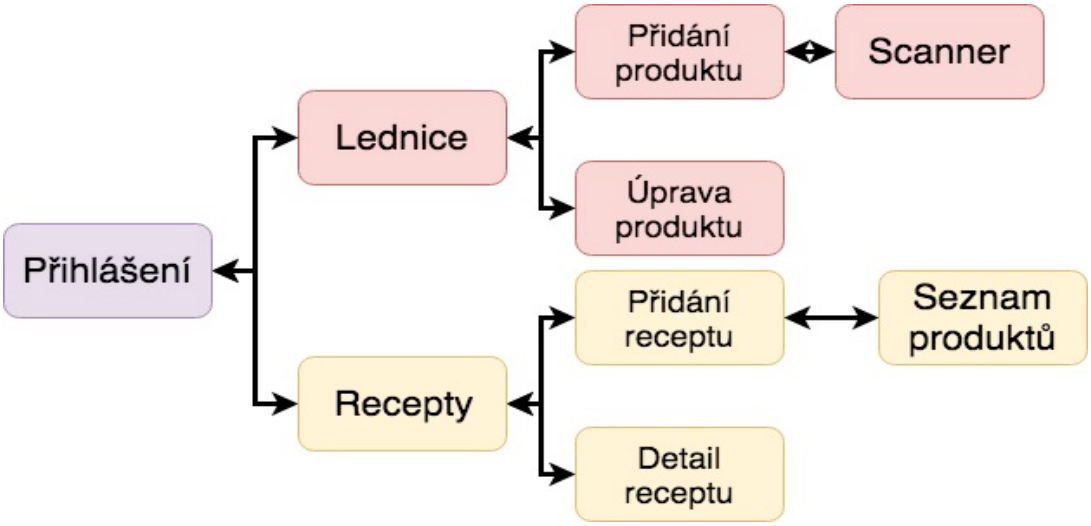
4.4.1 Seznámení s IDE Xcode

Jak bylo zmíněno v komparaci vývojových prostředí. IDE Xcode je zcela zdarma a dostupný je pouze v operačním systému macOS od Apple. Pokud není vývojář již vlastníkem zařízení s macOS, nemá jinou možnost než zvolit nainstalování macOS systému pomocí virtuálního prostředí např. Oracle VMVirtualBox nebo VMWare player. Bohužel virtuální operační systém nedosahuje zdaleka takového výkonu jako zařízení od společnosti Apple, ale při absenci těchto zařízení je pro vývoj v IDE Xcode nezbytný.

IDE Xcode lze nainstalovat přímo z aplikace AppStore (viz příloha č. 1). Xcode zabírá místo na pevném disku zhruba 4 GB. Při spuštění Xcode lze zvolit vytvoření nového projektu, playground a otevření existujícího projektu. V pravém panelu jsou vidět projekty, které byli nedávno spuštěny. Při vybrání nového projektu je poskytnut výběr operačních systémů pro které bude projekt, tedy námi tvořená aplikace, vytvořena. Na výběr je iOS, OS X a WatchOS. Při vytváření iOS aplikace lze upřesnit, pro jaký typ zařízení se aplikace vytváří. Pro tuto aplikaci byl vybrán ty „iPhone“, který podporuje grafické rozhraní pouze mobilních zařízení iPhone a iPod Touch. Aplikace je optimalizována pro všechny podporované typy těchto dvou zařízení a pomocí emulátoru je možné ji testovat.

4.4.2 Návrh architektury

Pro aplikaci byla zvolena architektura MVVM+. Jedná se o architekturu MVVM zvolenou v kapitole „Komparace architektury“, obohacenou řídící vrstvu. Tato vrstva vyplývá např. z architektury VIPER a její vrstvy Router. V tomto případě je vrstva nazvaná „Navigator“, někdy bývá nazývána „Coordinator“, nebo „FlowController“. Jedná se o vrstvu, která slouží jako navigátor mezi jednotlivými obrazovkami aplikace a jejich závislostmi. Nejprve je nutné architekturu sestavit logicky podle funkcionalit a obrazovek.



Obrázek 10 - Diagram rozdělení funkcionalit
[Vlastní zpracování]

Na výše uvedeném diagramu můžeme vidět obrazovky aplikace a přechody mezi nimi. Přechody jsou reprezentovány černými šipkami. Tyto přechody má na starosti „Navigator“.

Barevně jsou rozlišeny jednotlivé proudy aplikace, do kterých budou jednotlivé třídy a komponenty rozděleny pro rozšíření přehlednosti kódu.

4.4.3 Dependency injection

Pro dobrou testovatelnost aplikace iOS je velmi důležité oddělit od sebe jednotlivé komponenty aplikace tak, aby se co nejvíce snížil počet silných referencí na objekty. Silnou referencí na objekt dosáhneme jednoduchou deklarací a inicializací objektu v používané třídě. Silná reference vede k tomu, že objekt je stále uložen v paměti a není vymazán. Tím že v aplikaci je instance UITabBarController (dále jen TabBar), jenž má tu vlastnost, že drží referenci na UIViewControllery v ní inicializované, vede k tomu, že všechny instance ViewModel, Service a UIView, mají silnou referenci na své UIViewControllery, které jsou v TabBaru obsaženy. To vede k žití jednotlivých komponent po celou dobu používání aplikace a to samozřejmě má efekt na výpočetní výkon a využití paměti aplikace. Dependency injection, neboli vkládání závislostí, umožňuje odstranit tyto reference a tak pouze referenci na „Container“. Container je dá se říci kontejner pro instance jednotlivých objektových tříd v aplikaci. Který na požádání vrátí instanci na základě žádaného typu. Pro Dependecy injection byl použit Framework

„Swinject“. Jednotlivé instance musí být v Containeru zaregistrovány, poté mohou být použity kdekoliv v aplikaci pomocí funkce resolve().

Příklad registrace modelové vrstvy aplikace, konkrétně služeb.

```
// Registrace Modelové vrstvy aplikace
    func createServiceDependencies() {

        container.register(FacebookLoginService.self) { _ in
            FacebookLoginService()
        }
        container.register(FirebaseAuthService.self) {
            _ in FirebaseAuthService()
        }
        container.register(FirebaseService.self) {
            _ in FirebaseService()
        }
    }
```

Tato funkce createServiceDependencies() je volána při inicializaci Containeru.

```
public class DIContainer {

    static let shared = DIContainer()

    var container = Container()

    init() {

        createServiceDependencies()
        createViewModelDependencies()
        createViewControllerDependencies()
        Container.loggingFunction = nil
    }
}
```

DIContainer využívá návrhový vzor jedináček a to vzhledem k použitelnosti Containeru, který se využívá napříč aplikací. Jedná se tedy o jedinou instanci Containeru v aplikaci.

Příklad funkce resolve(), která vrací instanci objektu:

```
let fridgeListVC = DIContainer.shared.container.resolve(FridgeListVC.self)!
```

Díky vkládání závislostí přes Container je možné zamezit inicializaci jednotlivých závislostí v prezentační vrstvě.

Příklad:

```
var viewModel = RecipeListVM(firebaseService: FirebaseService())
```

Tento krok zhoršuje testovatelnost daného ViewControlleru. Objekt viewModel musí být nahrazen mockovanou verzí. S použitím Dependency Injection (dále jen „DI“), je možné instanci jednoduše nahradit mockovaným objektem stejného typu.

Příklad deklarace viewModel v UIViewController třídě:

```
var viewModel: RecipeListVM?
```

Funkce resolve() UIViewControlleru bude vypadat následovně:

```
let scannerVC = DIContainer.shared.container.resolve(ScannerVC.self)!
```

4.4.4 Navigator

Jedná se o řídící vrstvu, která má na starost pohyb uživatele v aplikaci. Tedy navigace mezi jednotlivými obrazovkami aplikace.

```
struct Navigator {

    var fridgeNavigationController: UINavigationController?
    var recipesNavigatorController: UINavigationController?

    init(
        fridgeNavigationController: UINavigationController? = nil,
        recipesNavigatorController: UINavigationController? = nil
        ) {

        self.fridgeNavigationController = fridgeNavigationController
        self.recipesNavigatorController = recipesNavigatorController
    }
}
```

Navigator obsahuje UINavigationController pro oba logické proudy aplikace. Prvním je Lednice a druhým Recepty. Příklad funkce Navigatoru:

```
func toCreateProducts() {

  let scannerVC = DIContainer.shared.container.resolve(ScannerVC.self)!

   fridgeNavigationController?.pushViewController(
       scannerVC,
       animated: true
    )
}
```

Výše uvedená funkce slouží pro zobrazení obrazovky „Přidat produkt“ zastoupená třídou „ScannerVC“, tato instance se přidá do pole „viewControllers“ a vykreslí pomocí UINavigationController a jeho funkce pushViewController(). Navigator využívá metodu knihovny UIKit pro zobrazování aktuální obrazovky.

4.4.5 **Model**

4.4.5.1 Entity

Modelová vrstva slouží pro komunikaci se serverem a obsahuje Entitní třídy používané v aplikaci. V tomto případě se jedná o entity Product a Recipe. Pokud aplikace přijímá data ze serverové strany, server obvykle vrací data typu JSON. Aplikace vrácená data zachytí a je potřeba je převést na datové typy, využívané v aplikaci. Například pokud se jedná o seznam produktů. Odpověď ze serveru bude JSON objekt, tedy slovník typu [String:Any].


```
{
 "-L8SKWk-YcGFeFdEJ3cl" : {
  "barCode" : 122413424242342342,
  "category" : "Dorty, trvanlivé pečivo, jemné pečivo a sušenky",
  "expirationDateFrom" : 1522070700,
  "expirationDateTo" : 1522070700,
  "measurementUnit" : "g",
  "name" : "Chléb",
  "quantity" : 300
 },
 "-L8SKYMURhqw2MVamcOY" : {
  "barCode" : 122413424242342342,
  "category" : "Dorty, trvanlivé pečivo, jemmé pečivo a sušenky",
  "expirationDateFrom" : 1522070700,
  "expirationDateTo" : 1522070700,
  "measurementUnit" : "g",
  "name" : "Chléb",
  "quantity" : 300
 }
}
```


Tento objekt je potřeba převést na pole datového typu Product. K tomu slouží framework ObjectMapper. Pokud odpovídá protokolu (rozhraní) Mappable. Lze odpověď ze serveru převést na námi určený datový typ. Příklad pomocí funkce mapping():

```
func mapping(map: Map) {

    id <- map[Product.firebaseIdKey]
    name <- map["name"]
    expirationDateFrom <- (map["expirationDateFrom"], DateTransform())
    expirationDateTo <- (map["expirationDateTo"], DateTransform())
    barCode <- map["barCode"]
```

```
    quantity <- map["quantity"]
    measurementUnitString <- (map["measurementUnit"],
EnumTransform<MeasurementUnit>())
    categoryString <- (map["category"], EnumTransform<Category>())
}
```

Mapování pomocí ObjectMapperu je parsování JSON objektu podle názvu klíčů. Poté jsou jednotlivé hodnoty převedené na určené datové typy např. String, Int apod., ale i vytvořené datové typy, které odpovídají protokolu Mappable.

Modelová vrstva dále obsahuje třídy nazvané „Service“. Jedná se o třídy, které obsahují logiku pro komunikaci se serverovou částí. Konkrétně reaktivní řešení požadavků na server. V případě této aplikace je serverová část řešena pomocí databáze Firebase.

4.4.5.2 Databáze Firebase

Pro serverové řešení byla použita webová aplikace Firebase od společnosti Google. Její výhodou je jednoduchost v ovládání i bez znalostí programování aplikace na straně serveru. Zároveň je Firebase již zabezpečená proti případným útokům a podporuje jazyky Swift, Java, Javascript a Objective-C, pro které je vytvořená také dokumentace viz zdroj. Obsahuje pro aplikaci klíčovou iOS SDK. Ve Firebase bylo potřeba aplikaci nejdříve nastavit, pomocí jednoduchého nastavení a návodu na stránce. V nastavení aplikace v záložce Login and Auth, byla povolena autentikace pomocí emailu a Facebooku, která je pro nás klíčová v přihlášení uživatele do aplikace. Firebase komunikuje s API třetích stran a k Facebook API bude přistupovat z důvodu autentikace uživatelů. Databáze je zastoupena použitím platformy Firebase od společnosti Google. Jedná se o cloudové řešení server-side aplikace určené především pro mobilní aplikace. Data v databázi jsou uložena a zašifrována. Jediný možný přístup je pomocí autorizačního tokenu, který je generovaný na zařízení a je spojený s aplikací. Tento token se ukládá do lokálního uložiště telefonu UserDefaults. (viz příloha č. 19)

4.4.5.3 Přihlášení pomocí Facebook

Pro autentikaci uživatele v aplikaci pomocí Facebooku je nutné využít Facebook API, tzn. Rozšíření rozhraní aplikace pro přístup k Facebook službám. Vše co bylo potřeba postupovat podle dokumentace jak Facebooku tak Firebase. Na stránce developers.facebook.com, je detailní návod jak využít právě Facebook API, je nutné se přihlásit na facebookový účet nebo účet vytvořit. Na developerském účtu Facebooku je

potřeba vytvořit aplikace u které se zvolí pouze jméno a zaměření aplikace. V tomto případě bylo zvoleno „Zdraví a fitness“. Poté bylo zvoleno určení aplikace a to konkrétně pro platformu iOS.

V nastavení je potřeba vyplnit v sekci Security redirect OAuth2.0 url a Bundle ID aplikace, Bundle ID je stejné jako Bundle Identifier v IDE Xcode. Které je dostupné ve Firebase jako URL adresa aplikace. V sekci Status and Review je potřeba povolit možnost veřejného publikování. Ve Firebase je v sekci Login and Auth potřeba vyplnit Facebook App secret a Facebook App ID, které bylo získáno z nastavení Facebook App. Pomocí CocoaPods aplikace byli frameworky Firebase a Facebook společně nainstalovány. Dalším krokem byla úprava souboru Info.plist v IDE Xcode a poté do něj byl vložen kód pro přístup k API (viz příloha č. 15). V souboru AppDelegate.swift byly importovány knihovny z Facebook SDK a přístup k Facebook API byl aktivován ve funkci (viz přílohač. 16).

S využitím frameworku RxSwift, FacebookLogin a FacebookCore, byla implementována FacebookLoginService. Pomocí její funkce facebookLogin() získáme sekvenci, kterou ve ViewModelu použijeme a napojí se na uživatelskou akci kliknutí na tlačítko „Přihlásit se přes Facebook“.

```
class FacebookLoginService {

  var facebookLoginManager: LoginManager?
  func facebookLogin() -> Observable<(FacebookLoginStatus,
AccessToken?)> {
        facebookLoginManager = LoginManager()
        return Observable.create { [weak self] observer in
            self?.facebookLoginManager?.logIn(readPermissions:
[.publicProfile]) { loginResult in
                switch loginResult {
                case .failed(let error):
                    print(error)
                    observer.on(.error(error))
                case .cancelled:
                    print("User cancelled login.")
                    observer.on(.next((.cancelled, nil)))
                case .success(_, _, let accessToken):
                    print("Logged in!")
                    observer.on(.next((.success, accessToken)))
                }
            }
            return Disposables.create()
        }
    }
}
```

Jak frameworky Facebook tak Firebase, nejsou implementovány reaktivně. Bylo zde potřeba vytvořit pro tyto frameworky reaktivní rozšíření. Všechna rozšíření jsou obsažena ve složce „Extensions“ v projektu. (viz příloha č. 20)

4.4.5.4 Přihlášení přes email

Přihlášení pomocí emailu a hesla je zprostředkováno skrz autentizaci Firebase. Podobně jako u přihlašování přes Facebook, tak i zde je autentizace prováděna skrz otevřený bezpečnostní protokol pro autentizace OAuth2.0. Správu uživatelských účtů lze provádět skrz konzoli aplikace Firebase. Firebase autentizace řeší i případ pokud je uživatel již zaregistrován, či zapomněl heslo, nebo heslo zadal špatně. Tyto případy lze v aplikaci pomocí FirebaseAuth frameworku řešit. Každému uživateli je vygenerován unikátní token pro přihlášení k aplikaci. Tento token je svázaný se zařízením a uživatelským účtem. Pro tento typ autentizace byla v aplikaci vytvořena třída s názvem FirebaseAuthService. (viz příloha č. 21)

Příklad rozšíření pro FirebaseAuth framework.

```
func rx_signinWithEmail(email: String, password: String) ->
Observable<User?> {
        return Observable.create { observer in
            self.signIn(withEmail: email, password: password)
            { (user, error) in
                if let error = error {
                    observer.onError(error)
                } else {
                    observer.onNext(user)
                    observer.onCompleted()
                }
            }
            return Disposables.create()
        }
    }
```

Příklad přihlášení pomocí emailu FirebaseAuthService:

```
func signIn(email: String, password: String) -> Observable<User?> {

 return Auth.auth().rx_signinWithEmail(email: email, password: password)
 }
```

Dalším případem přihlášení uživatele je i jeho registrace. Ta je řešena obdobným způsobem. Aplikace nejdříve ověří, jestli je účet již v databázi, a pokud není, účet aplikace na serverové straně vytvoří a uživatele přihlásí. Příklad rozšíření funkce createUser()

```
func rx_createUserWithEmail(email: String, password: String) ->
Observable<User?> {
        return Observable.create { observer in
            self.createUser(withEmail: email, password: password,
completion: { (user, error) in
                if let error = error {
                    observer.onError(error)
                } else {
                    observer.onNext(user)
                    observer.onCompleted()
                }
            })
            return Disposables.create()
        }
    }
```

Příklad FirebaseAuthService:

```
func createUser(email: String, password: String) -> Observable<User?> {

      return Auth.auth().rx_createUserWithEmail(
            email: email, password: password
      )
 }
```

4.4.5.5 Lednice a recepty

Lednice a recepty využívají stejnou modelovou vrstvu, jedná se konkrétně o třídu FirebaseService. Tato třída implementuje funkce FirebaseDatabase. Tato třída odpovídá protokolu FirebaseServiceType. Používá reaktivní rozšíření pro třídy DatabaseQuery a DatabaseReference. DatabaseQuery funkce slouží pro poslouchání sekvence dat v databázi. Např. sleduje seznam uživatelských produktů - pokud nastane změna na funkci, zavolá se subscribe() a lokální data se na základě této akce aktualizují. DatabaseReference funkce umožňují aplikaci ukládat, mazat a měnit data uložené v databázi na základě uvedených klíčů. Můžeme si všimnout, že reaktivní implementaci pouze obaluje návratové hodnoty funkce typu Void (callback). Pomocí vytvoření sekvence Observable.create a funkce observer.onNext a observer.onComplete definují událost, která se v sekvenci vyvolá.

Příklady funkcí DatabaseQuery protokolu Reactive, sledování jednotlivé sekvence:

```
func observeSingleEventOfType(of eventType: DataEventType) ->
Observable<DataSnapshot> {
        return Observable.create { observer in
            self.base.observeSingleEvent(of: eventType) { (snapshot) in
                observer.onNext(snapshot)
                observer.onCompleted()
            }
```

```
            return Disposables.create()
        }
    }
```

Příklad funkce pro sledování záznamu v databázi:

```
func observe(eventType: DataEventType) -> Observable<DataSnapshot> {
      return Observable.create({ observer in
          let handle = self.base.observe(eventType, with: { snapshot in
              observer.onNext(snapshot)
          })

          return Disposables.create {
              self.base.removeObserver(withHandle: handle)
          }
      })
    }
```

Příklad funkce změny záznamu DatabaseReference protokolu Reactive:

```
    func updateChildValues(values: [AnyHashable: Any]) ->
Observable<DatabaseReference> {
        return Observable.create { observer in
            self.base.updateChildValues(values) { (error, reference) in
                if let error = error {
                    observer.onError(error)
                } else {
                    observer.onNext(reference)
                    observer.onCompleted()
                }
            }

            return Disposables.create()
        }
    }
```

Příklad funkce uložení záznamu do databáze:

```
    func setValue(value: Any?, andPriority priority: Any? = nil) ->
Observable<DatabaseReference> {
        return Observable.create { observer in
            self.base.setValue(value, andPriority: priority) { (error,
reference) in
                if let error = error {
                    observer.onError(error)
                } else {
                    observer.onNext(reference)
                    observer.onCompleted()
                }
            }

            return Disposables.create()
        }
```

```
    }
```

A tato rozšíření jsou připravena k použití aktuálních případů. Níže je uveden případ implementace databáze firebase pomocí RxSwift. Máme funkci getProducts(), která vrací sekvenci produktů podle unikátního id uživatele.

```
func getProducts() -> Observable<[Product]> {
        return observeValueArray(path: "\(Auth.auth().currentUser?.uid
?? "")").debug()
    }
```

Funkce createProduct s parametrem product vytváří v databázi záznam, kde je vidět stromová hierarchie FirebaseDatabáze pomocí potomků (child), které jsou zastoupeny klíčem typu String. Pomocí potomků v databázi se lze dostat na jednotlivé záznamy. Jedná se o objektově orientovanou databázi.

```
    func createProduct(product: Product) -> Observable<Void> {
        let values = Mapper<Product>().toJSON(product)

        return database
            .child(Auth.auth().currentUser?.uid ?? "")
            .childByAutoId()
            .rx.updateChildValues(values: values)
            .mapToVoid()
    }
```

4.4.6 ViewModel

V aplikaci ViewModel slouží jako hlavní prvek architektury. Zpracovává akce a sekvence z prezentační vrstvy a reaguje na ně pomocí aktualizace datové vrstvy. Je to tedy prostředník mezi těmito dvěma vrstvami. Každý ViewModel v aplikaci odpovídá protokolu ViewModelType:

```
protocol ViewModelType {

    associatedtype Input
    associatedtype Output

    func transform(input: Input) -> Output
}
```

Protokol ViewModelType vyžaduje v každém modelu Input, Output a funkci transform(). U ViewModelu se jedná o transformaci uživatelského vstupu na výstup

pracující s datovou vrstvou aplikace. Do ViewModelu jsou vloženy potřebné závislosti datové vrstvy. (viz příloha č. 17)

Příklad:

```
private let firebaseService: FirebaseService

init(firebaseService: FirebaseService){
        self.firebaseService = firebaseService
    }
```

Každý ViewModel kromě závislostí Service, tedy datové vrstvy, obsahuje závislosti Navigatoru, pomocí kterého řídí přesun mezi jednotlivými obrazovkami aplikace. Příklad:

```
var navigator: Navigator?
```

Funkce datové vrstvy se poté využívají v transformační funkci. Uživatelské inputy jsou deklarovány v podobě struktur stejně jako Outputy. Obsahují jednotlivé sekvence nabindované (napojené) na uživatelské akce v prezentační vrstvě. Příklad:

```
    struct Input {
        let trigger: Driver<Void>
        let createPostTrigger: Driver<Void>
        let selection: Driver<IndexPath>
    }
    struct Output {
        let fetching: Driver<Bool>
        let products: Driver<[Product]>
        let createPost: Driver<Void>
        let selectedPost: Driver<Product>
        let error: Driver<Error>
    }
```

Transformační funkce pomocí operátorů reaktivního programování převede sekvence vstupů na výstupy. Příklad transformační funkce bez Error handlingu:

```
    func transform(input: Input) -> Output {
        let products = input.trigger.flatMapLatest {
            return self.firebaseService.getProducts()
                .trackActivity(activityIndicator)
                .debug()
                .asDriverOnErrorJustComplete()
        }
        let selectedPost = input.selection
         .withLatestFrom(products) { (indexPath, products) -> Product in
                return products[indexPath.row]
            }
            .do(onNext: navigator?.toProduct)
        let createPost = input.createPostTrigger
            .do(onNext: navigator?.toCreateProducts)

        return Output(products: products,
```

```
                    createPost: createPost,
                    selectedPost: selectedPost)
    }
```

4.4.6.1 Driver

ViewModel i View používají sekvence typu Driver. Driver je zvláštní případ Observable. Jedná se o velmi praktickou sekvenci. Jeho záměrem je poskytnout intuitivní způsobu psaní reaktivního kódu ve vrstvě uživatelského rozhraní nebo v případě, kdy chcete modelovat datový tok napříč aplikací. Hlavním důvodem proč byl vytvořen Driver, je řízení aplikace pomocí sekvencí datové vrstvy. Driver slouží pro napojení (Binding) prvků uživatelského rozhraní na Rx sekvence.

4.4.6.2 Error handling

Error handling je manipulace s chybami. V RxSwift každá sekvence, kromě události „Complete“, může nabývat události „Error“, tedy chyba. Chyba v horším případě může v aplikaci způsobit pád. Tomuto velice nechtěnému stavu lze předejít, správným nakládáním s chybami v aplikaci. Chyby pokud jsou správně zachyceny a následuje po nich reakce např. vypsání chyby do logu, či konzole, mohou být užitečné k odhalení nežádoucích stavů např. nedostupnost internetu, přetečení zásobníku, nebo chyba na serverové straně. Implementace error handlingu v aplikaci tvoří třída ErrorTracker. ErrorTracker implementuje funkci trackError.

```
func trackError<O: ObservableConvertibleType>(from source: O) ->
Observable<O.E> {

        return source.asObservable().do(onError: onError)
    }
```

Jedná se o generickou funkci, jejím parametrem je objekt odpovídající protokolu ObservableConvertibleType (sekvence Observable, Driver apod.). Pokud sekvence vrátí událost onError(), chyba se uloží do sekvence ErrorTrackeru. ErrorTracker jako sekvenci lze pak jednoduše použít v prezentační vrstvě pro zobrazení okna s chybou.

4.4.7 **View**

Prezentační vrstva aplikace slouží především pro implementaci grafického rozhraní aplikace a funkcí UIViewControlleru, pomocí kterého je možnost využití životního cyklu prezentační vrstvy. V případě RxSwift, tedy reaktivního programování, se zde využívá

framework RxCocoa. RxCocoa implementuje reaktivní rozšíření pro UI komponenty standartní knihovny Swift a UIKit. Umožňuje asynchronní poslouchání uživatelských akcí pomocí Driver a Observable instancí.
V aplikaci je hlavní část prezentační vrstvy tvořena pomocí UIViewController. Odpovídající protokolu RxViewControllerProtocol a StoryboardInit. Jednotlivé obrazovky v aplikaci jsou tvořené pomocí Storyboardu.

4.4.7.1 Storyboard

StoryBoard je interaktivní nástroj pro navržení uživatelského rozhraní. Storyboard obsahuje komponenty UIKit, které lze tažením myši přesunout na obrazovku. Tak aby byla zaručeno správné rozložení a velikost komponent na všech podporovaných zařízení, jsou komponenty uchyceny pomocí tzv. „Constraints“. Constraints lze využít například pro odsazení komponenty, zarovnání svisle i vodorovně nebo určení šířky a výšky komponenty. Storyboardy jsou s třídami UIKit propojeny referenčně, pomocí klíčů.
Pro využití storyboardu v aplikaci, je nutné pro vlastní třídy UIKit, inicializovat třídy pomocí získání instancí storyboardu. Implementace funkce pro inicializaci UIViewControlleru pomocí protokolu StoryboardInit (viz příloha č. 18):

```
    static func storyboardInit(_ storyboardName: String, viewModel:
ViewModel?) -> Self {
        var viewController = UIStoryboard(
            name: storyboardName,
            bundle: Bundle(for: self)
            ).instantiateViewController(
                withIdentifier: String(describing: self)
            ) as! Self

        viewController.viewModel = viewModel

        return viewController
    }
```

4.4.7.2 ViewController

ViewController jako jedna z hlavních komponent prezentační vrstvy odpovídá za komunikaci s ViewModelem. V této instanci se implementuje Binding uživatelského rozhraní na Drivery Inputu ViewModelu. Hlavním rozdílem při použití RxCocoa oproti UIKit je značné ušetření kódu. Při využití UIKit je nutnost použití funkcí delegátů

jednotlivých komponent. Například pro sestavení TableView je nutné při nejmenším implementovat tyto funkce:

```
func numberOfSections(in tableView: UITableView) -> Int
func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell
```

pomocí RxCocoa je implementace následovná:

```
tableView.rx.items(cellIdentifier:String:Cell.Type:_:)
```

Implementace je díky tomu velice snadná a přehledná. Použití „rx“ operátoru zastupuje Binding. Jedná se o napojení komponent uživatelského rozhraní na sekvenci. Kterou lze poslouchat (Observe).
ViewController odpovídající protokolu RxViewControllerProtocol příklad:

```
protocol RxViewControllerProtocol where Self: UIViewController {

    var disposeBag: DisposeBag { get set }

    func bindToVM()
}
```

Obsahuje dvě základní rozšíření a tím je funkce bindToVM() a disposeBag. BindToVM funkce obsahuje logiku spojující komponenty uživatelského rozhraní a ViewModel. Nachází se zde binding Driverů komunikující přes ViewModel s datovou vrstvou pomocí zmíněné transformační funkce.

4.5 Testování aplikace

Aplikace byla testována uživatelsky, používáním zhruba pěti uživatelů. Pomocí jejich zpětné vazby byly nalezeny chyby v aplikaci, které byly následně opraveny, například validace emailu a hesla na přihlašovací obrazovce, která byla opravena a byla vystavená nová verze s opravou. Na základě uživatelského používání byly provedeny i menší změny v designu.

4.5.1 Unit testy

Unit testování v aplikacích iOS obstarává XCTestCase. Jedná se o balíček určený k testování od společnosti Apple. Kromě iOS aplikací se využívá i v ostatních systémech např. macOS, watchOS a AppleTV. Unit testy jsou určené pro testování jednotlivých

komponent aplikace, pro zajištění jejich funkčnosti před vpuštěním do produkce. Unit testy v architektuře MVVM testují převážně datovou a ViewModel vrstvu. Testování reaktivně vyvinuté aplikace je založeno na testování jednotlivých sekvencí a jejich funkčnosti. Pro testování se využívají testovací data (Mock data). Tyto mockované data reprezentují, reálný stav chování aplikace pouze simulovaně. Díky tomu unit testy nevytváří nová data v databázi, či nezatěžují síť voláním požadavků na server. (viz příloha č. 22) Příklad:

```
class MockFirebaseService: FirebaseService {
    var posts_ReturnValue: Observable<[Product]> = Observable.just([])
    var posts_Called = false

    override func getPosts() -> Observable<[Post]> {
        posts_Called = true
        return posts_ReturnValue
    }
    override func createProduct(product: Product) -> Observable<Void> {
        var posts_Called = true
        return Observable.just()
    }
    override func updateProduct(product: Product) -> Observable<Void> {
        var posts_Called = true
        return Observable.just()
    }
}
```

S pomocí testovacích dat můžeme připravit testovací scénář pro testovaný objekt. Testovacím objektem byla zvolena vždy třída např. FridgeListVM (ViewModel Lednice). Která se inicializuje s testovacími daty pomocí funkce setUp() ze třídy XCTestCase. Příklad:

```
var mockedFirebaseService: MockFirebaseService!

var mockedNavigator: MockedNavigator!
var viewModel: FridgeListVM!
let disposeBag = DisposeBag!

override func setUp() {
    super.setUp()
    mockedFirebaseService = MockFirebaseService()
    mockedNavigator = MockedNavigator()
    disposeBag = DisposeBag()
    viewModel = FridgeListVM(useCase: mockedFirebaseService,
                             navigator: mockedNavigator)
}
```

Funkce setUp() se volá před vyvoláním každé testovací metody v testovacím scénáři. V případě testování třídy ViewModel, je potřeba vytvořit testovací inputy, které budou sloužit jako parametry transformační funkce. Příklad FridgeListVM inputu:

```
private func createInput(
    trigger: Observable<Void> = Observable.just(),
    createPostTrigger: Observable<Void> = Observable.never(),
    selection: Observable<IndexPath> = Observable.never()
    ) -> FridgeListVM.Input {

    return FridgeListVM.Input(
        trigger: trigger.asDriverOnErrorJustComplete(),
        createPostTrigger:
createPostTrigger.asDriverOnErrorJustComplete(),
        selection: selection.asDriverOnErrorJustComplete())
}
```

4.6 Komparace vyvinuté aplikace s možnými alternativami

Vyvinutá aplikace umožňuje organizaci jídla na základě jeho expirační doby. Pomocí nabízených receptů ze skladovaných potravin má aplikace za cíl snížit počet vyhozeného nespotřebovaného jídla a dát tak možnou inspiraci jaká jídla je možné s nespotřebovanými potravinami připravit. Tímto by měla zacílit skupinu uživatelů, kteří na základě této aplikace mohou ušetřit peníze za nákup nadbytečných potravin a ušetřit tak i životní prostředí. V obchodě AppStore, lze nalézt aplikace s podobným cílem, jako jsou Expiry, Prep & Pantry, Best Before a Freshbox. Všechny zmíněné aplikace pracují na podobném principu, jako vyvinutá aplikace, a tím je sledování expirační doby potravin, jenž uživatel naskenuje do aplikace. Rozdílem od těchto aplikací, je využití receptů. Využití receptů jako nástroj pro inspiraci uživatelů jak efektivně jídlo spotřebovat. Tato přidaná hodnota bude znatelnější při produkčním využití aplikace v obchodě AppStore. Čím větší uživatelská komunita aplikaci bude využívat, tím větší databáze receptů bude přístupná.

5 Výsledky a diskuse

5.1 Komparace architektury

Zvolení správné architektury pro iOS aplikace je stěžejní pro její údržbu a náklady na vývoj. Při chybném zvolení architektury se aplikace může stát při velkém rozsahu funkcionalit v podstatě neudržovatelnou. Oprava nalezených chyb by v tomto případě mohla být více nákladná, nežli je implementace nové funkcionality a přepsání aplikace do jiné architektury by náklady navýšilo v určitých případech i několikanásobně. Proto je potřeba výběr architektury řádně zvážit hned na začátku vývoje.
V komparaci architektury iOS aplikace byly vzaty k porovnání čtyři používané architektury: Cocoa MVC, MVP, MVVM a VIPER. Popsány byly jejich odlišnosti, především také výhody a nevýhody. Porovnávajícími kritérii byla zvolena čtveřice parametrů.

Prvním parametrem je „rozdělení odpovědností“, které určuje separaci jednotlivých vrstev architektury a jejích funkcionalit.

Druhým parametrem je „testovatelnost“. Testovatelnost je pro vývoj aplikace dalším stěžejním kritériem, pomocí testů v aplikaci, lze předejít chybám, které by se projevili v produkci, při uživatelském používání. Také testy napomáhají chyby v aplikaci najít, a tudíž vede k jejich rychlejší opravě.

Třetím parametrem je „snadné použití“. Snadné použití ve smyslu implementace a porozumění architektury novým programátorem. Tento parametr také velmi ovlivňuje nákladovost aplikace především ve vývojové fázi. Složitější architektury znatelně zvyšují množství napsaného kódu a tím strávený čas vývojáře jeho implementací. To znatelně zvyšuje nákladovost aplikace na vývoj.

Čtvrtým parametrem byla zvolena udržovatelnost. Udržovatelnost aplikace souvisí s produkční fází aplikace. Kdy je potřeba opravovat nalezené chyby, či měnit funkcionality v aplikaci. Udržovatelnost aplikace souvisí s rozsahem aplikace, ale i se zkušenostmi vývojáře, či logické návaznosti kódu.

Každá z porovnávaných architektur má své přednosti, ale také nedostatky. U Cocoa MVC (Model View Controller) se jedná o její snadné použití a rozsáhlou dostupnost materiálů. Nedostatky této architektury jsou testovatelnost aplikace a rozdělení odpovědností. Testovatelnost aplikace většího rozsahu je velice obtížná a to z důvodu

úzkého spojení prezentační a řídící vrstvy, instance třídy UIViewController obsahují veškerou logiku aplikace, díky tomu zde datová vrstva působí pouze v roli entitních tříd inicializovaných ve třídě UIViewController. Lze říci, že se zde prezentační a datová vrstva stávají netestovatelnými.

MVP (Model View Presenter) je architektura také snadná pro použití a částečně doplňuje nedostatky v rozdělení odpovědností a testovatelnosti, ovšem její nevýhodou je přítomnost prezentační vrstvy nazývané Passive View, která nesouměrně rozděluje odpovědnosti aplikace.

Dalším adeptem je architektura VIPER (View Interactor Presenter Entity Router). Jednotlivé vrstvy architektury VIPER jsou velmi dobře separované a rozdělení odpovědností se rozprostírá souměrně mezi pět vrstev, kde každá má svou přesně určenou funkci. Architektura VIPER je velice dobře testovatelná a udržovatelná. Jediným úskalím je její složitost a nutnost psaní velkého množství kódu. Vyžaduje tak zkušené vývojáře a není vhodná pro aplikace menšího rozsahu. Z tohoto důvodu nebyla architektura VIPER zvolena pro vývoj této aplikace

A posledním adeptem je architektura MVVM (Model View ViewModel). Tato architektura má také velice dobré rozdělení odpovědností mezi tři hlavní vrstvy, které zaručuje v tomto případě i dobrou testovatelnost. Jedná se o druhou nejpoužívanější architekturu v iOS a to díky její separaci a snadnému použití. Tato architektura se často používá při vývoji mobilních aplikací pomocí reaktivního programování, které je v této aplikace také využito. Při komparaci architektur podle daných parametrů byla vybrána architektura MVVM.

5.2 Logický a grafický návrh aplikace

Logický návrh aplikace probíhal na základě porovnání uživatelského rozhraní konkurenčních aplikací. Bylo připraveno rozvržení a koncepce jednotlivých komponent pro jednoduché a přehledné využití funkcionalit aplikace, pomocí metod „User Experience“. Pro zaručení snadného použití aplikace a pochopení funkcionalit cílovými uživateli, byly využité nativní komponenty poskytující přímo společnost Apple pro vývojáře s využitím knihovny UIKit. Nativní komponenty podléhají koncepci „Human Interface Guidelines“, jedná se o funkce a použití jednotlivých komponent na základě vize

společnosti Apple. Tyto komponenty Apple již využívá ve svých vlastních aplikací, se kterými jsou uživatelé již seznámeny a jsou zvyklí na jejich používání.

Grafický návrh aplikace vyplívá z logického návrhu, byly zde použity již reálné komponenty, ikony a obrázky. Aplikace využívá čtyři klíčové barvy. Modrou, šedou, bílou a černou barvu. Tak aby byl zaručen jednotný vzhled aplikace. Aktivní prvky aplikace a ikony jsou v modré barvě. Pasivní prvky aplikace jsou v šedé barvě a zadaný text uživatele a nadpisy jednotlivých položek jsou v černé barvě.

5.3 Implementace

Prvním krokem, před implementací kódu, byl návrh zvolené architektury MVVM. Návrh architektury vyplývá z poznatků knihy „Clean Architecture“. Architektura MVVM byla rozšířena o koordinátor aplikace nazvaný „Navigator“, jedná se o vrstvu aplikace, která zodpovídá za přechody mezi jednotlivými obrazovkami. S použitím navigační vrstvy se jedná o architekturu nazvanou MVVM+.

Model zodpovídá za datovou vrstvu aplikace, obsahuje třídy komunikující se serverem a entitní třídy.

ViewModel slouží jako řídící prostředník. Jeho hlavní funkce je transformovat vstupy (uživatelské akce) na výstupy (požadavek na server). Tedy pomocí transformační funkce napojuje uživatelské rozhraní na datovou vrstvu, které na sebe navzájem reagují. Další funkcí ViewModelu je využití instance Navigatoru a jeho funkcí pro přechod mezi jednotlivými obrazovkami.

View zodpovídá za implementaci uživatelského rozhraní a životní cyklus jednotlivých obrazovek. Na reakci uživatelských akcí se využívají sekvence typu Driver. Jedná se o sekvence sloužící pro správu uživatelských akcí, které se pomocí ViewModelu propojí se sekvencemi Modelu. Architektura byla rozdělena do tří logických větví zastupujících hlavní funkcionality aplikace.

Prvním je „Přihlašování“, které využívá autentizaci uživatele pomocí emailu nebo sociální sítě. Autentizace probíhá pomocí frameworku FirebaseAuth, jedná se o cloudové řešení server side aplikace, určené především pro mobilní aplikace, kromě přihlašování uživatele Firebase poskytuje i real-time databázi. Při implementaci přihlašování byla

použita integrace API sociální sítě Facebook a pro validaci jména a hesla byly použité regulární výrazy.

Další funkcionalitou je „Lednice“, která zodpovídá za řízení skladovaných potravin, tedy nese informace o skladovaných potravinách, především o jejich expiraci. Další funkcionalitou pod touto větví je funkce skenování produktu. Jedná se o skenování čárového kódu produktu a následné přidání produktu do databáze. V této větvi byla implementována komunikace s databází Firebase. Konkrétně, načtení produktů z databáze, jejich změna a uložení nového produktu.

Implementace třetí části aplikace „Recepty“ byla zajištěna obdobným způsobem, jako je tomu u „Lednice“. Pracuje především s daty databáze Firebase. Recepty se zobrazují na základě skladovaných produktů.

Při implementaci byla použita „Dependency injection“, vkládání závislostí. Pomocí vkládání závislostí bylo docíleno lepší zprávy paměti díky odstranění silných referencí mezi jednotlivými vrstvami objektů. A také znatelně lepší separaci jednotlivých komponent, které vyžadují inicializaci jednotlivých závislostí vždy v dané třídě. Díky vkládání závislostí se znatelně zjednoduší testování, kde se místo dané závislosti dosadí testovací objekt stejného typu. [13]

5.4 Testování

Aplikace byla otestovaná skupinou pěti uživatelů. Uživatelské testování sloužilo jak pro nalezení chyb, tak případných grafických změn aplikace. Nalezena byla chyba při validaci hesla při přihlašování uživatele, která byla následně odstraněna. Dalším doporučením je změna přidávání receptu, které na mobilním zařízení není příliš efektivní, bude potřeba implementace webové aplikace určené minimálně pro přidávání nových receptů.

Kromě uživatelského testování byly v aplikaci použity unit testy. Sloužící pro otestování správné funkcionality komponent v aplikace. V aplikaci jsou testy pro ViewModel a Model aplikace. Tedy řídící a datovou vrstvu. Pro prezentační vrstvu jsou vhodnější UI testy, jejich implementace díky uživatelskému testování není potřeba a pro funkcionalitu aplikace nehrají klíčovou roli. Unit testy se zaměřují na vytvořené sekvence, pro které jsou vytvořeny testovací data.

5.5 Zamyšlení a zhodnocení aplikace

Vyvinutá aplikace diplomové práce se nevyhnula bez chyb. Při implementaci a testování bylo nalezeno pár nesrovnalostí, především při implementaci receptů, které pracují s jednotlivými produkty, na základě jejich seznamu v receptu, tyto produkty se využívají především pro jejich zobrazování založené na aktuálním stavu skladovaných receptů. V případě sestavování receptů je bohužel jeden zásadní problém a tím je v prototypové verzi aplikace nedostatek produktů. V produkci by to mohl být velký problém, pokud chce uživatel přidat recept a nenalezne produkt v databázi, nemá jinou možnost jak správně operaci dokončit. Proto je potřeba pozměnit implementaci přidávání produktů do receptu, jen na základě jejich názvu. Tedy indexace při vyhledávání by probíhala na základě názvů jednotlivých produktů.

Pro tuto aplikaci je naplánováno rozšíření v podobě dalších funkcionalit. Například uživatel by mohl dostávat notifikace, na základě upozornění aplikace o brzké expiraci produktů, či hodnocení jednotlivých receptů. Dalším nápadem na rozšíření aplikace, je možnost prémiových funkcí pro uživatele obsahující integraci některé z větších databází receptů, např. spolupráce s recepty.cz, nebo apetitonline.cz.

Dalším přínosným krokem aplikace by mělo být zlepšení kódu s využitím větší míry protokolově orientovaného programování a návrhového vzoru kompozice.

Zhodnocení této aplikace po implementační stránce je v celku pozitivní. Kód aplikace je čitelný, rozdělený do jednotlivých vrstev a je velice dobře testovatelný. Velkým přínosem pro implementaci bylo využití „Dependency injection“. Velká část funkcionalit aplikace je znovupoužitelných a modifikovatelných.

6 Závěr

Mobilní aplikace jsou v dnešní době velikým trendem. A to díky dostupnosti chytrých telefonů na trhu a jejich každodennímu využití. Společně s vývojem chytrých zařízení přibývají i mobilní aplikace a jejich možnosti. Není tomu dávno, kdy chytré telefony měly pouze přístup na internetový prohlížeč, možnost volání a posílání zpráv. Dnes chytré telefony mohou být použity pro účely chytré domácnosti, virtuální reality nebo zařízení internetu věcí. Například pomocí telefonu můžete regulovat teplotu domácnosti, sledovat bezpečnostní zařízení, ovládat světla, ale také můžete získávat informace o svém autě, například můžete pomocí aplikace a internetu věcí sledovat stav paliva, nebo zjistit, kdy je potřeba další servis. Mobilní zařízení jsou jistě na velikém vzestupu a společně s nimi i aplikace.

V této diplomové práci byla nastíněna problematika vývoje mobilních aplikací pro platformu iOS pomocí reaktivního programování. Na základě zjištěných poznatků z komparace jednotlivých architektur se potenciální vývojáři mohou rozhodnout, která architektura by pro jejich aplikaci byla vhodná využít. Při komparaci byla brána v potaz kritéria, jako jsou snadné použití, testovatelnost, rozdělení odpovědností a udržovatelnost aplikace. K vývoji aplikace bylo použito vývojové prostředí Xcode od společnosti Apple. Použitým programovacím jazykem je Swift ve verzi 4.1. Byl vypracován logický a grafický návrh aplikace, ve kterých byly navrženy použité komponenty a jejich rozvržení společně s funkčností. V grafickém zpracování byly využity nativní komponenty iOS aplikací tak, aby s nimi již uživatel měl zkušenost a používání aplikace mu nečinilo problémy v pochopení funkcionalit. Aplikace slouží k organizaci potravin v domácnosti pomocí jejich expirační doby. Upozorní tak uživatele na potraviny s brzkým datem expirace. Snížení vyhazování nespotřebovaných potravin by měla aplikace docílit díky navržení receptů na základě skladovaných potravin. Aplikace je určena pro ušetření peněžních prostředků tím, že nakoupené potraviny zbytečně nevyhazujeme a nekupujeme místo nich nové.

7 Seznam použitých zdrojů

1) iOS 9. *Apple (CZ).* [online]. [cit. 2018-03-14]. Dostupné z: *https://www.apple.com/cz/ios/*

2) iOS technologie. *iOS Developer Library.* [online]. 17.9.2016 [cit. 2018-03-14]. Dostupné z:*https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898-CH1-SW1*

3) Apple Inc.. *The Swift Programming language* [online]. 9.6.2015. [cit. 2018-03-14]. Dostupné z: *https://itunes.apple.com/us/book/swift-programming-language/id1002622538?mt=11*

4) David Grebeň. Kompletní historie iOS: od prvního iPhonu až po iOS 9. *Letem světem applem.* [online]. 6.3.2016 [cit. 2018-03-14]. Dostupné z: *https://www.letemsvetemapplem.eu/2016/03/06/kompletni-historie-ios/*

5) Cocoa (Touch). *iOS Developer Library.* [online]. 21.10.2015 [cit. 2018-03-14]. Dostupné z: *https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia- CocoaCore/Cocoa.html*

6) Vrstva Core Services. *iOS Developer Library.* [online]. 17.9.2014 [cit. 2018-03-14]. Dostupné z: *https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOS TechOverview/CoreServicesLayer/CoreServicesLayer.html*

7) Core OS vrstva. iOS Developer Library. [online]. 17.9.2014 [cit. 2016-03-14]. Dostupné z: https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOS/TechOverview/CoreOSLayer/CoreOSLayer.html

8) *IOS Architecture Patterns: Demystifying MVC, MVP, MVVM and VIPER* [online]. 2015 [cit. 2018-3-29]. Dostupné z: *https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52*

9) LLDB a Xcode. iOS Developer Library. [online]. 18.9.2013 [cit. 2016-03-14]. Dostupné z:

https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/Introduction.html#//apple_ref/doc/uid/TP40012917

10) MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. London, England: Prentice Hall, 2018. ISBN 0134494164.

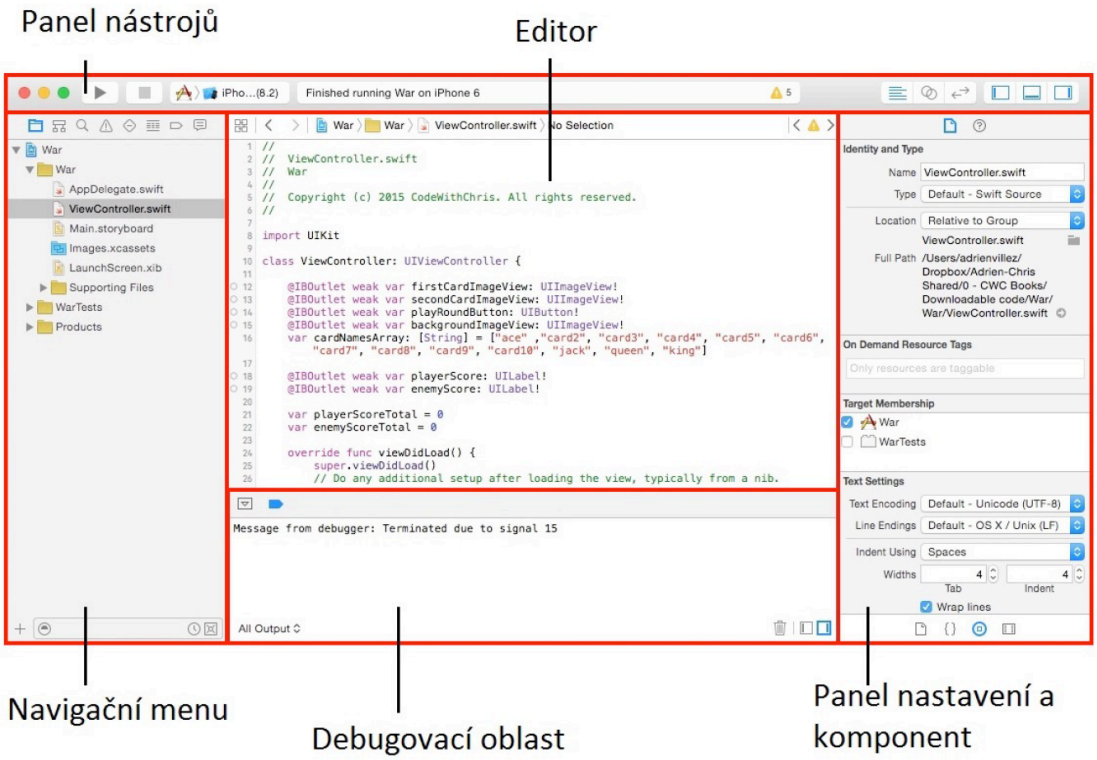
11) *The Reactive Manifesto* [online]. 2014 [cit. 2018-3-29]. Dostupné z: https://www.reactivemanifesto.org/

12) *The introduction to Reactive Programming you've been missing* [online]. 2014 [cit. 2018-3-29]. Dostupné z: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

13) PILLET, Florent, Junior BONTOGNALI, Marin TODOROV a Scott GARDENER. RxSwift: Reactive Programming with Swift, Second Edition [online]. raywenderlich.com, c2017 [cit. 2018-03-19]. ISBN 9781942878469.

8 Přílohy

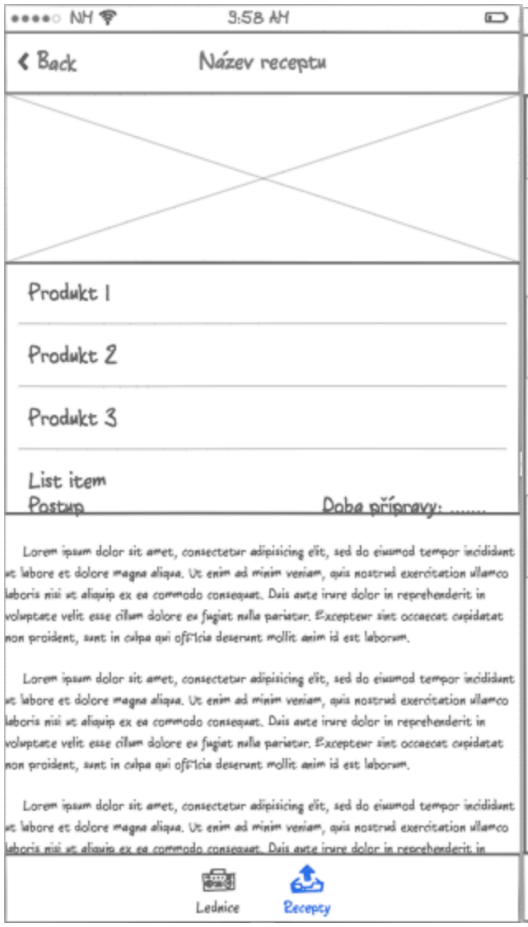
Příloha č. 1



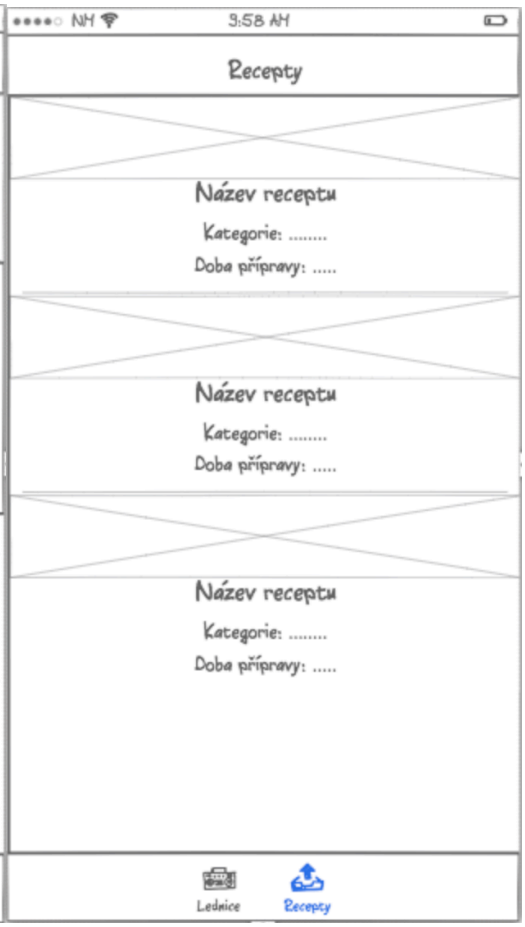
Příloha č. 2

```
class Observable1<T> {

  typealias Observer = (T) -> Void
  private(set) var observer: Observer?

  func observe(_ observer: Observer?) {
        self.observer = observer
  }

  var value: T {
        didSet {
          observer?(value)
        }
  }

  init(_ v: T) {
        value = v
  }
}
```

Příloha č. 3

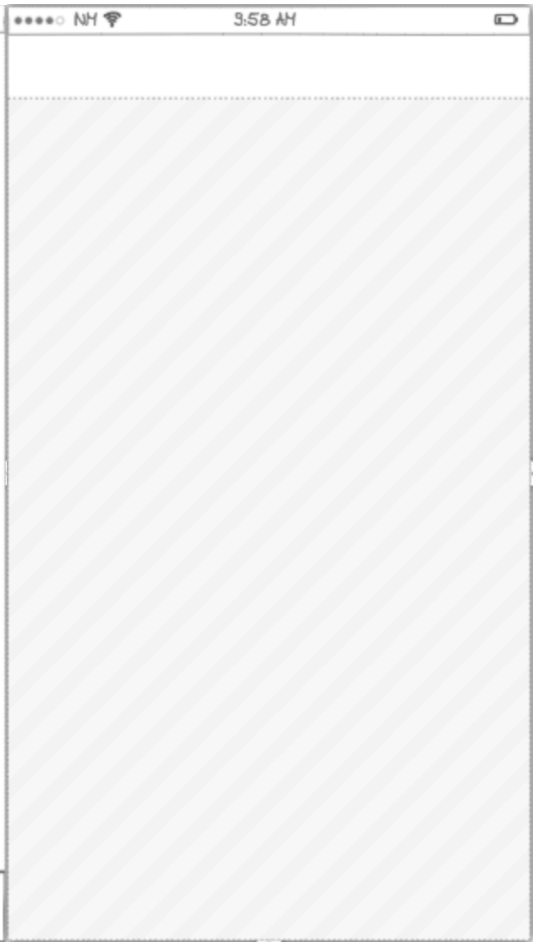


Příloha č. 4



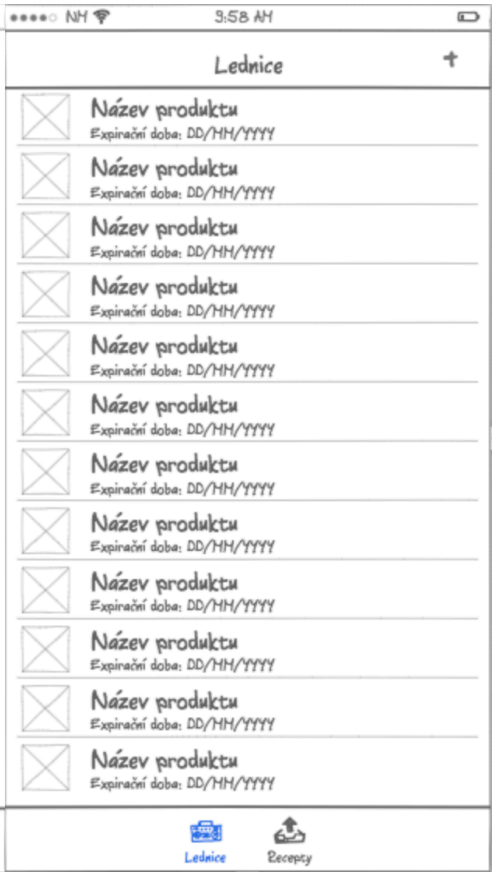
Příloha č. 5

Příloha č. 6



Příloha č. 7

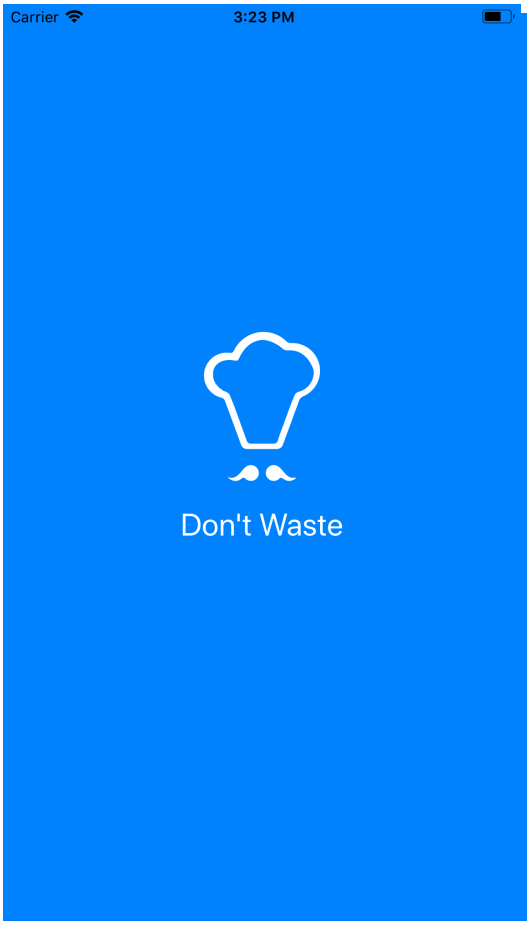
Příloha č. 8



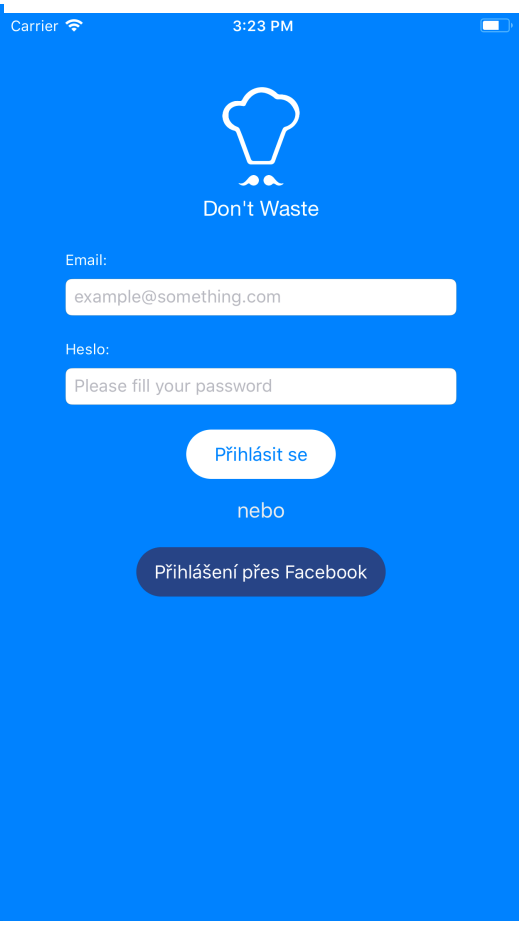
Příloha č. 9


NH
3:58 AM
Text box
Text box
Přihlásit se přes email
nebo
Přihlásit se přes Facebook

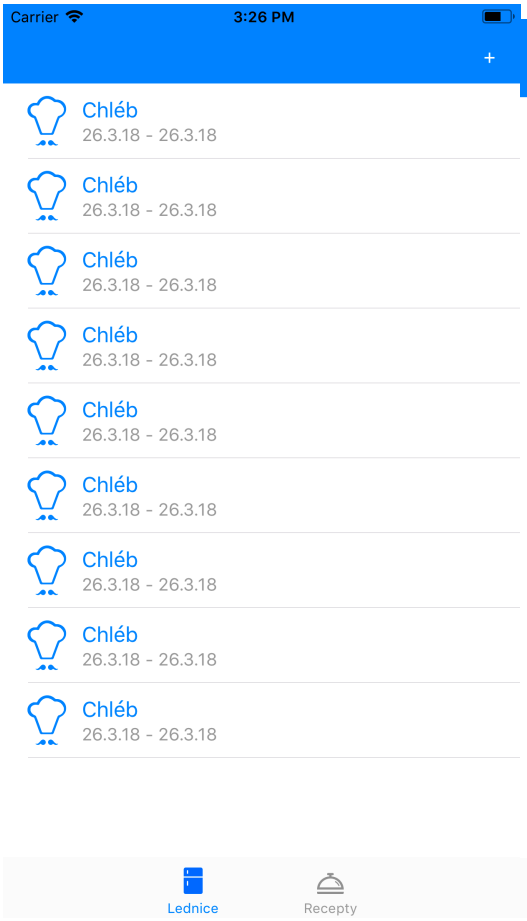
Příloha č. 10



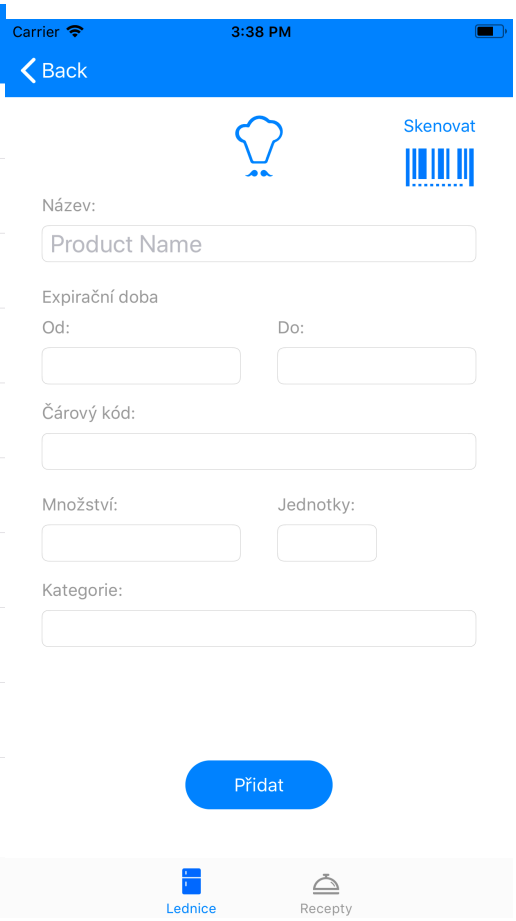
Příloha č. 11



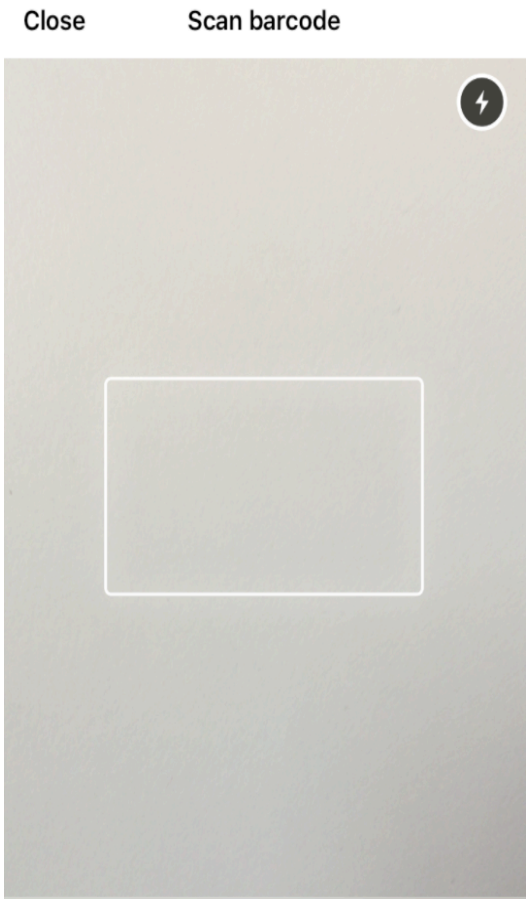
Příloha č. 12



Příloha č. 13



Příloha č. 14

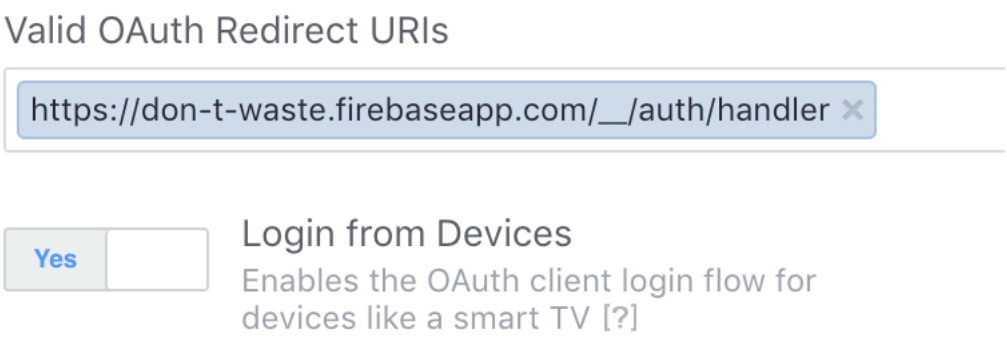


Příloha č. 15

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
  <key>CFBundleURLSchemes</key>
  <array>
    <string>fb190592988205591</string>
  </array>
  </dict>
</array>
<key>FacebookAppID</key>
<string>190592988205591</string>
<key>FacebookDisplayName</key>
<string>Don't-Waste</string>

<key>LSApplicationQueriesSchemes</key>
<array>
```

```
  <string>fbapi</string>
  <string>fb-messenger-share-api</string>
  <string>fbauth2</string>
  <string>fbshareextension</string>
</array>
```

Příloha č. 16



Příloha č. 17

```
protocol ViewModelType {

    associatedtype Input
    associatedtype Output

    func transform(input: Input) -> Output
}

public class RxViewModel: ViewModelType {

    struct Input {}
    struct Output {}

    func transform(input: Input) -> Output {

        return Output()
    }
}
```

```
public protocol StoryboardInit {

    associatedtype ViewModel

    // MARK: - Variables

    var viewModel: ViewModel? { get set }
}

public extension StoryboardInit where Self: UIViewController {

    static func storyboardInit(_ storyboardName: String, viewModel:
ViewModel?) -> Self {
        var viewController = UIStoryboard(
            name: storyboardName,
            bundle: Bundle(for: self)
            ).instantiateViewController(
                withIdentifier: String(describing: self)
            ) as! Self

        viewController.viewModel = viewModel

        return viewController
    }
}
```

Příloha č. 19

```
private let database = Database.database().reference()
    lazy var products: Observable<[Product]> = {
        return getProducts()
    }()

    func getProducts() -> Observable<[Product]> {
        return observeValueArray(path: "\(Auth.auth().currentUser?.uid ??
"")").debug()
    }

    func createProduct(product: Product) -> Observable<Void> {
        let values = Mapper<Product>().toJSON(product)
        return database
            .child(Auth.auth().currentUser?.uid ?? "")
            .childByAutoId()
            .rx.updateChildValues(values: values)
            .mapToVoid()
    }

    func updateProduct(product: Product) -> Observable<Void> {
        guard let id = product.id else {
            return Observable.error(RxError.unknown)
        }
        let values = Mapper<Product>().toJSON(product)
        return database
            .child(Auth.auth().currentUser?.uid ?? "")
            .child(id)
            .rx.updateChildValues(values: values)
            .mapToVoid()
    }

    func getRecipes() -> Observable<[Recipe]> {
        return observeValueArray(path: "recipes").debug()
    }

    func createRecipe(recipe: Recipe) -> Observable<Void> {
        let values = Mapper<Recipe>().toJSON(recipe)
        return database
            .child("recipes")
            .childByAutoId()
            .rx.updateChildValues(values: values)
            .mapToVoid()
    }

    func updateRecipe(recipe: Recipe) -> Observable<Void> {
        guard let id = recipe.id else {
            return Observable.error(RxError.unknown)
        }
        let values = Mapper<Recipe>().toJSON(recipe)
        return database
            .child("recipes")
            .child(id)
            .rx.updateChildValues(values: values)
            .mapToVoid()
  }
```

Příloha č. 20

```
public enum FacebookLoginStatus{

    case success
    case cancelled
    case failed
}

class FacebookLoginService {

    var facebookLoginManager: LoginManager?
    func facebookLogin() -> Observable<(FacebookLoginStatus, AccessToken?)>
{
        facebookLoginManager = LoginManager()
        return Observable.create { [weak self] observer in
            self?.facebookLoginManager?.logIn(readPermissions:
[.publicProfile]) { loginResult in
                switch loginResult {
                case .failed(let error):
                    print(error)
                    observer.on(.error(error))
                case .cancelled:
                    print("User cancelled login.")
                    observer.on(.next((.cancelled, nil)))
                case .success(_, _, let accessToken):
                    print("Logged in!")
                    observer.on(.next((.success, accessToken)))
                }
            }
            return Disposables.create()
        }
    }
}
```

Příloha č. 21

```
class FirebaseAuthService {

    func signIn(email: String, password: String) -> Observable<User?> {

        return Auth.auth().rx_signinWithEmail(email: email, password:
password)
    }

    func signInWithFacebook() -> Observable<User?>{

        guard let accessToken = AccessToken.current?.authenticationToken
else {

            return Observable.create { observer in

                observer.onError(FirebaseAuthError.failed)

                return Disposables.create()
            }
        }

        let credential = FacebookAuthProvider.credential(withAccessToken:
accessToken)

        return Auth.auth().rx_signInWithCredentials(credentials:
credential)
    }

    func createUser(email: String, password: String) -> Observable<User?> {

        return Auth.auth().rx_createUserWithEmail(email: email, password:
password)
    }

    var isUserAuthenticated: Bool {

        return true
    }
}
```

Příloha č. 22

```
@testable import DontWaste
import Domain
import RxSwift

class MockedNavigator: Navigator {

    var toPosts_Called = false

    func toPosts() {
        toPosts_Called = true
    }

    var toCreatePost_Called = false

    func toCreatePost() {
        toCreatePost_Called = true
    }

    var toPost_post_Called = false
    var toPost_post_ReceivedArguments: Post?

    func toPost(_ post: Post) {
        toPost_post_Called = true
        toPost_post_ReceivedArguments = post
    }
}
```