Architektury mobilních aplikací pro platformu Apple

Bc. Pavel Odstrčilík

Diplomová práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Pavel Odstrčilík**
Osobní číslo:	**A21163**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Architektury mobilních aplikací pro platformu Apple**
Téma práce anglicky:	**Mobile Application Architectures for the Apple Platform**

Zásady pro vypracování

1. Proveďte literární rešerši běžně používané architektury a návrhových vzorů využívaných při vývoji iOS a iPadOS aplikací.
2. Věnujte se také možnostem testování kódu v iOS.
3. Stanovte funkcionální a nefunkcionální požadavky na demonstrační aplikaci, která bude implementována v rámci praktické části.
4. Implementujte aplikaci dle požadavků ve dvou zvolených architekturách.
5. Věnujte se porovnání obou použitých architektur, procesu implementace a dalších odlišností a prezentujte výsledky zjištění.

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

Seznam doporučené literatury:

1. NEUBURG, Matt. IOS 15 programming fundamentals with Swift : Swift, Xcode, and Cocoa basics. Sebastopol, CA: O'Reilly, 2021. ISBN 978-1-098-11850-1.
2. LACKO, Ľuboslav. Vývoj aplikací pro iOS. Brno: Computer Press, 2018. ISBN 978-80-251-4942-3.
3. Documentation. Swift [online]. Apple, 2022 [cit. 2022-11-27]. Dostupné z: https://www.swift.org/documentation/
4. Apple Developer Documentation. Apple Developer [online]. Apple, 2022 [cit. 2022-11-27]. Dostupné z: https://developer.apple.com/documentation/
5. SwiftUI. Apple Developer [online]. Apple, 2022 [cit. 2022-11-27]. Dostupné z: https://developer.apple.com/documentation/swiftui/
6. VÁVRŮ, Jiří. IPhone: vývoj aplikací. Praha: Grada, 2012. Průvodce (Grada). ISBN 978-80-247-4457-5.
7. MARK, Dave a Jeff LAMARCHE. IPhone SDK: průvodce vývojem aplikací pro iPhone a iPod touch. Brno: Computer Press, 2010. ISBN 978-80-251-2820-6.

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **2. prosince 2022**
Termín odevzdání diplomové práce: **26. května 2023**

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

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

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

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 26. května 2023

Bc. Pavel Odstrčilík, v.r.
podpis studenta

ABSTRAKT

Tato diplomová práce se zabývá architektonickými a návrhovými vzory, které lze využít k vývoji mobilních aplikací pro Apple platformu. Vývojáři mobilních aplikací na platformě iOS mají na výběr z několika možných architektonických vzorů využitelných při tvorbě projektu. Práce se zabývá právě specifiky jednotlivých řešení a jejím cílem je doporučení konkrétní architektury. Práce se primárně zaměřuje na tvorbu mobilních aplikací pro iOS s využitím jazyka Swift a knihovny SwiftUI. Cílem teoretické části je představit vybrané architektury a návrhové vzory pro vývojáře mobilních aplikací. Dále jsou popsány možnosti testování kódu v operačním systému iOS. V praktické části jsou zvoleny dvě architektury, pomocí kterých jsou vytvořeny demo aplikace sloužící k porovnání jejich vhodnosti a vyvození závěrů a doporučení.

Klíčová slova: iOS, Swift, SwiftUI, mobilní aplikace, architektury

ABSTRACT

This thesis explores architectural and design patterns that can be used to develop mobile applications for the Apple platform. Developers of mobile applications on the iOS platform have a choice of several possible architectural patterns to use in the development of the project. This thesis deals specifically with the specifics of each design and aims to recommend a particular architecture. The work primarily focuses on the development of iOS mobile applications using the Swift language and the SwiftUI library. The aim of the theoretical part is to present selected architectures and design patterns for mobile application developers. Furthermore, the possibilities of code testing in the iOS operating system are described. In the practical part, two architectures are selected and used to create demo applications used to compare their suitability and draw conclusions and recommendations.

Keywords: iOS, Swift, SwiftUI, mobile apps, architectures

Tímto bych rád poděkoval vedoucímu mé diplomové práce Ing. Radku Valovi, Ph.D. za konzultace a odborné vedení.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Tato diplomová práce se zaměřuje na analýzu architektonických a návrhových vzorů, které lze využít k vývoji mobilních aplikací pro Apple platformu. Vývojáři mobilních aplikací na platformě Apple mají na výběr z mnoha možných architektonických vzorů. Každý vzor má svoje specifika, která ovlivňují konkrétní implementaci projektu. Práce se zaměřuje na tvorbu mobilních aplikací pro iOS s využitím jazyka Swift a knihovny SwiftUI. Cílem práce je představení vybraných architektur a návrhových vzorů. Dále budou zvoleny dvě architektury, pomocí kterých jsou vytvořeny demo aplikace sloužící k porovnání.

V teoretické částí bude obecně popsána knihovna SwiftUI a její důležité prvky. V další kapitole budou popsány jednotlivé architektury, a to MVC, MVVM, MVVM-C, MVP, Clean Architecture, Viper a Redux. V dalším bodě práce budou popsány návrhové vzory, které se využívají v Apple platformě, jako je například vzor Target-Action. Diplomová práce v závěru teoretické části popisuje možnosti testování kódu v operačním systému iOS.

V praktické části bude popsána tvorba demo aplikací. V první kapitole této části budou vyobrazeny funkcionální a nefunkcionální požadavky na demo aplikaci. Následně bude obecně popsána demo aplikace a její význam. Poté bude popsána implementace aplikací ve dvou vybraných architekturách. V závěru práce budou tyto implementace porovnány a budou vyvozeny závěry a doporučení.

I. TEORETICKÁ ČÁST

1 SWIFTUI

SwiftUI je multiplatformní knihovna od společnosti Apple pro vytváření uživatelského rozhraní. Knihovnu lze využít v aplikacích pro operační systémy iOS, iPadOS, tvOS, macOS a watchOS, což například s knihovnou UIKit nebylo možné a je tím jednodušší psát aplikace pro celou Apple platformu. Knihovna byla představena v roce 2019 na konferenci Worldwide Developers Conference. Základním přístupem implementace pomocí této knihovny je deklarativní přístup. SwiftUI využívá jazyk Swift a obsahuje všechny důležité prvky uživatelského rozhraní známé z UIKit, jako jsou seznamy, tlačítka, tlačítka pro zvolení času a podobně. Samozřejmostí jsou také animace, podpora různých gest a zabudované funkcionality pro zpřístupnění, tzn. pro uživatele s omezenou schopností vidět a slyšet. Knihovna poskytuje nástroje pro řízení toku dat v aplikaci od modelu k uživatelskému rozhraní. Hlavním cílem SwiftUI je poskytnou nástroj pro vytváření uživatelského rozhraní jednoduše a rychle proto, aby se vývojář mohl soustředit na funkcionalitu na pozadí. Na pozadí tato knihovna stále využívá komponenty z knihoven UIKit a AppKit.[1][2]

1.1 Syntax

Jak již bylo zmíněno v úvodu, SwiftUI využívá deklarativní syntax. Výsledkem je jednoduchý a intuitivní popis toho, jak má uživatelské rozhraní vypadat. Například lze definovat, že chceme zobrazit seznam položek, který se skládá z textových polí a následně určit font a barvu pro tyto prvky. Kód je díky tomu čistější a jednodušší na čtení. Výsledkem je úspora času při údržbě kódu.

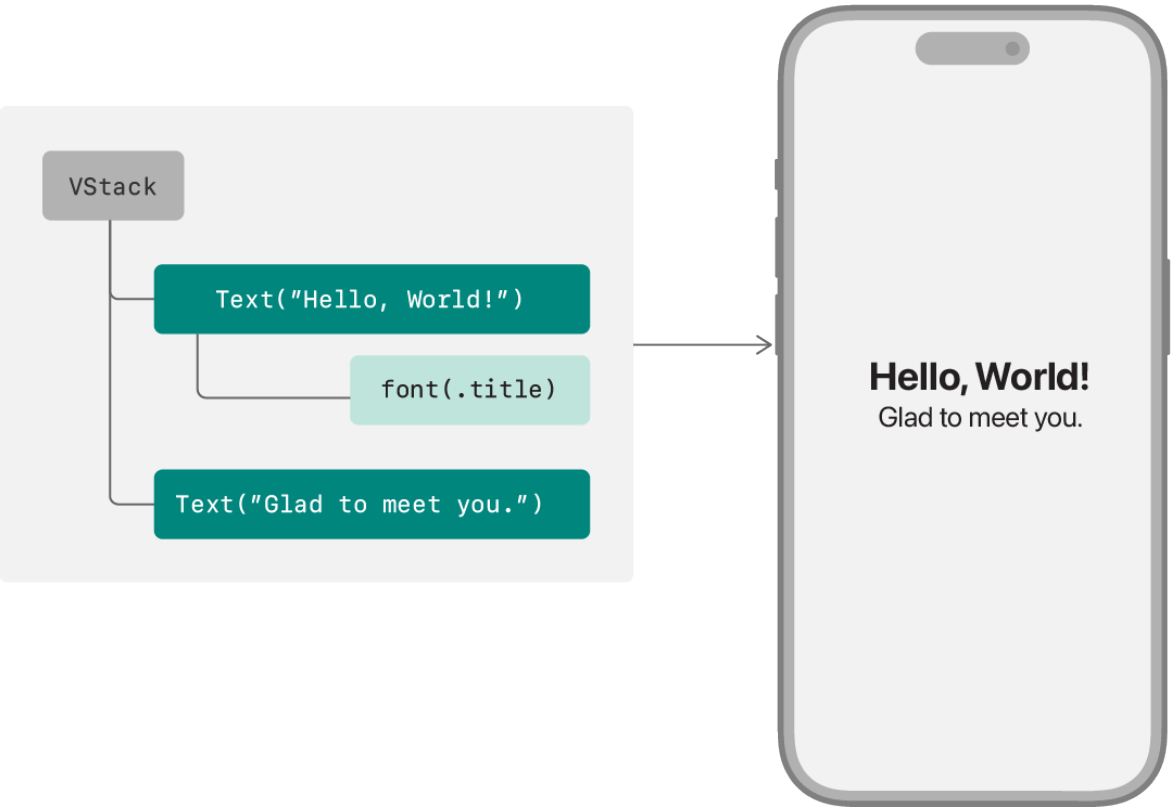
Deklarativní styl je také aplikován pro komplexnější koncepty jako jsou animace. Animace lze snadno přidat do většiny ovládacích prvků a lze využít předdefinované animace, které lze využít pomocí napsání pár řádků kódu. Systém během běhu programu řeší všechny potřebné kroky pro vytvoření plynulého pohybu animace, a dokonce se i vypořádává s uživatelskou interakcí a změnami stavu během běžící animace. [3]

1.2 Uživatelské rozhraní

SwiftUI nabízí deklarativní přístup pro vytváření uživatelského rozhraní. Tím je lehce vytvořen popis uživatelského rozhraní, který reflektuje požadovaný vzhled aplikace. SwiftUI se stará o vykreslování a aktualizaci uživatelského rozhraní v reakci na změnu dat či uživatelský vstup.

Knihovna poskytuje nástroje pro definování a konfiguraci jednotlivých uživatelských prvků. Vlastní prvky lze seskládat z již existujících komponent, které knihovna nabízí, nebo z vlastních komponent. Poté jsou tyto komponenty nakonfigurovány pomocí modifikátorů a propojeny s datovým modelem a vloženy do hierarchie aplikace. [4]

Výhodou SwiftUI je možnost využití živého náhledu kódu uživatelského rozhraní ve vývojovém prostředí Xcode.



Obrázek 1 Tvorba uživatelského rozhraní pomocí SwiftUI [4]

1.2.1 View Protokol

Základním prvkem aplikace je protokol View. Představuje část uživatelského rozhraní aplikace a poskytuje modifikátory sloužící pro konfiguraci View. Protokol je aplikován na datový typ struktury. Po aplikaci a splnění tohoto protokolu lze definované View zobrazit v aplikaci. [4]

1.2.1.1 Implementace protokolu

Pro vytvoření vlastního uživatelského rozhraní je tedy nutno aplikovat protokol na strukturu a následně implementovat protokolem požadovanou počítanou proměnnou pojmenovanou jako body typu some View, což je takzvaný neprůhledný typ. Ten říká, že typ těla odpovídá

protokolu View. Přesný typ závisí na obsahu těla této proměnné, který se mění v průběhu vývoje. Odvození přesného typu provede Swift automaticky.

SwiftUI čte hodnotu body kdykoliv, kdy potřebuje aktualizovat View. To nastává několikrát během životního cyklu View, jako je například reakce na uživatelskou interakci nebo systémové události. Hodnota, která je z této počítané proměnné vrácena je vykreslena na obrazovku uživateli.

Následně lze do těla proměnné body definovat jednotlivé prvky uživatelského rozhraní. [4]

1.3 App protokol

Základem SwiftUI aplikace je protokol App. Struktura, která tento protokol implementuje musí obsahovat počítanou proměnnou pojmenovanou body. Tato proměnná definuje obsah aplikace. To znamená, že určuje to, které View se po spuštění aplikace zobrazí. Před deklarací této struktury je nutno uvést atribut @main. Tento atribut slouží pro indikování hlavního vstupního bodu do aplikace, a tato struktura je tedy volána systémem po zapnutí aplikace. Z toho vyplývá, že lze mít v aplikaci pouze jednu strukturu označenou jako @main. [5]

1.4 Modifikátory

Modifikátory jsou důležitou části SwiftUI. Pomocí modifikátorů lze upravovat vlastnosti View. Pomocí modifikátorů lze upravit například barvu, velikost fontu či styl prezentace. Modifikátory lze aplikovat na libovolný typ, který implementuje protokol View. Při aplikaci modifikátoru je modifikátorem vráceno nové View, které zaobaluje původní View a toto nové View nahrazuje originální View v hierarchii. Běžně se modifikátory řetězí a každý z nich obaluje výsledek předchozího. Modifikátory jsou aplikovány jeden po druhým. Ve SwiftUI lze aplikovat modifikátor na rodičovský prvek a poté potomci tohoto prvku přebírají daný modifikátor. [6]

1.5 Property wrappers

Knihovna SwiftUI poskytuje různé druhy property wrappers, což jsou v překladu obaly na proměnné. Tyto obaly proměnných jsou velmi důležité pro práci s tokem dat a podporují znovu používání kódu. Veškeré View prvky ve SwiftUI jsou struktury, což znamená, že je nelze změnit. Pokud v takové struktuře ale použijeme obalenou proměnnou, které jsou obstarávané SwiftUI, je tento problém vyřešen. Jakmile je hodnota v proměnné změněna,

tak je následně vzhled View znehodnocen a následně je vzhled znovu přepočítán a vykreslen. [7]

V následujících podkapitolách jsou popsány vybrané obaly proměnných, které se ve SwiftUI používají.

1.5.1 State

SwiftUI využívá obal na proměnnou @State k povolení úpravy hodnoty uvnitř struktury. Úprava hodnoty uvnitř struktury by jinak nebyla možná z důvodu toho, že jsou hodnotovým typem. State označuje proměnnou jako jediný zdroj pravdy pro daný typ hodnoty, který se nachází v hierarchii zobrazení. Proměnná s tímto obalem musí mít definovanou inicializační hodnotu.

Pokud označíme proměnnou pomocí tohoto obalu, tak je místo v paměti proměnné přesunuto ze struktury do sdílené paměti, kterou spravuje SwiftUI. Výsledkem je to, že SwiftUI může smazat a znovu vytvořit původní strukturu kdykoli je to potřeba bez ztráty stavu, který byl uložen. Tento obal proměnné je využíván pro hodnotové datové typy jako je například String a Int.

Obecně by taková proměnná neměla být sdílena mezi více prvky uživatelského rozhraní, na tyto účely existují další obaly, které jsou popsány v dalších podkapitolách. Z tohoto důvodu je doporučeno označovat tyto proměnné jako private. [8] [9]

1.5.2 StateObject

Tento obal označuje proměnnou jako takzvaný jediný zdroj pravdy pro referenční typ dat, které jsou uloženy ve View hierarchii. Řeší tedy problém, kdy potřebujeme mít referenční typ dat ve View struktuře. Jedná se o podobný případ obalu proměnné jako je State s rozdílem referenčního typu dat. Obal je aplikován přidáním @StateObject atributu k deklaraci proměnné. Proměnná musí aplikovat ObservableObject protokol a zároveň by měla být označená jako private. [10] [11]

1.5.3 Binding

Binding je typ obalu proměnné, který může číst a zapisovat do hodnoty označené jako takzvaný jediný zdroj pravdy, což je proměnná označená jako @State. Obal propojuje proměnnou, která ukládá data a View, které zobrazuje a případně upravuje data. Binding se propojuje se zdrojem dat, které jsou uloženy někde jinde a nemusí být tedy data uložena

přímo v konkrétním místě. Hodnota je propisována v obou směrech. Binding říká, že je hodnota uložena někde jinde a mezi proměnnou a původním zdrojem hodnoty se sdílí. [12] [13]

1.5.4 Published

Obal Published patří mezi často vyžívané ve SwiftUI. Důvodem je to, že umožňuje vytvářet sledovatelné objekty, které automaticky oznamují, že došlo k nějaké změně. SwiftUI toto automaticky kontroluje a v případě změny je aktualizované uživatelské rozhraní, které je na tuto proměnnou napojeno. [14]

1.5.5 ObservedObject

Tento obal se nám umožňuje ve View sledovat stav externího objektu a reagovat na změny objektu. Chování je podobné jako v případě StateObject ale obal nesmí být použít pro vytváření nového objektu. ObservedObject se používá pouze pro objekty, které byly vytvořeny někde jinde, jinak by tento objekt mohl být knihovnou SwiftUI smazán. Objekt, na který se tento obal aplikuje musí splňovat protokol ObservableObject. Tento protokol lze aplikovat pouze na třídu, nelze jej tedy aplikovat na strukturu. ObservedObject je používán pro data, které jsou uloženy jinde a mohou tedy být sdíleny mezi více View. [15]

1.5.6 Environment

Proměnná označená jako Environment umožňuje číst data ze systému. Příkladem může být získání aktuálního barevného schématu, tedy zda má zařízení zapnutý světlý nebo tmavý režim. Hodnotu, kterou chceme získat definujeme pomocí key path hodnoty v definici proměnné. [16]

1.5.7 EnvironmentObject

EnvironmentObject nám umožňuje vytvářet View, které je závislé na sdílených datech. Taková data mohou být sdílena celou SwiftUI aplikací. Příkladem může být sdílení uživatele mezi celou aplikací. Při používání tohoto obalu lze vytvořit View A, které vytvoří objekt a uloží jej jako EnvironmentObject. Následně jakékoliv View ve View A může získat přístup k uloženému objektu. Výsledkem je, že není nutno explicitně předávat objekt přes jednotlivá View. [17]

1.5.8 AppStorage

SwiftUI poskytuje obal pro čtení a zápis hodnoty do UserDefaults. V případě změny hodnoty v UserDefaults je View zneplatněno a aktualizováno. Tím je velmi efektivně sledována hodnota nacházející se v UserDefaults. Zde stojí za zmínku, že UserDefaults není určeno pro ukládání citlivých dat, jelikož se nejedná o bezpečné uložiště a uložená data lze získat. [18]

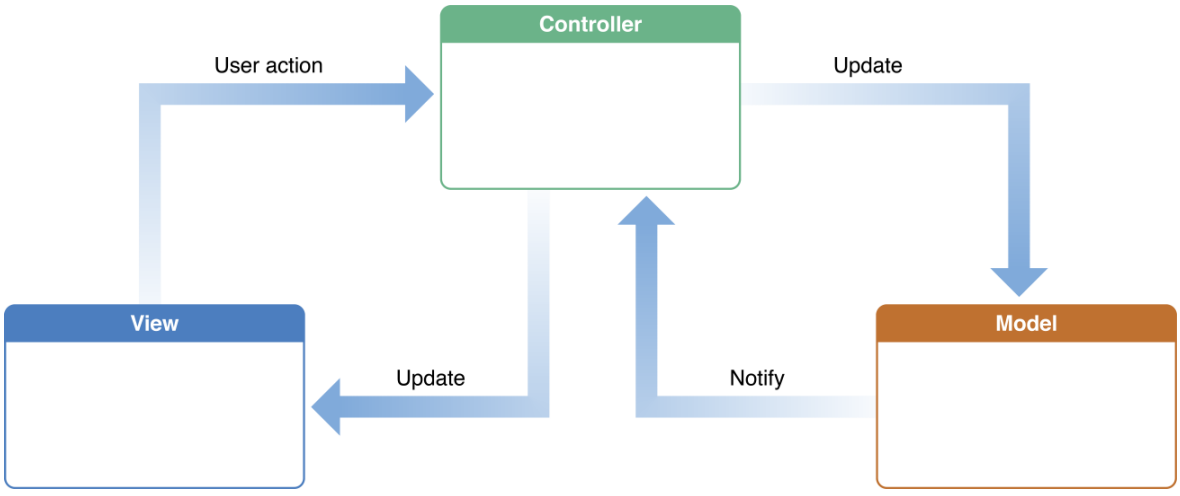
2 ÚVOD DO ARCHITEKTUR

Architektury jsou velmi důležitou částí vývoje všech aplikací. Zvolená architektura ovlivňuje, jak lze aplikace dlouhodobě udržovat. V praxi může nastat případ, kdy budeme chtít aktualizovat část kódu, například networking, a tato změna může na základě architektury být jednodušší či pracnější. Dalším důležitým bodem je testovatelnost. Určité architektury jsou vytvářeny pro lepší testovatelnost. Zvolená architektura ovlivňuje i výkon aplikace. V neposlední řadě architektura určuje rozšiřitelnost aplikace.

Existuje velké množství architektur, které lze využít pro vývoj mobilních aplikací pro platformu Apple. Následující kapitola popisuje vybrané architektury, které lze využít pro vývoj aplikací se zaměřením na Apple platformu.

2.1 Model-View-Controller

Model-View-Controller (zkráceně MVC) je již dlouze využívaná architektura, která rozděluje kód do částí, které jsou nazývány Model, View a Controller. Každá část má přesně definovanou funkčnost. Využití MVC má mnoho výhod. Mnoho tříd či objektů lze jednoduše znovu využívat a aplikace lze lépe přizpůsobovat měnícím se požadavkům. Tento vzor nedefinuje pouze role objektů v aplikaci, ale definuje i komunikaci mezi objekty. Každý ze tří typů objektů je oddělen od ostatních. Mnoho částí frameworku Cocoa je založeno na architektuře MVC. [19]



Obrázek 2 Schéma Model-View-Controller [19]

2.1.1 Model

Model zapouzdřuje specifická data pro aplikaci a definuje logiku a výpočty, které s těmito daty počítají či je zpracovávají. Příkladem modelu může být kontakt v seznamu kontaktů. Model může mít vazbu k dalším modelům. Většina dat, která aplikace využívá, by měla být po načtení do aplikace uložena v modelech. Modely reprezentují znalost ke konkrétnímu doménovému problému. Modely lze opakovaně používat v doménových oblastech, které jsou si podobné. Model by neměl mít přímé spojení s objekty, které řeší uživatelské zobrazení. Tato část by neměla řešit problémy spojené s prezentací dat pro uživatele.

Uživatelské akce z objektu ze skupiny View, které vytvářejí nebo upravují data v modelu jsou komunikovány skrz objekt ze skupiny Controller. V případě změny dat v modelu, což může být například načtení nových dat z internetu, model notifikuje Controller, a ten aktualizuje View objekt. [19] [20]

2.1.2 View

Objekt ze skupiny View je objekt, který uživatel může v aplikace vidět. View má informace o tom, jak se má vykreslit a může také reagovat na uživatelskou interakci. Hlavním účelem je zobrazovat data z modelu uživateli a případně umožňovat jejich úpravu. Nemusí ale vždy zobrazovat veškerá data z modelu, pro uživatele může být zobrazena pouze konkrétní část modelu. View není zodpovědné za ukládání dat. View by mělo zajišťovat správné zobrazení dat z modelu. View zobrazuje informace uživateli.

Objekty View jsou typicky znovu používány a konfigurovány pro konkrétní použití v aplikaci. V knihovně UIKit nebo AppKit jsou obsaženy třídy, které jsou určeny pro prezentaci uživateli.

Jak již bylo zmíněno v části o modelu, objekty View komunikují s modelem přes Controller. Příkladem může být vyplnění textu do textového pole, kdy je následně tato informace z View předána do Controlleru a následně do Modelu. View komunikuje pouze s objekty typu Controller. [19] [20]

2.1.3 Controller

Poslední částí MVC je Controller, který slouží jako prostředník mezi jedním a více View a jedním a více Modely. Prostřednictvím Controlleru je View notifikováno o změnách v Modelu a naopak. Objekty z části Controller mohou nastavovat objekty, koordinovat

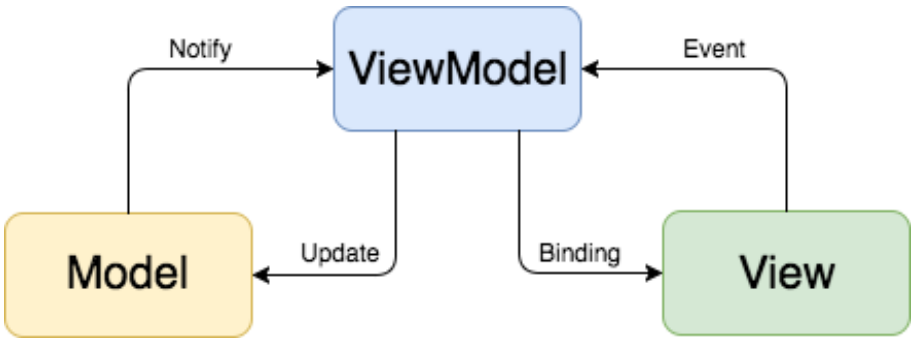
aplikaci a řídit životní cykly ostatních objektů. Tyto objekty mohou být také znovu používány.

Typicky v MVC, pokud uživatel zadá hodnotu pomocí View objektu, tak je tato hodnota v Controlleru zpracována specifickým způsobem pro aplikaci a následně je notifikován Model, jak s danou informací pracovat, jako může být přidání nového záznamu do Modelu. Controller může na základě dat z Modelu informovat View o změně stavu na základě znalosti dat, jako může být deaktivace tlačítka či změna barvy tlačítka. Při změně dat v Modelu může být Controller požádán o aktualizaci View objektů.

V některých případech mohou být role jednotlivých objektů slučovány. Příkladem může být kombinace části View a Controller do jednoho objektu. V praxi vývoje mobilních aplikací pro Apple platformu bývá toto spojení častým příkladem. [19] [20]

2.2 Model-View-ViewModel

Architektura Model-View-ViewModel (zkráceně MVVM) rozděluje kód opět do tří částí, kdy každá část má danou odpovědnost. Kód je dělen do částí Model, View a ViewModel. Tato architektura je odvozená od MVC.



Obrázek 3 Schéma MVVM [21]

2.2.1 Model

Model má stejné vlastnosti jako model popsaný v architektuře MVC. Model zapouzdřuje data, se kterými aplikace pracuje. Tento objekt je využíván objektem ViewModel a je aktualizován na základě ViewModelu. Model notifikuje ViewModel o změnách. [21]

2.2.2 View

Objekt View má opět podobné vlastnosti jako v architektuře MVC. View zapouzdřuje uživatelské rozhraní. Tato část komunikuje s Modelem skrz ViewModel. [21]

2.2.3 ViewModel

ViewModel je zodpovědný za business logiku aplikace, za spojení Modelu a View a zachovává také stav aplikace. ViewModel je nezávislý od View. V případě uživatelské akce ve View je ViewModel notifikován, následně je informace zpracována a případně předána do Modelu. ViewModel připravuje data z Modelu pro zobrazení ve View. [22]

V následujícím kódu lze vidět ukázková implementace ViewModelu v knihovně SwiftUI. Kód definuje třídu BookViewModel, která dědí z ObservableObject. Třída obsahuje proměnnou books označenou jako @Published. Tím je zajištěna automatická aktualizace ve všech místech, kde se tato proměnná používá, což může být například uživatelské rozhraní. ViewModel obsahuje funkce pro přidání a smazání knihy. Jak bylo zmíněno, ve ViewModelu se nevyskytuje žádná logika zobrazování jednotlivých View.

```
class BookViewModel: ObservableObject {
    @Published var books: [Book] = []

    public func addBook(_ book: Book) {
//          ...
    }

    public func deleteBook(id: String) {
//        ...
    }
}
```

2.3 Model-View-ViewModel-Coordinator

V případě předešlé architektury MVVM lze jako nedostatek určit absence správy navigace. Architektura MVVM-C rozšiřuje MVVM o objekt zvaný Coordinator. Části Model, View a ViewModel mají stejné vlastnosti jako u architektury MVVM.

2.3.1 Coordinator

Tento vzor zajišťuje logiku zodpovědnou za přechody v aplikaci. Coordinator je objekt, který je zodpovědný za řízení navigace mezi obrazovkami, říká tedy, které View se následně zobrazí či které View se skryje. Jeho úkolem je i řídit tok dat a definovat, jaký druh přechodu mezi jednotlivými View se provede na základě dané události. Výhodou využití tohoto objektu je to, že View se nestará o řízení navigace a tím více dodržuje princip jediné odpovědnosti. Tím je více zaručena znovu použitelnost View. [23]

```
class Coordinator {
    var navigationController = UINavigationController()

    func showDetail(of book: Book) {
        let bookVC = BookDetailViewController()
//          ...
        navigationController.pushViewController(bookVC,
                                                animated:true)
    }
}
```

V ukázkovém kódu je zobrazena ukázková implementace Coordinator v knihovně UIKit. Třída obsahuje proměnnou typu UINavigationController, která zajišťuje navigaci v UIKit. Následně je ve třídě funkce showDetail, která v aplikaci zobrazí třídu BookDetailViewController. Tato funkce by se v praxi volala například při kliknutí na buňku v UITableView. Tím by byla logika navigace přesunuta do třídy Coordinator.

2.4 Model-View-Presenter

Tato architektura se skládá z Modelu, View a Presenteru. Funkce Modelu je stejná jako u architektury MVC.

Obrázek 4 Schéma Model-View-Presenter [24]

2.4.1 View

View zajišťuje přípravu a zobrazení uživatelského rozhraní. Tato část neobsahuje žádnou další logiku.

2.4.2 Presenter

Nejvýraznější částí této architektury je Presenter. Tato část je zodpovědná za reakce na uživatelské akce, aktualizaci View a logiku aplikace. Výsledkem v praxi je redukce velikost ViewControlleru, který obvykle obsahuje velké množství kódu. Redukcí je i zajištěna lepší testovatelnost. Presenter je přímo napojen k View. [24]

V následující ukázce lze vidět implementaci Presenteru v knihovně UIKit. Je vytvořena třída Presenter, která obsahuje v ukázce logiku aplikace a zachovává referenci na View, což je

v našem případě BookView. BookViewController je závislý na třídě Presenter. Výsledkem je redukce kódu v BookViewControlleru.

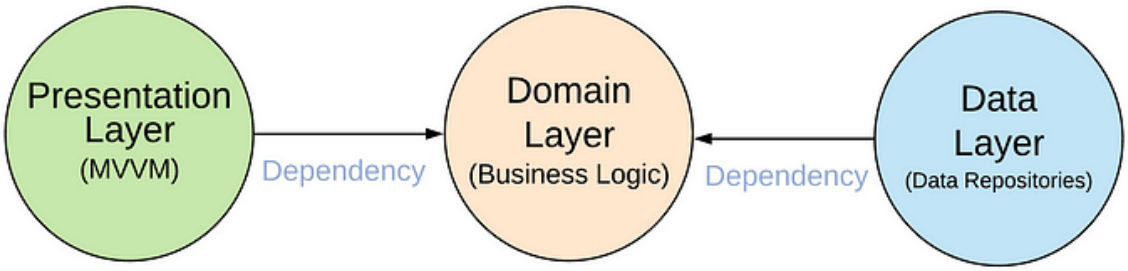
```
protocol BookView: AnyObject {
//    ...
}

class BookViewController: UIViewController, BookView {
    var presenter: Presenter?

    override func viewDidLoad() {
        super.viewDidLoad()

        presenter = Presenter(view: self)

        presenter?.fetchData()
    }
}

class Presenter {
    weak var view: BookView?

    init(view: BookView) {
        self.view = view
    }

    func fetchData() {
//        ...
    }
}

```

2.5 Clean Architecture

Clean architecture je návrhová filozofie, která rozděluje části aplikace do různorodých vrstev. Každá vrstva má svoji specifickou odpovědnost. V překladu je nazývána jako čistá architektura z důvodů, že kód je snadno čitelný, testovatelný a udržovatelný. Zároveň není kód vázán na konkrétní knihovnu či technologii.

Kód je rozdělen do tří částí, a to Presentation layer, Domain layer a Data access layer. Každá vrstva má přesně definovanou odpovědnost. Každá vrstva komunikuje s okolními vrstvami skrz přesně definované rozhraní. Tím je umožněno jednotlivé vrstvy vyvíjet, testovat a udržovat nezávisle na sobě. Díky tomu je zaručena jednoduchost při úpravě či přidávání nových funkcionalit do jedné vrstvy bez zasažení do ostatních vrstev aplikace.

Pokud je například chceme rozšířit aplikace a přidat nový zdroj dat pro aplikaci, je do Data acces layeru přidána nová třída, která danou funkcionalitu implementuje. Tato změna by neměla mít vliv na ostatní vrstvy, které by měly fungovat jako před touto úpravou. [25]



Obrázek 5 Schéma Clean Architecture [25]

2.5.1 Presentation layer

Tato vrstva je nejvíce vnější vrstvou. Je zodpovědná za zpracování uživatelského vstupu a zobrazení výstupu uživateli. Tato vrstva obsahuje SwiftUI View prvky, které určují uživatelské rozhraní aplikace. Prezentační vrstva je závislá pouze na doménové vrstvě. [25] [26]

2.5.2 Domain layer

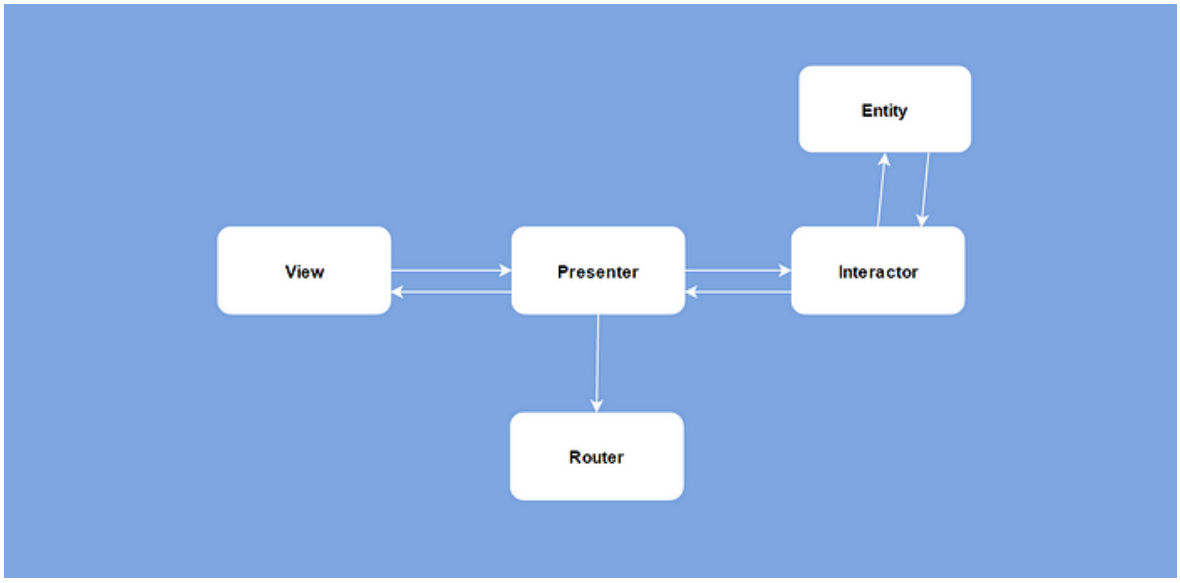
Doménová vrstva je středem architektury a nemá závislosti na jiných vrstvách. Vrstva je kompletně izolovaná. Tato část je zodpovědná za implementaci hlavní business logiky aplikace. Vrstva obsahuje třídy definující datové modely a jejich logiku. Doménová vrstva lze teoreticky znovu použít v různých projektech. Tato vrstva by neměla obsahovat nic z ostatních vrstev, jako mohou být prezentační funkcionality z UIKit nebo SwiftUI. [25] [26]

2.5.3 Data access layer

Tato vrstva je nejvíce vnitřní vrstvou. Je zodpovědná za získání a ukládání dat. Do této části architektury zapadají třídy, které načítají data ze sítě nebo z lokální databáze a následně je poskytují doménové vrstvě. Vrstva je závislá na doménové vrstvě. V této vrstvě se nachází například funkce sloužící pro mapování z JSON na doménové modely. [25] [26]

2.6 VIPER

VIPER je architektura, která rozděluje kód do pěti různých částí.



Obrázek 6 Schéma architektury VIPER [27]

2.6.1 View

Tato část obsahuje všechen kód, spojený s uživatelským rozhraním, které uživatel vidí a se kterým interaguje. View po získání informace od uživatele komunikuje s částí Presenter. [27]

2.6.2 Presenter

Presenter je hlavní část této architektury. Získává uživatelský vstup z View vrstvy a podle toho pracuje. Tato vrstva je jediná, která komunikuje se všemi ostatními komponentami této architektury. Presenter volá Interactor pro získání dat a Router pro řešení zobrazovaných View. [27]

2.6.3 Interactor

Vrstva je zodpovědná za zpracování logiky aplikace. Přistupuje tedy k datové vrstvě a provádí výpočty a operace s daty. Pokud aplikace obsahuje logiku založenou na volání internetových požadavků, tak tato část je za to odpovědná. Interactor komunikuje pouze s Presenterem. [27]

2.6.4 Router

Router je zodpovědný za navigaci mezi obrazovkami aplikace. Router zajišťuje vytvoření View a následnou prezentaci v aplikaci na základě volání z Presenteru. [27]

2.6.5 Entity

Jedná se o datový model aplikace. Obsahuje modelové třídy. Tyto třídy jsou používány vrstvou Interactor. [27]

2.7 Redux

Architektura Redux je založena na ochraně aktuálního stavu aplikace. Každá specifická akce má jasně definovaný stav, do kterého se aplikace přesune po vykonání akce. V porovnání s UIKit aplikacemi je tato architektura mnohem lépe použitelná v kombinaci se SwiftUI. V této architektuře je definováno 5 jednoduchých komponent, které jsou v následujících sekcích popsány.

2.7.1 State

Aplikace by měla mít globální stav, který obsahuje vše, co definuje View a jednotlivé funkce. Tento globální stav je většinou složen z mnoha menších stavů, nejedná se o jednu velkou strukturu. Zde lze mít uložené libovolné množství informací, jako například zda má být textový vstup povolen či nikoliv. Množství uložených stavů je na vývojáři ale v případě nedostatečného množství stavů může nastat problém, kdy View nelze plně obnovit do daného stavu. Pro definici stavu je často využívána struktura. [28]

2.7.2 Action

Akce definuje jednoduchý příkaz, který informuje Reducer o tom, co se stalo s aplikací. Příkaz může také obsahovat data spojená s informací, jako například id. Akce jsou předávány do Store skrz View nebo Middleware. Akce mohou být definované jako výčtový typ. [28]

2.7.3 Reducer

Reducer je zodpovědný za vrácení nového stavu založeném na starém stavu a akci. Reducer existuje pro každou akci. Zjednodušeně se jedná o funkci, která příchozí akci přemění na stav. Tato komponenta by neměla využívat nic mimo svůj rozsah a vždy by měla vrátit stejný výsledek pro stejný stav a akci. V této části Redux architektury lze jasně vidět, jak se stavy mění na základě akcí. Reducer lze jednoduše přečíst a testovat. [28]

2.7.4 Middleware

Do této části patří například logika jako je logování, volání API nebo přístup k uložišti. Jakmile je akce odeslána, měla by projít přes všechny Middleware objekty spolu se stavem. Na základě toho může ale nemusí Middleware odeslat novou akci. Příkladem na objasnění chování Middleware může být volání API požadavku. Middleware obdrží akci na načtení dat, zavolá API požadavek a následně vyšle další akci s výsledkem. Díky této komponentě je pěkně rozdělen kód do daných logických celků. V aplikaci lze mít několik různých Middleware, kdy se každý stará o konkrétní činnost. Jeden může obstarávat logování a druhý API požadavky. [28]

2.7.5 Store

Store je hlavní částí Redux architektury a propojuje všechny části. Je zodpovědný za ukládání stavu, přijímání akcí, předávání akcí prostřednictvím Middleware, výpočet nového stavu pomocí Reducer částí a přenos aktuálního stavu. Je důležité, aby existovala pouze jedna globální instance Store. Všechny akce se musí odesílat v jednom vlákně a nové stavy se musí zpracovávat postupně tak, aby nedocházelo k problémům se stavy. [28]

3 NÁVRHOVÉ VZORY

Při návrhu a implementaci softwaru se setkáváme s řešením podobných typů problémů. Návrhové vzory jsou důležitou částí při této činnosti pro vytváření efektivních, znovupoužitelných a komplexních systémů. Návrhové vzory představují osvědčené a doporučované postupy řešení konkrétního problému. Lze tedy říct, že se jedná o vzorce, podle kterých problém řešíme. Návrhové vzory jsou používány napříč celou oblastí software. Existuje velké množství návrhových vzorů. Správné zvolení a použití těchto vzorů šetří čas například i z důvodu úprav implementace v budoucnosti.

Obecně se návrhové vzory dělí do tří skupin, a to Creational, Structural a Behavioral.

První zmíněná skupina návrhových vzorů řeší vytváření objektů. Structural skupina se soustředí na organizaci tříd. Poslední skupina se stará o chování tříd a objektů. [29]

V následujících sekcích jsou popsány základní návrhové vzory, které jsou využívány v Apple platformě.

3.1 Singleton

Singleton je návrhový vzor, který zajišťuje, že existuje pouze jedna instance konkrétní třídy. Tato instance je sdílená mezi všechna místa, kde je potřeba. Singleton je často používán ve třídách z knihoven Apple platformy, jako je například FileManager a UserDefaults. Základem při vytváření třídy jako Singleton je nutno mít privátní konstruktor, čímž je zaručena existence pouze jediné instance třídy. [30]

V nadcházejícím kódu lze vidět implementaci tohoto návrhového vzoru v jazyce Swift.

```
1  class AccountSettings {
2      static let shared = AccountSettings()
3      var username: String?
4
5      private init() {}
6  }
```

3.2 Abstract Factory

Tento návrhový vzor řeší problém, který nastává při vytváření objektů produktových rodin, které spolu souvisí bez přesného specifikování jejich konkrétních tříd. Abstract Factory definuje rozhraní pro vytváření všech různých produktů. Vytváření produktu obstarává konkrétní tovární třída. Určitý druh produktu odpovídá určitému typu Factory. [31]

3.3 Adapter

Návrhový vzor Adapter převádí rozhraní jedné třídy na jiné rozhraní, které klientský kód očekává. Pomocí Adapter je umožněna spolupráce tříd, které by jinak nemohly spolupracovat skrz nekompatibilní rozhraní. Vzor odděluje kód klienta od cílového objektu. Pro tento vzor využívá Apple ve svých knihovnách Protocol a je velmi často využíván. Příkladem může být UITableViewDelegate. Pomocí Adapter vzoru lze v praxi docílit spolupráce starého kódu s novým moderním kódem, což v případě velkých projektů je důležité. Tento návrhový vzor zjednodušeně propojuje dva odlišné objekty. [32] [33]

3.4 Command

Command zapouzdřuje požadavek do samostatného objektu, který obsahuje všechny informace o daném požadavku. Tím je umožněno uchovávat historii požadavku, řadit požadavky a podporovat operace, které lze vrátit. Vzniklý objekt z požadavku je propojen s jednou či více akcí na specifickém přijímači. Tento vzor odděluje vzniklý požadavek od objektů, které tento požadavek přijímají a vykonávají. [33]

3.5 Target-Action

Návrhový vzor Target-Action je hojně využíván v knihovnách Apple platformy. Jak z názvu vyplývá, skládá se z části Target a Action. Vzor umožňuje aby řídící objekt, což může být objekt jako tlačítko, posuvník či textové pole, zaslal zprávu jinému objektu. Cílový objekt zaslanou zprávu interpretuje a zpracuje ji jako instrukci specifickou pro aplikaci. Zasílaná zpráva se nazývá jako akční zpráva a je řešena pomocí selektoru, což je jedinečný runtime identifikátor metody. [33]

3.6 Key-value observing

Key-value observing je specifický návrhový vzor pro Cocoa knihovny, který slouží pro notifikování objektů ohledně změn vlastností jiného objektu. Vzor je využitelný pro komunikování změn mezi logicky oddělenými celky aplikace jako je komunikace mezi Modelem a View. Tento vzor lze použít pouze pro třídy které dědí z NSObject. V následující části je popsána implementace tohoto vzoru.

Proměnné, které chceme sledovat pomocí Key-Value observing je nutno označit pomocí @objc atributu a dynamic modifikátoru. V následujícím kódu lze vidět definici ObjectToObserve třídy s proměnnou date, kterou lze sledovat.

```
class ObjectToObserve: NSObject {
    @objc dynamic var date = NSDate(timeIntervalSince1970: 0)
    func update() {
        date = date.addingTimeInterval(20)
    }
}

```

Instance třídy observeru, neboli pozorovatele, spravuje informace o změnách, které byly provedeny v jedné či více proměnných. Při vytvoření observeru je nastartováno pozorování pomocí volání observe(_:options:changeHandler:) metody s key path hodnotou, která referuje na proměnnou, kterou chceme pozorovat.

```
class MyObserver: NSObject {
    @objc var objectToObserve: ObjectToObserve
    var observation: NSKeyValueObservation?

    init(object: ObjectToObserve) {
        objectToObserve = object
        super.init()

        observation = observe(
            \.objectToObserve.date,
            options: [.old, .new]
        ) { object, change in
            print("\(change.oldValue) \(change.newValue)")
        }
    }
}
```

V předchozí ukázce lze vidět key path \.objectToObserve.date, která referuje na proměnnou date z třídy ObjectToObserve. Jak lze vidět v ukázce, v kódu lze použít oldValue a newValue hodnoty NSKeyValueObservedChange struktury na základě kterých lze zjistit informace, jak se změnila sledovaná hodnota.

Následně jsou tyto dvě třídy propojeny tím, že objekt třídy ObjectToObserve je předán v inicializaci třídy MyObserver.

```
let observed = ObjectToObserve()
let observer = MyObserver(object: observed)
observed.update()
```

Objekty, které jsou nastaveny pro pozorování pomocí Key-Value observing tak, jak je uvedeno v ukázkách, notifikují své pozorovatele o svých změnách. V případě volání update

metody v ukázce toto volání automaticky spustí obsluhu změny pozorovatele. Tím by byla vypsána stará a nová hodnota proměnné date do konzole. [34]

3.7 Repository pattern

Tento návrhový vzor je určen k oddělení logiky pro práci s daty a logikou řešící získání dat. Obecně bývá vytvořen protokol, který definuje funkce pro získání dat. Může být tedy typicky funkce pro získání, smazání, přidání a podobně. Následně tento protokol implementuje třída repozitáře a je pouze na konkrétní třídě, jak tyto data zajistí. Může jít například i získání dat z API nebo z lokální databáze. To ale řeší konkrétní repozitář a neovlivňuje další kód například uživatelského rozhraní. Uživatelské rozhraní skrz různé třídy následně pouze volá funkce, které definuje protokol. Tím je zaručena abstrakce od logiky řešící získání dat. Další výhodou je také možnost jednoduchého změnění repozitáře za jiný. [35]

4 TESTOVÁNÍ V IOS

Testování mimo jiné ověřuje korektní chování aplikace v testovaných případech. Testování je nedílnou součástí života softwaru. Selhání aplikace může mít důležitý důsledek, jako například ztráta důvěry, naštvání uživatele aplikace a ztráta peněz. Chyby v aplikaci mohou nastat z různých důvodů, a to například z důvodu lidského faktoru, změnou technologie, časového vytížení a neznalosti aplikace. K řešení a předcházení těchto problémů lze využít následující testy.

4.1 UI testy

Pomocí testů uživatelského rozhraní zjišťujeme, zda se uživatelské rozhraní chová tak jak by mělo a zda jsou předpokládané akce provedeny. Tyto testy interagují s aplikací stejně, jako by aplikaci používal reálný uživatel. Uživatelské testy jsou pouze o zkoušení a interagování s viditelnými prvky na obrazovce pomocí testů. To nám ověřuje a validuje, zda se uživatelské rozhraní chová tak jak očekáváme.

UI testy nám mohou otestovat chování aplikace v různých variantách velikosti zařízení, což enormně ušetří čas při testování různých cílových zařízení oproti manuálnímu průchodu aplikace. Testy nám umožní automatizovat činnost, která by jinak musela být prováděna manuálně. Pokud je aplikace pokrytá testy uživatelského rozhraní, tak lze po každé změně v kódu tyto testy spustit a ověřit, zda se aplikace chová korektně jako před novou změnou v kódu. Další možností je zapojení testů do distribučního procesu, kdy jsou testy před distribucí aplikace spuštěny a zkontrolovány. Tím ověříme, že je aplikace pro uživatele v co nejvíce funkčním stavu a v aplikaci v testovaných případech se nevyskytuje fatální chyba. U velkých projektů jsou tyto testy mnohem více uplatnitelné než v případě malých aplikací, které mají pouze pár obrazovek. V případě velkých aplikací, které mají mnoho obrazovek, manuální otestování průchodu aplikace vyžaduje markantní časové nároky.

Základní knihovnou, kterou nám Apple poskytuje pro tyto účely je XCTest. Základním principem je popsání interakce s uživatelským rozhraním v kódu a následným porovnáním předpokládaného stavu rozhraní. Pokud je předpokládaný stav stejný jako výsledek po provedení testu, tak test prošel. V případě odlišného stavu test neprošel a víme, že je v aplikaci problém. [36] [37]

4.2 Unit testy

Unit testy zajišťují to, že napsaný kód funguje tak jak očekáváme. Na základě zadaného vstupu do kódu očekáváme specifický výstup z kódu. Tyto testy jsou automatizované testy, které spouští a validují část kódu. Výsledkem je ujištění, že se kód chová tak, jak byl původně navržen. Opět jako u předchozího typu testů lze tyto testy spouštět před distribucí aplikace a ověřit tím funkčnost.

Unit testy nám pomohou snížit nutnost manuálního testování kódu. Stejně jako v případě UI testů, tak v případě úpravy kódu lze pomocí unit testů ověřit, zda kód funguje jako před úpravou. Zároveň nám mohou pomoct při reprodukci specifických chyb a při hledání chyb v souvislosti s pamětí. Unit test je v podstatě jen funkce, která vyvolá určenou část našeho kódu a následně je vyhodnoceno, zda proběhla požadovaná věc. Funkce pro tento typ testování se nachází stejně jako v případě testů uživatelského rozhraní v knihovně XCTest. [38] [39]

4.3 Performance testy

Rychlost aplikace je důležitou vlastností, která ovlivňuje zážitek uživatele z aplikace. Uživatelé nemají rádi aplikace, které jsou pomalé a pomalu reagují na jejich interakci. Pro měření rychlosti aplikace lze využít knihovnu XCTest. Knihovna umožňuje měřit výkonnost části kódu uvnitř UI či Unit testu.

Pokud chceme měřit určitý kód, tak se měření daného bloku kódu provede několikrát v závislosti na nastavené hodnotě. V základu se toto měření provede desetkrát. Následně se čas zprůměruje, a tato hodnota se používá pro vyhodnocení v dalším testu výkonnosti. Samozřejmostí je možnost nastavení této hodnoty manuálně. Pokud následující test trvá déle než určená hodnota, tak je test vyhodnocen negativně.

V performance testech je potřeba dávat pozor na to, co testuje. V případě testování zpracování náhodných dat nemusí být výsledek testu adekvátní oproti stejným datům pro každý běh testu. [40]

4.4 Statická analýza kódu

Často využívaným nástrojem pro statickou analýzu kódu je nástroj SwiftLint. Tento nástroj dohlíží v kódu na styl a konvence jazyka Swift. Tím je zaručena konzistence a udržovatelnost kódu. V případě většího týmu vývojářů pracujících na stejném projektu může

nastat situace, kdy každý vývojář může napsat stejný kód různými způsoby. Tím může být velmi ztížena údržba a případné hledání chyb v kódu. O dodržování stejného stylu a konvencí se stará právě tento nástroj. Nástroj umožňuje vlastní definici pravidel, které následně v kódu hlídá. V pravidlech lze nastavit například ignorování specifických souborů nebo maximální délku zdrojového souboru. Výhodou je, že i nově příchozí vývojář který nezná styl kódu v dané firmě či projektu, je tímto nástrojem automaticky pohlídán, aby dodržoval pravidla. SwiftLint lze začlenit přímo do Xcode a díky tomu lze vidět varování a chyby přímo v IDE. Tento nástroj oproti ostatním druhům testů v této kapitole není vytvořen společností Apple.

V případě nasazení tohoto nástroje do projektu, který již obsahuje zdrojové kódy lze využít příkaz autocorrect. Zavoláním tohoto příkazu provede SwiftLint automatickou opravu kódu dle definovaných pravidel, což velmi zjednodušuje integraci, do již existujícího projektu. [41]

II. PRAKTICKÁ ČÁST

5 APLIKAČNÍ POŽADAVKY

Tato kategorie popisuje funkcionální a nefunkcionální požadavky pro demo aplikaci.

5.1 Funkcionální požadavky

Tabulka 1 Funkcionální požadavky

ID Požadavku	Popis
01	Aplikace musí umožnit přihlášení
02	Aplikace musí umožnit odhlášení
03	Aplikace musí umožnit zobrazení seznamu servisních událostí
04	Aplikace musí umožnit zobrazení detailu servisní události
05	Aplikace musí umožnit provést hlášení stavu servisní události
06	Aplikace musí umožnit přidat k hlášení stavu servisní události fotografie
07	Aplikace musí umožnit zobrazit detail zákazníka
08	Aplikace musí umožnit zobrazit informace přihlášeného uživatele

5.2 Nefunkcionální požadavky

Tabulka 2 Nefunkcionální požadavky

ID Požadavku	Popis
01	Aplikace musí používat jazyk Swift
02	Aplikace musí podporovat verzi iOS 16.0 a novější
03	Aplikace musí podporovat režim zobrazení typu portrait
04	Aplikace musí podporovat zařízení typu iPhone

6 POPIS DEMO APLIKACE

Kapitola popisuje demo aplikaci, která byla použitá pro implementaci ve dvou různých architekturách.

6.1 Obecný popis

V dnešní době se společnosti zaměřují na digitalizaci svých procesů. Jako demo aplikace byla zvolena aplikace pro servisního technika. Digitalizace servisního technika je ideálním smysluplným příkladem pro tuto aplikaci. Technik potřebuje při své práci evidovat konkrétní činnosti spojené s jeho prací. Samozřejmostí jsou také informace o servisních událostech spojené s lokací a kontaktem. V závěru je pro technika důležité, aby informace ze servisní události uložil do systému. Výhodou této aplikace je následná minimalizace papírových dokumentů. Servisní technik je většinu času v terénu a mobilní aplikace, která napomáhá jeho činnosti je ideálním případem.

6.2 Technologie

Aplikace byla implementována pro platformu iOS. Pro vývoj bylo použito vývojové prostředí Xcode od společnosti Apple. Jako programovací jazyk byl zvolen Swift s knihovnou SwiftUI. SwiftUI je relativně nová knihovna, která je určena pro vývoj uživatelských rozhraní pro Apple platformu. Tato knihovna používá deklarativní syntaxi a vývoj oproti knihovně UIKit může být rychlejší a jednodušší. SwiftUI se stává stále více používanou pro vývoj aplikací. Velmi často lze vidět, že je tato knihovna pro Apple důležitější do budoucna než UIKit. Z těchto zmíněných důvodů jsem zvolil implementaci pomocí SwiftUI. Aplikace je celá situována do tmavého režimu.

6.3 Ukázka uživatelského rozhraní aplikace

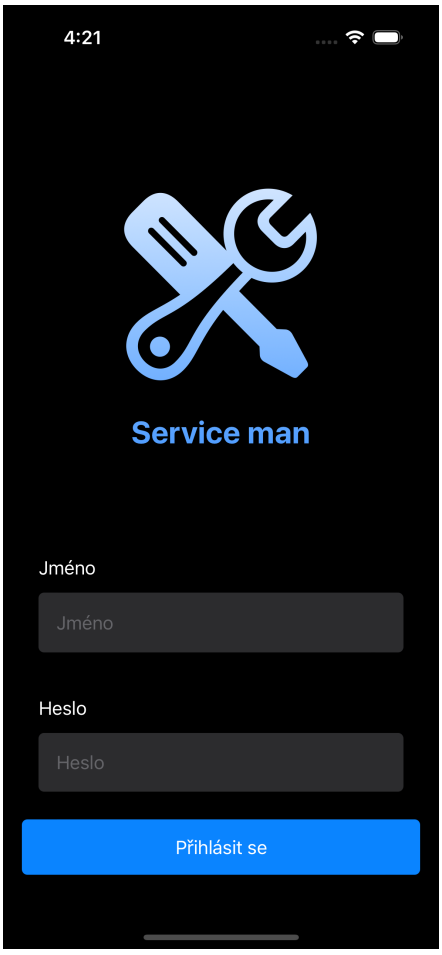
Následující podkapitoly ukazují implementované uživatelské rozhraní aplikace.

6.3.1 Přihlašovací obrazovka

Po spuštění aplikace je zobrazena přihlašovací obrazovka, která je klíčovým prvek uživatelského dojmu, jelikož se jedná o první interakci mezi uživatelem a aplikací. Obrazovka zajišťuje bezpečné přihlášení do aplikace servisního technika.

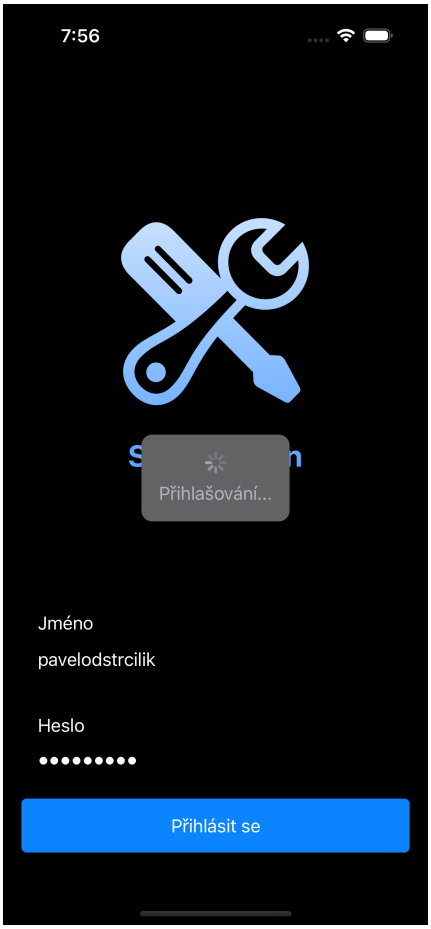
Hlavním cílem je poskytnutí přihlášení do svého účtu. Registrace není v aplikaci implementována, jelikož je předpoklad, že v případě skutečné implementace by byla správa uživatelů buď v externím systému, či v aplikaci sloužící dispečerovi.

V obrazovce se nachází jednoduché logo a dva textové vstupy. První textový vstup je určen pro přihlašovací jméno uživatele a druhý vstup je určen pro heslo uživatele. Pod těmito vstupy se nachází tlačítko s nápisem Přihlásit se, které vyvolá akci k ověření, zda zadané údaje uživatele jsou správné a poté případně je uživatel autorizován. Interakční prvky na této obrazovce jsou ve spodní části z důvodu lepší uživatelské zkušenosti, kdy lze ideálně na tyto prvky dosáhnout pomocí palce. Samozřejmostí je také spodní odsazení tlačítka od domovského indikátoru.



Obrázek 7 Přihlašovací obrazovka

Po stisknutí tlačítka pro přihlášení je zobrazen načítací indikátor, který informuje uživatele a probíhající akci a zablokuje uživatelskou interakci s uživatelským rozhraním.



Obrázek 8 Přihlašovací obrazovka s indikátorem načítání

6.3.2 Seznam servisních událostí

Po přihlášení se zobrazí seznam servisních událostí. Seznam slouží k evidenci veškerých servisních událostí a jedná se o základní přehled. Tento seznam informuje technika o tom, jaké servisní události by měl vyřešit. V seznamu jsou ke každé události zobrazeny důležité informace.

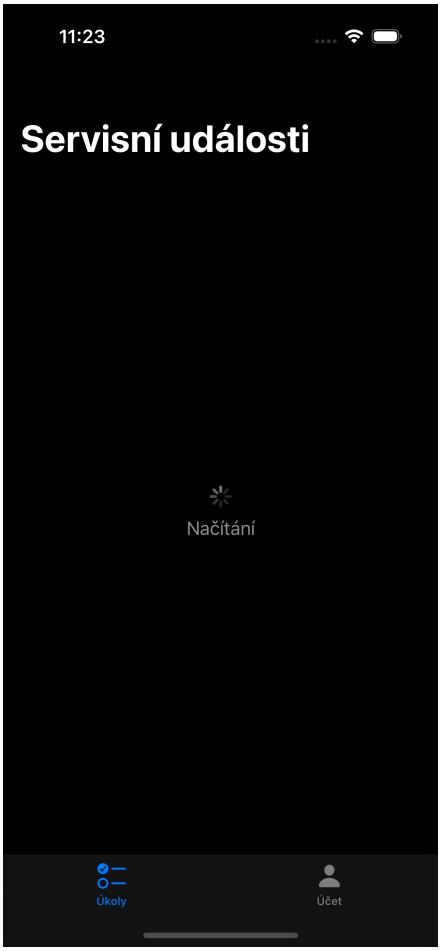
V buňce je jako nejvýraznější prvek zobrazeno datum servisní události, které se nachází v levé části. V pravé části se nachází popis servisní události a adresa, kde se servisní událost nachází. V neposlední řadě je zobrazena společnost, u které servisní událost nastala.

V buňce jsou zobrazeny pouze minimální ale důležité informace tak, aby uživatel měl přehled o daných servisních událostech. Cílem je, aby uživatelské zobrazení neobsahovalo příliš mnoho informací, které by jej vytvářely nepřehledné.

Uživatel může pomocí kliknutí na buňku přejít na detail servisní události. Možnost této akce je i indikována šipkou, která se nachází na pravé straně buňky. Tato šipka je základní

funkcionalitou ve SwiftUI při indikování možnosti přechodu na detail a vývojář ji nemusí přímo deklarovat.

Při načítání seznamu je zobrazen indikátor načítání.



Obrázek 9 Servisní události během načítání

Seznam událostí také podporuje gesto pull to refresh, které znovu načte seznam servisních událostí.

Ve spodní části obrazovky se nachází Tab Bar komponenta, která obsahuje dvě položky, a to Úkoly, což je seznam událostí a Účet. Druhá zmíněná položka je popsána v jedné z následujících kapitol.

Obrázek 10 Obrazovka servisních událostí

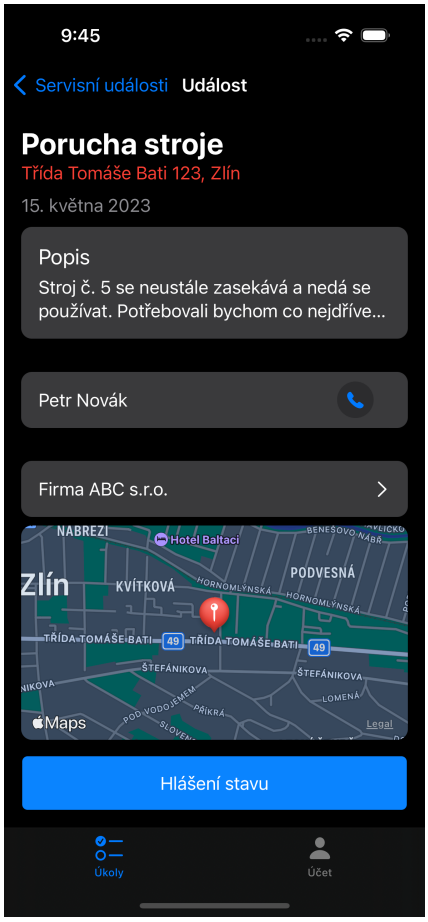
6.3.3 Detail servisní události

Detail servisní události poskytuje podrobné informace o konkrétní servisní události. Obrazovka rozšiřuje informace uvedené v buňce na seznamu událostí.

Servisní technik v horní části vidí informaci o názvu události a adresu události. Samozřejmostí je také zobrazení datumu a popisu události. Popis umožňuje technikovi zjistit důležité informace o události a pomoci s řešením. Dále se na obrazovce nachází tlačítko s kontaktní osobou. Po kliknutí na toto tlačítko je číslo zavoláno. Servisní technik tak může jednoduše kontaktovat kontaktní osobu k dané servisní události.

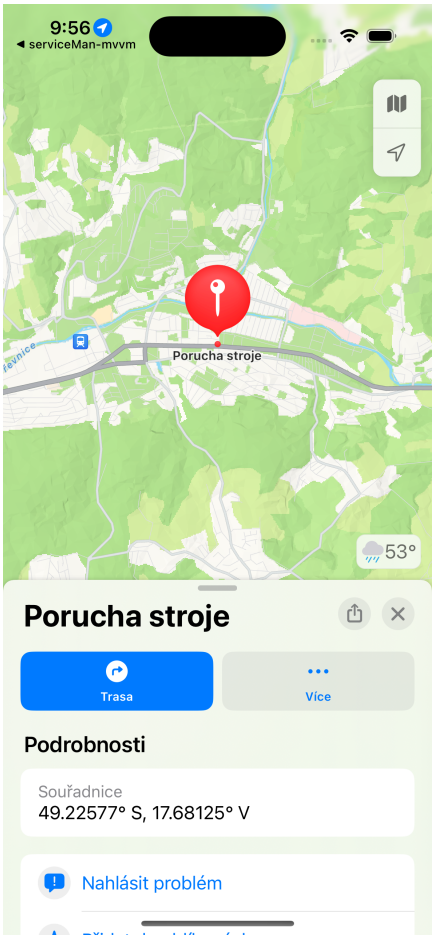
Mezi dalšími zobrazenými prvky je tlačítko s názvem společnosti, které zobrazí další obrazovku. Tato obrazovka je popsána posléze.

Ve spodní části se nachází část mapy, ve které je zobrazena lokace servisní události. Lokace je zvýrazněna pomocí takzvaného pinu uprostřed mapy. Pin je tedy centrován a je zobrazeno okolí.



Obrázek 11 Detail servisní události

Po kliknutí na mapu je zobrazena lokace v nativní aplikaci Mapy s názvem události. Ukázka lze vidět v následujícím obrázku. Tato funkcionalita je určena pro jednoduché zjištění lokace a případného zapnutí navigace na servisní událost. Uživatel nemusí kopírovat adresu a vkládat ji do jiné mapové aplikace a tím je proces zjednodušen a zpříjemněn pro uživatele. Servisní technik tak dokáže zapnout navigaci na místo události velmi rychle a bez zbytečných kroků.



Obrázek 12 Zobrazení servisní události v aplikaci Mapy

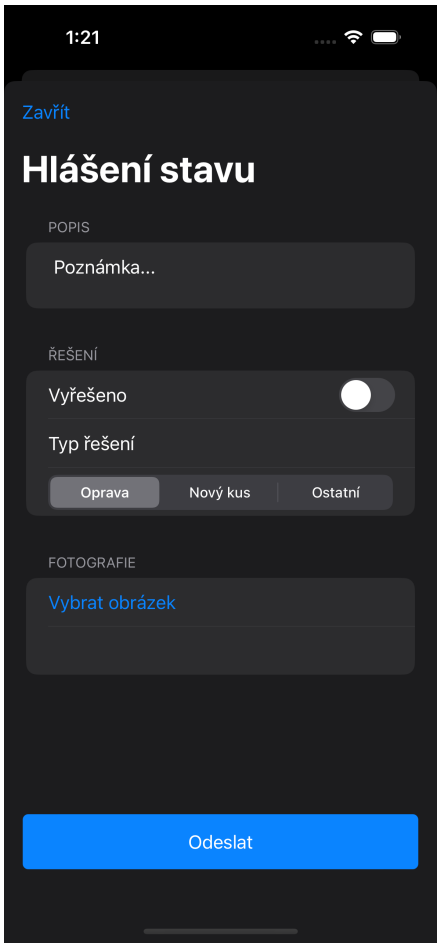
V dolní části obrazovky detailu servisní události je zobrazeno tlačítko pro hlášení stavu. Po stisku tlačítka je zobrazena další obrazovka, která je popsána posléze.

6.3.4 Hlášení stavu

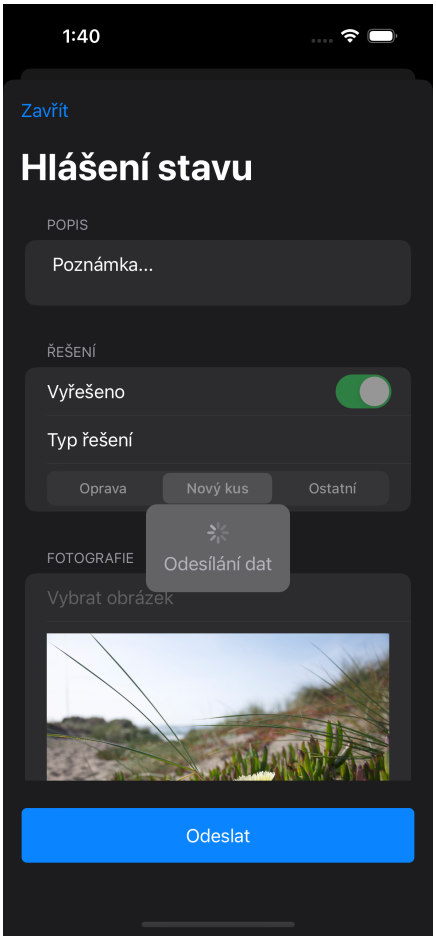
Z detailu servisní události se lze prokliknout na hlášení stavu. Hlášení stavu je důležitou částí této ukázkové aplikace. Servisní technik do této obrazovky zapisuje informace o dané servisní události. Obrazovka je prezentována modálně. V levé horní části se nachází tlačítko pro zavření této obrazovky. Vstupy na této obrazovce jsou rozděleny do tří sekcí tak, aby byly vstupy přehledné a oddělené.

Servisní technik v této obrazovce po ukončení servisní události zapíše textový popis události. V druhé sekci technik pomocí přepínače stanoví, zda byla servisní událost dokončena anebo zda bude potřeba další práce. Dále pomocí přepínacího segmentového prvku určí typ řešení. Zde může zvolit opravu, výměnou za nový kus anebo ostatní. Na závěr technik může připojit libovolné množství fotografií, které evidují jeho činnost a další informace. Přidání fotografií k hlášení stavu může být přidanou hodnotou této aplikace. Poté

po stisku tlačítka odeslat je zobrazen indikátor a uživatelské rozhraní je zablokováno. Po dokončení odeslání dat je tento modál skryt.



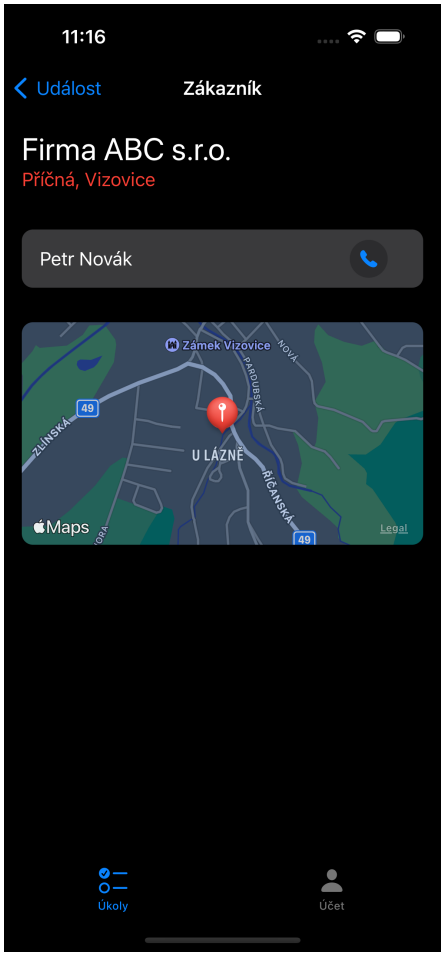
Obrázek 13 Hlášení stavu



Obrázek 14 Hlášení stavu při odesílání dat

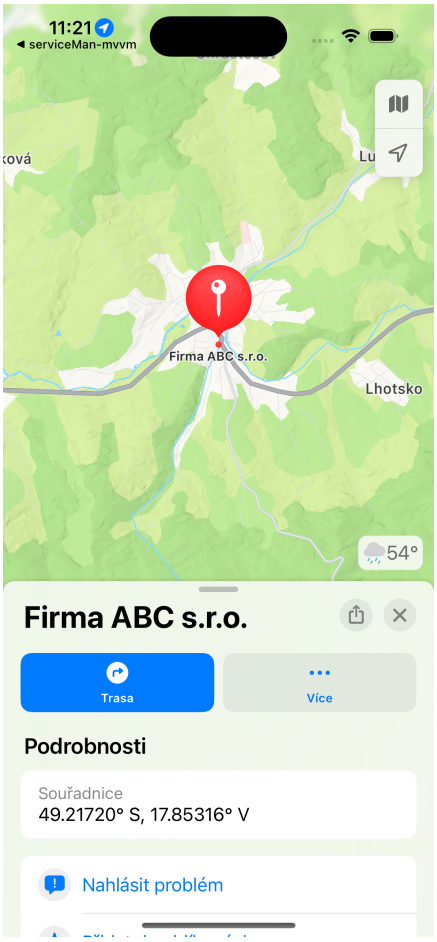
6.3.5 Detail zákazníka

Z detailu servisní události se lze prokliknout na detail zákazníka. Tato obrazovka informuje uživatele o konkrétním zákazníkovi. Na obrazovce se nachází informace o jméně zákazníka a adrese. Další důležitou informací je také kontaktní osoba, které je ve formě tlačítka. Po stisku tohoto tlačítka je zavoláno konkrétní osobě. Podobná funkcionalita je i v detailu servisní události. Tlačítko pro tuto akci vypadá stejně a je tím zachována konzistence v aplikaci.



Obrázek 15 Detail zákazníka

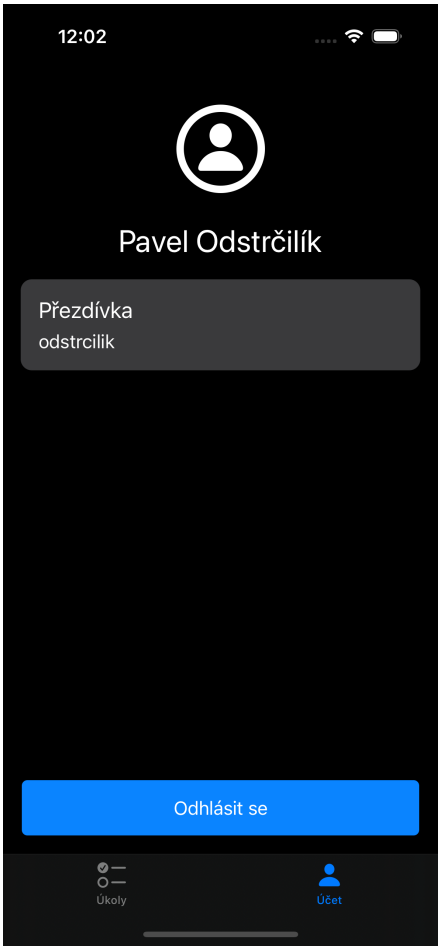
Jako poslední se na této obrazovce zobrazuje mapa s adresou zákazníka. Po kliknutí na mapu je zobrazena lokace v nativní aplikaci Mapy stejně jako v případě otevření mapy v detailu servisní události. Při otevření v aplikaci Mapy lze vidět lokace a název firmy, což lze vidět v následujícím obrázku.



Obrázek 16 Zobrazení zákazníka v aplikaci Mapy

6.3.6 Účet

Po kliknutí na záložku účet, která se nachází v komponentě Tab Bar, se zobrazí následující jednoduchá obrazovka. Na obrazovce se nachází informace o přihlášeném uživateli, to je jméno, příjmení a jeho přezdívka. V dolní části obrazovky se nachází tlačítko sloužící k odhlášení uživatele. Po stisknutí tohoto tlačítka je uživatel odhlášen a je zobrazena obrazovka sloužící pro přihlášení.



Obrázek 17 Obrazovka účet

7 IMPLEMENTACE APLIKACE V MVVM

Jako první architekturu pro implementaci ukázkové aplikace jsem zvolil architekturu MVVM. Tato architektura má několik výhod. MVVM relativně dobře spolupracuje se SwiftUI a tato architektura je často používaná při implementaci aplikací pro Apple platformu. Mezi další výhodu patří její relativní jednoduchost a jednoduché pochopení pro méně zkušené vývojáře.

7.1 Implementace modelů

Tato podkapitola popisuje jednotlivé modely, které byly použity pro implementaci demo aplikace.

7.1.1 Model UserCredentials

Struktura UserCredentials obsahuje dvě proměnné, a to name a password. Obě proměnné jsou datového typu String. Model je využíván v obrazovce pro přihlášení.

```
1  struct UserCredentials {
2      var name: String
3      var password: String
4  }
```

7.1.2 Model User

Tento model definuje třídu User, která implementuje protocol ObservableObject. Třída obsahuje proměnné pro hodnotu id, jména, příjmení a přezdívky a jsou typu String. Jednotlivé proměnné této třídy jsou označeny jako @Published, čímž je zajištěna automatická notifikace uživatelského rozhraní při změně. Tento model se využívá jako takzvaný EnvironmentObject a je v něm uložen aktuálně přihlášený uživatel. Objekt, který využívá sdílení objektu pomocí EnvironmentObject musí splňovat protokol ObservableObject. Z tohoto důvodu je zde využita třída namísto struktury, jelikož tento protokol lze aplikovat pouze na třídu. Dále v tomto modelu je rozšíření této třídy o statickou proměnnou sampleData, která poskytuje testovací údaje potřebné například při využití živého náhledu uživatelského rozhraní v Xcode.

```
class User: ObservableObject {
    @Published var id: String = ""
    @Published var username: String = ""
    @Published var firstName: String = ""
    @Published var lastName: String = ""
}

extension User {
    static var sampleData: User {
        let user = User()
        user.id = "10"
        user.username = "Pavel"
        user.lastName = "Odstrčilík"
        user.username = "odstrcilik"
        return user
    }
}
```

7.1.3 Model Customer

Model Customer je určen pro reprezentaci zákazníka. Model obsahuje konstanty pro id, název, adresu, telefonní číslo a zeměpisné souřadnice. Opět z již zmíněných důvodu je struktura Customer rozšířená o statickou proměnnou sampleData.

```
struct Customer {
    let id: String
    let name: String
    let address: String
    let contactName: String
    let telephoneContact: String
    let latitude: CLLocationDegrees
    let longitude: CLLocationDegrees
}

extension Customer {
static let sampleData: Customer = Customer(id: "1", name:
 "Firma ABC s.r.o.", address: "Lísková 456, Vizovice",
 contactName: "Petr Novák", telephoneContact: "777 123 456",
 latitude: 49.23062, longitude: 17.6575)
}
```

7.1.4 Model ServiceEvent

Nejdůležitějším modelem aplikace je struktura ServiceEvent. Model reprezentuje servisní událost. ServiceEvent obsahuje konstanty pro id, název servisní události, popisu servisní události, adresu servisní události, zákazníka, datum servisní události, jméno kontaktní osoby, telefonní kontakt, a zeměpisnou šířku a délku servisní události. Dále je struktura rozšířená o statické pole hodnot typu ServiceEvent sloužící jako testovací data.

```swift
struct ServiceEvent: Identifiable {
    let id: String
    let name: String
    let description: String
    let address: String
    let customer: Customer
    let date: Date
    let contactName: String
    let telephoneContact: String
    let latitude: CLLocationDegrees
    let longitude: CLLocationDegrees
}

extension ServiceEvent {
    static let sampleData: [ServiceEvent] =
    [
        ServiceEvent(id: "1", name: "Porucha stroje",
 description: "Stroj č. 5 se neustále zasekává a nedá se
 používat. Potřebovali bychom co nejdříve opravit.", address:
 "Třída Tomáše Bati 123, Zlín", customer: Customer(id: "1",
 name: "Firma ABC s.r.o.", address: "Vizovice", contactName:
 "Petr Novák", telephoneContact: "+420777123456", latitude:
 49.23062, longitude: 17.6575), date: Date(), contactName:
 "Jan Novotný", telephoneContact: "602 555 666", latitude:
 49.23062, longitude: 17.6575)
// . . .
    ]
}
```

7.1.5 Model ServiceEventCompletionTypeEnum

Tento model je definován jako výčtový typ a je určen k reprezentaci existujících typů dokončení servisní události. Hodnota repaired určuje, že byla servisní událost vyřešena opravou, hodnota newPiece říká, že událost byla vyřešena výměnou za nový kus. Jako poslední se v tomto výčtovém typu vyskytuje hodnota other, která určuje jiný druh vyřešení události. Tento model tedy specifikuje, jakým způsobem byla servisní událost vyřešena. Tento model se využívá v následujícím modelu.

```swift
enum ServiceEventCompletionTypeEnum: String, CaseIterable {
    case repaired = "Oprava"
    case newPiece = "Nový kus"
    case other = "Ostatní"
}
```

7.1.6 Model ServiceEventCompletion

Tato struktura slouží pro reprezentaci informací o dokončení servisní události. ServiceEventCompletion obsahuje konstantu id a proměnné pro id servisní události, zda jde servisní událost dokončena, poznámku, typ dokončení a pole obrázků. Struktura implementuje protokol Identifiable.

```
struct ServiceEventCompletion: Identifiable {
    let id: String
    var serviceEventId: String
    var isCompleted: Bool
    var note: String
    var selectedItemCompletionType:
 ServiceEventCompletionTypeEnum
    var selectedImage: [UIImage]
}
```

7.1.7 Model Marker

Struktura Marker obsahuje konstantu id typu UUID a proměnnou pro uložení lokace typu MapMarker. Tato struktura se využívá pro zobrazení mapy pro uživatele.

```
struct Marker: Identifiable {
    let id = UUID()
    var location: MapMarker
}
```

7.2 Rozšíření

V projektu jsou využity dvě rozšíření, které přidávají novou funkcionalitu do již existující funkcionality.

První rozšíření rozšiřuje strukturu Color o barvu darkRed, kterou jsem si sám definoval jako Color Set v Assets. Výhodou této definice v Assets je možné zobrazení barvy přímo v uživatelském rozhraní vývojového prostředí.

Obrázek 18 Definovaná barva DarkRed

Dále je tato struktura rozšířena o barvy z objektu UIColor, které jsou zaobalené do struktury Color. Toto bylo provedeno z důvodu lepší práce s konkrétními barvami.

```
extension Color {
    static let darkRed = Color("DarkRed")

    static let systemGray2 = Color(UIColor.systemGray2)
    static let systemGray4 = Color(UIColor.systemGray4)
    static let systemGray5 = Color(UIColor.systemGray5)
}
```

Druhé rozšíření rozšiřuje třídu DateFormatter o statickou proměnnou shortDayMonth. Tento formáter slouží pro nastavení formátu datumu. Formát „d.M“ znamená, že bude datum zobrazeno ve formátu „den.měsíc“, což může být například 15.5. Proměnná vrací DateFormatter a je napsána pomocí takzvaného closure bloku.

```
extension DateFormatter {
    static let shortDayMonth: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateFormat = "d.M."
    return formatter
    }()
}
```

7.3 Vstupní bod aplikace

Struktura serviceMan_mvvmApp je označena jako @main a implementuje protokol App. Tím je označen vstupní bod aplikace. Tato struktura zobrazuje BaseView. Ve struktuře BaseView se nachází instance třídy LoginViewModel. Tato třída je popsána posléze.

Pokud je uživatel přihlášen, tak je vytvořena a zobrazena instance MainTabView a zároveň je nastaven environmentObject s hodnotou přihlášeného uživatele. Tato hodnota je předána

z ViewModelu. Tím je velmi jednoduše zajištěno předání přihlášeného uživatele do všech hierarchicky podřízených podhledů MainTabView.

Pokud uživatel není přihlášen, což lze zjistit z proměnné viewModel, tak je zobrazen LoginView s parametrem proměnné viewModel.

```
@main
struct serviceMan_mvvmApp: App {
    var body: some Scene {
        WindowGroup {
            BaseView()
        }
    }
}


struct BaseView: View {
    @StateObject var viewModel = LoginViewModel()

    var body: some View {
        Group {
            if viewModel.isLoggedIn {
                withAnimation {
                    MainTabView(isLogged:
                                    $viewModel.isLoggedIn)

                     .environmentObject(viewModel.loggedUser)
                }
            } else {
                LoginView(viewModel: viewModel)
            }
        }
    }
}

```

7.4 Přihlašovací obrazovka

Přihlašovací obrazovka je reprezentována strukturou LoginView. V této struktuře je proměnná viewModel typu LoginViewModel, která je označená jako @ObservedObject. Tato proměnná je propojená s uživatelským rozhraním. V následující ukázce kódu lze vidět tuto třídu. Třída obsahuje proměnné pro přihlašovací údaje, indikace, zda je uživatel přihlášený, zda probíhá načítání dat a přihlášeného uživatele. Ve ViewModelu se nachází funkce logIn. Tato funkce simuluje dotaz na přihlášení. Na počátku této funkce je nastavena proměnná indikující načítání na hodnotu true a poté je za půl sekundy nastaven přihlášený uživatel. Proměnná isLoggedIn je nastavena na true a poté je zobrazen MainTabView.

ViewModel splňuje protokol ObservableObject a proměnné jsou označeny jako @Published z důvodu automatické aktualizace uživatelského rozhraní.

```
1  final class LoginViewModel: ObservableObject {
2      @Published var credentials: UserCredentials =
    UserCredentials(name: "", password: "")
3      @Published var isLoggedIn = false
4      @Published var loggedUser = User()
5      @Published var isLoading = false
6
7
8      func logIn() {
9          isLoading = true
10         DispatchQueue.main.asyncAfter(deadline: .now() + 0.5)
11          {
12             self.isLoading = false
13             self.isLoggedIn = true
14             self.loggedUser.firstName = "Pavel"
15             self.loggedUser.lastName = "Odstrčilík"
16             self.loggedUser.username = "odstrcilik"
17          }
18     }
19 }
```

V následující ukázce lze vidět vybrané části kódu ze struktury LoginView. Struktura přebírá viewModel jako parametr. Následně lze na vidět na řádku číslo 11 ukázka propojení uživatelského rozhraní TextField s viewModelem. Na řádku 13 se nachází tlačítko, které při stisku tlačítka volá funkci nacházející se ve ViewModelu. V této ukázce lze jednoduše vidět, jak v architektuře MVVM komunikuje View a ViewModel.

```
1  struct LoginView: View {
2      @ObservedObject var viewModel: LoginViewModel
3
4      var body: some View {
5  // ...
6          if viewModel.isLoading {
7              ProgressView("Přihlašování...")
8             // ...
9          }
10 // ...
11         TextField("Jméno", text: $viewModel.credentials.name)
12 // ...
13         Button(action: viewModel.logIn) {
14             Text("Přihlásit se")
15             .frame(maxWidth: .infinity, maxHeight: 50)
16         }
17 // ...
18     }
19 }
```

7.5 TabView

TabView slouží k základní navigaci v aplikaci. Tato komponenta obsahuje dvě záložky, a to úkoly a účet. Jak lze vidět v následujícím kódu, tak tato struktura obsahuje proměnnou pro předání přihlášeného uživatele a proměnnou isLogged, která určuje, zda je uživatel přihlášen. Tato proměnná je dále předána pomocí bindování do AccountView. Zároveň do AccountView je předán přihlášený uživatel pomocí environmentObject. V této struktuře není příliš mnoho kódu, který by byl vázán k architektuře MVVM. Jako ikony jednotlivých tabItem jsou použity systémové ikony.

```
20  struct MainTabView: View {
21      @EnvironmentObject var loggedUser: User
22      @Binding var isLogged: Bool
23
24      var body: some View {
25          TabView {
26              ServiceEventList()
27                  .tabItem {
28                      Label("Úkoly", systemImage: "checklist")
29                  }
30              AccountView(isLogged: $isLogged)
31                  .tabItem {
32                      Label("Účet", systemImage: "person")
33                  }
34                  .environmentObject(loggedUser)
35
36          }
37          .preferredColorScheme(.dark)
38      }
39  }
```

7.6 Seznam servisních úkolů

Struktura, která zaobaluje celé uživatelské rozhraní pro seznam servisních úkolů je pojmenována jako ServiceEventList. Prvky uživatelského rozhraní jsou provázány s ViewModelem typu ServiceEventListViewModel. Instance tohoto ViewModelu je vytvořena při inicializaci ServiceEventList.

V následující ukázce lze vidět část, kde je využita SwiftUI komponenta List, která je určena pro prezentaci uspořádaných řádkových dat. Tyto řádky reprezentují jednotlivé servisní události a jsou získány z proměnné events, která se nachází ve ViewModelu. Dále je použit NavigationLink, který slouží k přechodu na detail servisní události ServiceEventDetailView. Toto View má jako parametr objekt ServiceEvent, který reprezentuje vybranou servisní událost. Následně je jako buňka jednotlivého řádku v listu

zobrazen ServiceEventCellView s parametrem servisní události. Nejdůležitější částí je pro uživatele poslední zmíněné View, jelikož jej uživatel vidí. Zbylé popsané věci jsou určeny pro navigaci do detailu servisního úkolu.

Při zobrazení ServiceEventList je pomocí metody onAppear zavolána akce fetchData na ViewModelu. Tato funkce nám zajistí získání dat pro seznam servisních událostí. Zde je ideální situace na poukázání toho, že tato logika se nachází ve ViewModelu a není ve View.

```
@StateObject var viewModel = ServiceEventListViewModel()
// ...
 List(viewModel.events) { serviceEvent in
      NavigationLink(destination:
         ServiceEventDetailView(
             serviceEvent: serviceEvent)) {
          ServiceEventCellView(event: serviceEvent)
                   .listRowInsets(.init(top: 0, leading: 0,
                                        bottom: 10, trailing: 0))
                    .listRowBackground(Color.clear)
           }
           .listRowSeparator(.hidden)
 }
 // ...
.onAppear(perform: {
        viewModel.fetchData()
})
```

V již zmíněném ServiceEventListViewModel se nachází proměnná určující, zda se data načítají, která slouží pro zobrazení indikátoru aktivity. Jako další se na tomto místě nachází proměnná events, která obsahuje pole modelů ServiceEvent. Data, které se nachází v této proměnné se následně zobrazují v seznamu servisních událostí skrz List komponentu uživateli.

Dále je zde implementována funkce fetchData, která nám simuluje získání dat z externího systému. V této funkci je vytvořeno zpoždění a po uplynutí definované doby jsou do proměnné events uloženy data ze statické proměnné sampleData z modelu ServiceEvent.

```
final class ServiceEventListViewModel: ObservableObject {
    @Published var isLoading = false
    @Published var events: [ServiceEvent] = []

    func fetchData() {
        isLoading = true
        DispatchQueue.main.asyncAfter(deadline: .now() + 1.5)
        {
            self.events = ServiceEvent.sampleData
            self.isLoading = false
        }
    }
}
```

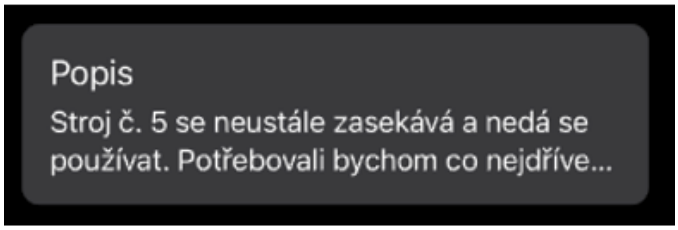
7.7 Detail servisního úkolu

Pro zobrazení detailu servisního úkolu je vytvořena struktura ServiceEventDetailView, která obsahuje proměnou viewModel typu ServiceEventDetailViewModel. Prvky uživatelského rozhraní jsou zde rozloženy ve vertikálním uspořádání pomocí VStack struktury.

```
struct ServiceEventDetailView: View {
    @StateObject var viewModel: ServiceEventDetailViewModel
// ...
    var body: some View {
        VStack(alignment: .leading) {
            // ..
            Text(viewModel.serviceEvent.name)
            // ..
            TextWithBackground(title: "Popis",
                    text: viewModel.serviceEvent.description)
// ..
            ButtonInsideButtonView(title:
              viewModel.serviceEvent.customer.contactName,
              iconString: "phone.fill",
              function: {
                      viewModel.callNumber()
              })
// ...
           TextWithArrow(title:
                      viewModel.serviceEvent.customer.name)
        }
    }
// ...
}
```

Ve zmíněné ukázce lze vidět proměnnou viewModel a ukázku propojení SwiftUI komponenty Text, která zobrazuje text, s ViewModelem určeným pro tuto obrazovku. Jak lze vidět, princip práce s ViewModelem je stále podobný jako v případě předchozích obrazovek, proto je zde ukázáno propojení pouze pár prvků.

Pro vizuální zobrazení popisu servisní události byla vytvořena vlastní komponenta. Pomocí této komponenty, která lze vidět na následujícím obrázku, bylo možné docílit vlastního požadovaného vzhledu a umožnit jednoduchou znovupoužitelnost a tím odstranit zbytečné kopírování stejného kódu. Jako další výhodu komponenty lze zmínit, že v případě úpravy vizuální stránky je komponenta upravena pouze na jednom místě a není nutno upravovat každé místo, kde se tato komponenta používá. Ve výsledku se jedná o jednoduchou strukturu, která ušetřila pracnost při tvorbě uživatelského rozhraní.



Obrázek 19 Komponenta pro popis

Implementace této komponenty lze vidět v následujícím kódu. Jako parametr tato struktura přebírá horní titulek a text, který se zobrazí pod titulkem.

```
struct TextWithBackground: View {
    @State var title: String
    @State var text: String

    var body: some View {
        VStack(alignment: .leading, spacing: 6.0) {
            Text(title)
            .font(.title3)
            Text(text)
        }
        .foregroundColor(.white)
        .frame(minWidth: 0,maxWidth: .infinity,
        alignment: .leading)
        .padding()
        .background(Color.systemGray4)
        .cornerRadius(10.0)
        .preferredColorScheme(.dark)
    }
}
```

Mezi další vlastní komponentu patří tlačítko, které obsahuje další tlačítko. Tato komponenta lze vidět v následujícím obrázku a je využívána pro zavolání na kontakt k servisní události. Motivací pro tuto komponentu byl převážně vlastní vzhled, který se pomocí nativních SwiftUI prvků nepodařil napodobit. Komponenta umožňuje, aby se volala odlišná akce po stisku komponenty a odlišná akce pro tlačítko s ikonou, které se nachází na pravé straně.

V rámci implementace ale nakonec byla využita funkčnost, kdy komponenta po libovolném stisku volá jednu konkrétní akci.

Obrázek 20 Komponenta ButtonInsideButtonView

ButtonInsideButtonView vyžaduje jako parametr titulek, systémovou ikonu a akci, kterou po stisku provede. Tím je zaručena poměrně dobrá znuvupoužitelnost i pro případ rozšiřování aplikace.

Jako další vlastní komponenta byla vytvořena vizuální komponenta, která zobrazuje text na levé straně a na pravé straně se nachází šipka směrem doprava. Tato šipka indikuje uživateli možnost navigace. Komponenta se využívá pro indikaci přechodu na detail zákazníka. Opět byla vytvořena z důvodu znuvupoužitelnosti na více místech v aplikaci. Tato komponenta je pojmenována jako TextWithArrow.

Obrázek 21 Komponenta TextWithArrow

Na pozadí se jedná o velmi jednoduchý kód, kdy je do horizontálního stacku umístněný text a obrázek. Implementace toho je pomocí SwiftUI velmi jednoduchá a příjemná. Jako parametr je vyžadován text, který se následně zobrazí. V následujícím kódu lze vidět kompletní implementaci, kdy je na pár řádcích docíleno požadované chování.

```
struct TextWithArrow: View {
    @State var title: String

    var body: some View {
        HStack {
            Text(title)
            .padding()
            Spacer()
            Image(systemName: "chevron.right")
                .frame(width: 50.0, height: 50.0)
                .foregroundColor(.white)
        }
        .background(Color.systemGray4)
        .cornerRadius(10.0)
        .preferredColorScheme(.dark)
    }
}
```

Poslední použitou vlastní komponentou v detailu servisního úkolu je MapWrapper. Komponenta slouží jako obal pro práci se SwiftUI komponentou pro zobrazení mapy Map a takzvaných annotationItems. V uživatelském rozhraní je uživateli zobrazena mapa a takzvaný pin, který vizualizuje konkrétní adresu v mapě. Toto je docíleno pomocí již zmíněných annotationItems. SwiftUI bohužel neposkytuje příliš dobrou podporu pro docílení této vlastnosti a z toho důvodu byla vytvořena tato komponenta, která jako parametr přebírá datový typ MKCoordinateRegion a pak následně automaticky vytvoří Map s pinem a tím je velmi usnadněna práce a znovu použitelnost. V následující ukázce lze vidět, že v případě dotyku na tento prvek je zavolána funkce openAppleMaps, která se nachází ve ViewModelu. ViewModel pro tuto obrazovku bude popsán posléze.

```
MapWrapper(coordinateRegion: $viewModel.region)
    .frame(height: 200)
    .cornerRadius(10.0)
    .onTapGesture {
        viewModel.openAppleMaps()
    }
```

Implementace lze vidět v následujícím kódu. Proměnná markers obsahuje inicializační hodnoty, které jsou následně přepsány daty z ViewModelu. Zde je i využít model Marker. Z osobních pocitů zde bohužel SwiftUI působí velmi nedokončeně a vývojář musí pro docílení zmíněné konkrétní situace vytvářet vlastní komponentu.

```
struct MapWrapper: View {
    @Binding var coordinateRegion: MKCoordinateRegion
    @State var markers = [Marker(location:
 MapMarker(coordinate: CLLocationCoordinate2D(latitude:
           -25.342863,
           longitude: 131.036974), tint: .blue))]

    var body: some View {
        let region = Binding (
            get: {
                self.coordinateRegion
            },
            set: { newValue in
                DispatchQueue.main.async {
                    self.coordinateRegion = newValue
                    markers = [Marker(location:
 MapMarker(coordinate: CLLocationCoordinate2D(latitude:
 coordinateRegion.center.latitude, longitude:
 coordinateRegion.center.longitude), tint: .darkRed))]
                }
            }
        )
        return Map(coordinateRegion: region,
               annotationItems: markers) { marker in
                  marker.location
       }
    }
}

```

Při zobrazení ServiceEventDetailView se zavolá funkce ViewModelu getCoordinatesFromPlace.

```
.onAppear( perform: {
      viewModel.getCoordinatesFromPlace()
})
```

V detailu servisní události se nachází tlačítko Hlášení stavu. Po stisku tohoto tlačítka je změněna logická hodnota proměnné isCompletionViewPresenting z ViewModelu.

```
Button(action:
    {
        viewModel.isCompletionViewPresenting.toggle()
    }) {
        Text("Hlášení stavu")
        .frame(maxWidth: .infinity, maxHeight: 50)
}
```

Na základě hodnoty isCompletionViewPresenting je zobrazována pomocí modální prezentace obrazovka sloužící pro hlášení stavu. Prezentace je tedy řízena na základě stavu hodnoty nacházející se ve ViewModelu.

```
.sheet(isPresented:
    $viewModel.isCompletionViewPresenting) {
// ...
}
```

ViewModel pro tuto obrazovku obsahuje oproti předchozím ViewModelům více funkcí. Tato třída vyžaduje při inicializaci objekt modelu servisní události a mimo jiné implementuje protokol PhoneProtocol a CoordinatesProtocol.

```
final class ServiceEventDetailViewModel: ObservableObject,
 PhoneProtocol, CoordinatesProtocol {
    @Published var serviceEvent: ServiceEvent
    @Published var region = MKCoordinateRegion()
    @Published var isCompletionViewPresenting = false

    init(serviceEvent: ServiceEvent) {
        self.serviceEvent = serviceEvent
    }

    func openAppleMaps() {
        openAppleMaps(region: region,
                  name: serviceEvent.name)
     }

    func callNumber() {
         callToNumber(serviceEvent.telephoneContact)
    }

    func getCoordinatesFromPlace() {
        getCoordinatesFromAddress(serviceEvent.address,
             completion: { (region) in
        guard let region else {
            return
            }
            self.region = region
        })
    }
}
```

Tyto protokoly byly vytvořeny pro znuvupoužitelnost konkrétních funkcionalit, které se využívají na více místech v aplikaci. Jak lze vidět v následující ukázce, PhonoProtocol obsahuje funkci callToNumber. Tato funkce zavolá na telefonní číslo, které je vyžadováno v parametru funkce.

```swift
public protocol PhoneProtocol {
    func callToNumber(_ number: String)
}

public extension PhoneProtocol {
    func callToNumber(_ number: String) {
        guard let url = URL(string: "tel://\(number)"),
              UIApplication.shared.canOpenURL(url) else {
            return
        }
        UIApplication.shared.open(url, options: [:],
 completionHandler: nil)
    }
}
```

Druhý zmíněný protokol obsahuje dvě funkce. První funkce vrací datovou hodnotu typu MKCoordinateRegion. Účelem funkce je převést adresu ve formátu textového řetězce na zmíněnou hodnotu. Toto je využito při práci s mapou. Druhá funkce je určena k otevření nativní aplikace Mapy. Funkce vyžaduje v parametru hodnotu typu MKCoordinateRegion a název bodu. Po zavolání této funkce je otevřena adresa v nativní aplikaci na konkrétní adrese s popisem adresy.

```swift
public protocol CoordinatesProtocol {
    func getCoordinatesFromAddress(_ address: String,
 completion: @escaping (MKCoordinateRegion?) -> Void)
    func openAppleMaps(region: MKCoordinateRegion, name:
 String)
}

extension CoordinatesProtocol {
    func getCoordinatesFromAddress(_ address: String,
 completion: @escaping (MKCoordinateRegion?) -> Void) {
// ...
    }

    func openAppleMaps(region: MKCoordinateRegion, name:
 String) {
        // ...
    }
}
```

7.8 Detail zákazníka

Z detailu servisního úkolu se lze prokliknout na detail zákazníka. Tato obrazovka zobrazuje informace o konkrétním zákazníkovi, který je vázaný na servisní událost. Struktura, která zajišťuje zobrazení se jmenuje CustomerDetailView a vyžaduje při inicializaci objekt zákazníka typu Customer. Provázání View s ViewModelem pro detail zákazníka je

v podobném duchu jako u předchozích obrazovek a nebude tedy popisováno. CustomerDetailViewModel opět implementuje ObservableObject, PhoneProtocol a CoordinatesProtocol. Poslední dva zmíněné protokoly implementuje z důvodu toho, že obsahuje možnost zavolat na definované telefonní číslo a otevřít adresu zákazníka v mapě. Jedná se tedy o podobnou funkcionalitu jako v detailu servisního úkolu. Jako parametr této třídy je vyžadován model typu Customer, což reprezentuje konkrétního zákazníka.

```
final class CustomerDetailViewModel: ObservableObject, PhoneProtocol, CoordinatesProtocol {
    @Published var customer: Customer
    @Published var region: MKCoordinateRegion = MKCoordinateRegion()

    init(customer: Customer) {
        self.customer = customer
    }

    func callNumber() {
        callToNumber(customer.telephoneContact)
    }

    func getCoordinatesFromPlace() {
        getCoordinatesFromAddress(customer.address, completion: { (region) in
            guard let region else {
                return
            }
            self.region = region
        })
    }

    func openAppleMaps() {
        openAppleMaps(region: region, name: customer.name   )
    }
}
```

7.9 Hlášení stavu

Modální obrazovka hlášení stavu je dostupná z detailu servisního úkolu a je určena pro hlášení stavu servisní události. Tato struktura je pojmenovaná jako ServiceEventCompletionView a jako parametr vyžaduje id servisní události. Jednotlivé prvky jsou opět stejně provázány s ViewModelem stejně jako v předchozích případech. Implementace uživatelského rozhraní využívá SwiftUI strukturu Form, která zaobaluje a vytváří skupiny prvků. Jednotlivé části uživatelských vstupních prvků jsou zaobaleny do sekcí. Pomocí těchto komponent lze velmi pěkně vytvořit seznam pro uživatelské vstupy a nastavení. V ServiceEventCompletionView se nachází ve spodní části rozhraní tlačítko,

které slouží pro odeslání vyplněných dat. Tlačítko po stisku volá akci sendData, která se nachází ve třídě ViewModelu.

```
// ...
    Form {
        Section(header: Text("Popis")) {
            TextEditor(text: $viewModel.fullText)
        }
    }
// ...
    Button(action:
    {
         viewModel.sendData()
    }
    ) {
        Text("Odeslat")
        .frame(maxWidth: .infinity, maxHeight: 50)
    }
```

ViewModel lze vidět v další ukázce. V třídě se nachází funkce sendData, která simuluje odeslání dat vyplněných uživatelem do externího systému. Po simulovaném odeslání dat je modální obrazovka skryta pomocí změny hodnoty proměnné isCompletionViewPresenting. Tato proměnná typu Binding bool je předána při inicializaci třídy zároveň s objektem typu ServiceEventCompletion.

```
final class ServiceEventCompletionViewModel: ObservableObject {
    @Published var serviceEventCompletion: ServiceEventCompletion
    @Published var selectedImage: [UIImage] = []
    @Published var selectedItemCompletionType = ServiceEventCompletionTypeEnum.repaired
    @Published var fullText: String = "Poznámka..."
    @Published var isLoading = false
    var isCompletionViewPresenting: Binding<Bool>

    init(serviceEventCompletion: ServiceEventCompletion, isCompletionViewPresenting: Binding<Bool>) {

        self.serviceEventCompletion = serviceEventCompletion
        self.isCompletionViewPresenting = isCompletionViewPresenting
    }

    func sendData() {
        isLoading = true
        DispatchQueue.main.asyncAfter(deadline: .now() + 1.5)
        {
        self.isLoading = false
        self.isCompletionViewPresenting.wrappedValue.toggle()
        }
    }
}
```

7.10 Účet

Následující struktura je zodpovědná za zobrazení uživatelského účtu. Odlišností oproti ostatním strukturám, které slouží pro zobrazení dat je to, že zde není použit žádný ViewModel. Data přihlášeného uživatele jsou totiž získány z EnvironmentObject proměnné, která je sdílena z MainTabView. Ve spodní části uživatelské rozhraní se nachází tlačítko pro odhlášení. Odhlášení je zpracováno pomocí nastavení hodnoty proměnné isLogged na false, a jelikož je tato hodnota provázána s BaseView tak je poté zobrazena obrazovka pro přihlášení. Tato struktura vyžaduje jako parametr provázání pomocí Bindingu na proměnnou, která určuje, zda je uživatel přihlášen.

```
21  struct AccountView: View {
22      @EnvironmentObject var user: User
23      @Binding var isLogged: Bool
24
25      var body: some View {
26          VStack {
27              Image(systemName: "person.circle")
28                  .resizable()
29                  .frame(width: 80, height: 80).padding()
30              Text("\(user.firstName) \(user.lastName)")
31                  .font(.title)
32              TextWithBackground(title: "Přezdívka",
33                  text: user.username)
34              Spacer()
35              Button(action:
36                      {
37                        isLogged = false
38                      }
39              ) {
40                  Text("Odhlásit se")
41                      .frame(maxWidth: .infinity,
42                             maxHeight: 50)
43              }
44              .buttonStyle(ConfirmButtonStyle()).padding(-15)
45          }
46          .padding()
47          .preferredColorScheme(.dark)
48      }
49  }
50
```

Na tlačítku pro odhlášení je aplikován vlastní buttonStyle. Tento buttonStyle s názvem ConfirmButtonStyle nastavuje konkrétní vizuální styl a byl vytvořen z důvodu znuvupoužitelnosti na více místech v aplikaci. Výsledkem je to, že v případě použití tlačítka s určitým stylem není nutno tlačítko na každém místě, kde je využíváno jej stylovat ale stačí použití před definovaného stylu. Tento styl se používá ve vícero obrazovkách v aplikaci.

```
1   struct ConfirmButtonStyle: ButtonStyle {
2       func makeBody(configuration: Configuration)
3                                   -> some View {
4       configuration.label
5        .frame(maxWidth: .infinity, maxHeight: 50)
6        .background(.blue)
7        .foregroundColor(.white)
8        .cornerRadius(5)
9        .padding()
10  }
```

Výsledkem aplikace stylu na tlačítko je následující vizuální styl.

Obrázek 22 Tlačítko s vlastním aplikovaným stylem

8 IMPLEMENTACE APLIKACE V REDUX

Jako další možná architektura pro implementaci iOS aplikace byla zvolena architektura Redux. Tato architektura je na rozdíl od MVVM architektury velmi málo používaná pro mobilní aplikace určené pro platformu Apple. Zároveň pro tuto architekturu existuje v dnešní době malé množství zdrojů a ukázkových implementací na rozdíl od architektury MVVM. Oproti MVVM dělí kód do více částí a umožňuje zajímavé výhody. V architektuře Redux je vždy jasný stav aplikace. Implementace je inspirována na základě této ukázky implementace [28].

8.1 Store

Hlavní částí této architektury je Store. Třída je označena jako final, čímž je zaručeno, že žádná další třída nemůže z této třídy dědit. Zároveň Store implementuje ObservableObject. Třída obsahuje proměnnou state. V této proměnné se nachází aktuální stav aplikace. Proměnná je označena jako @Published, čím je zajištěna automatická aktualizace závislých prvků. Při inicializaci je nutno předat State, Reducer a pole Middleware. Třída obsahuje funkci dispatch s parametrem typu Action. V této funkci je synchronně volána funkce dispatchToNewState. Volání této funkce se provádí na frontě dispatchQueue. Každé volání funkce dispatchToNewState je voláno až po dokončení předchozího volání. Jedná se tedy o frontu, kdy jsou akce provedeny v pořadí, ve kterém byly přidány do fronty. Díky tomuto chování je zajištěna správná a konzistentní aktualizace stavu aplikace. Funkce dispatchToNewState nám zajišťuje přechod do nového stavu. Na řádku číslo 23 je získán nový stav z funkce reducer na základě aktuálního stavu a akce, která v aplikaci nastala. Poté je pomocí smyčky vytvořen průchod všech Middleware funkcí. Po vykonání funkce Middleware je zavolána funkce dispatch s novým stavem. V závěru funkce je nastaven nový stav aplikace.

```
final class Store<State>: ObservableObject {
    @Published private(set) var state: State

    private var subscriptions: [UUID: AnyCancellable] = [:]

    private let dispatchQueue = DispatchQueue(label:
"serviceMan", qos: .userInitiated)
    private let reducer: Reducer<State>
    private let middlewares: [Middleware<State>]

    init(initial: State, reducer: @escaping Reducer<State>,
middlewares: [Middleware<State>]) {
        self.reducer = reducer
        self.state = initial
        self.middlewares = middlewares
    }

    func dispatch(_ newAction: Action) {
        dispatchQueue.sync {
            dispatchToNewState(actualState: state, action:
newAction)
        }
    }

    private func dispatchToNewState(actualState: State,
action: Action) {
        let newState = reducer(actualState, action)

        for middleware in middlewares {
            let key = UUID()
            middleware(newState, action)
                .receive(on: RunLoop.main)
                .handleEvents(receiveCompletion: { [weak
self] _ in self?.subscriptions.removeValue(forKey: key) })
                .sink(receiveValue: dispatch)
                .store(in: &subscriptions, uuIDKey: key)
        }

        state = newState
    }
}

```

8.2 Implementace klíčových prvků

Tato architektura používá několik klíčových prvků, které jsou v této části popsány, jak jsou implementovány.

V této architektuře jsou využívány typy Reducer, Action a Middleware. Popis jejich významů se nachází v teoretické části práce.

Middlewares jsou definované jako výčtové typy. V implementaci je použit alias typu Middleware. Tento alias je funkcí, která přijímá argument State a Action. Jako návratová hodnota je AnyPublisher, která vydává akci typu Action a nevrací žádnou chybovou hodnotu. Action je definován jako protokol. Dále je v implementaci použit Reducer alias pro funkci, která přijímá State a Action a vrací State.

```
enum Middlewares {}

typealias Middleware<State> = (State, Action) ->
 AnyPublisher<Action, Never>

protocol Action {}

typealias Reducer<State> = (State, Action) -> State
```

Další důležitou částí je rozšíření třídy AnyCancellable o funkci store, která jako parametry přijímá slovník a klíč typu UUID. Tato funkce je využívána v třídě Store a funkci dispatchToNewState.

```
extension AnyCancellable {
    func store(in dictionary: inout [UUID: AnyCancellable],
               key: UUID) {
         dictionary[key] = self
    }
}
```

Další částí je struktura ActiveScreensState, která obsahuje konstantu screens, která obsahuje pole typu AppScreenState.

```
struct ActiveScreensState {
    let screens: [AppScreenState]
}
```

Další částí je výčtový typ AppScreenState, který definuje různé stavy obrazovek v aplikaci. Každý výčtový typ představuje obrazovku aplikace s konkrétním stavem obrazovky. Tento konkrétní stav reprezentuje potřebná data a informace pro danou obrazovku. Každá obrazovka je spojena se stavem obrazovky.

```
enum AppScreenState {
    case serviceEventList(ServiceEventListState)
    case serviceEvent(ServiceEventDetailState)
    case customer(CustomerDetailState)
    case serviceEventCompletion(ServiceEventCompletionState)
    case tabBar(TabBarState)
    case accountView(AccountViewState)
    case loginView(LoginViewState)
}
```

Dále je vytvořeno rozšíření, které umožňuje porovnat AppScreenState a ActiveAppScreen. Toto porovnání je využíváno v implementaci této architektury.

```
extension AppScreenState {
    static func == (lhs: AppScreenState, rhs:
 ActiveAppScreen) -> Bool {
        switch (lhs, rhs) {
        case (.loginView, .loginView):
            return true
            // ...
        }
    }

    static func == (lhs: ActiveAppScreen, rhs:
 AppScreenState) -> Bool {
        rhs == lhs
    }

    static func != (lhs: ActiveAppScreen, rhs:
 AppScreenState) -> Bool {
        !(lhs == rhs)
    }

    static func != (lhs: AppScreenState, rhs:
 ActiveAppScreen) -> Bool {
        !(lhs == rhs)
    }
}
```

Pro ActiveScreenState máme definované akce ve formě výčtového typu. ActiveScreensStateAction obsahuje dva typy, a to showScreen a dismissScreen. První případ reprezentuje akci, kdy je zobrazena nějaká obrazovka a přijímá hodnotu typu ActiveAppScreen, což je konkrétní obrazovka. Druhým případem je akce dismissScreen, která opět přijímá hodnotu typu ActiveAppScreen a říká, která obrazovka byla zavřena. Výčtový typ implementuje protokol Action. Obecně řečeno, ActiveScreensStateAction poskytuje možnosti pro úpravu stavu obrazovek v aplikaci.

```
enum ActiveScreensStateAction: Action {
    case showScreen(ActiveAppScreen)
    case dismissScreen(ActiveAppScreen)
}
```

Další částí je ActiveScreenStateReducer. Reducer je implementován jako statická proměnná, která rozšiřuje ActiveScreenState. Tato proměnná slouží jako funkce, která vrací nový stav na základě akce a předchozího stavu. Jak lze vidět v následující implementaci, tento Reducer vrací ActiveScreensState s polem aktivních obrazovek. V případě zavolání akce showScreen je obrazovka přidána a v případě akce dismissScreen je obrazovka odebrána. Následně je zavolán pro každou obrazovku Reducer, kdy je provedena aktualizace stavů jednotlivých obrazovek. Funkce přesně pracuje na základě akce a stavu.

```
extension ActiveScreensState {
    static let reducer: Reducer<Self> = { state, action in
        var screens = state.screens

        if let action = action as? ActiveScreensStateAction {
            switch action {
            case .showScreen(.loginView):
                screens = [.loginView(LoginViewState())]
            case .showScreen(.tabBar):
                screens = [.tabBar(TabBarState())]
            case .showScreen(.accountView):
                screens += [.accountView(AccountViewState())]
            case .showScreen(.serviceEventList):
                screens +=
[.serviceEventList(ServiceEventListState())]
            case .showScreen(.serviceEvent(let id)):
                screens +=
[.serviceEvent(ServiceEventDetailState(id: id))]
            case .showScreen(.customer(let id)):
                screens += [.customer(CustomerDetailState(id:
id))]
            case .showScreen(.serviceEventCompletion(let
id)):
                screens +=
[.serviceEventCompletion(ServiceEventCompletionState(id:
id))]
            case .dismissScreen(let screen):
                screens = screens.filter({$0 != screen})
            }
        }

        screens = screens.map { AppScreenState.reducer($0,
action) }

        return ActiveScreensState(screens: screens)
    }
}
```

Dále se v implementaci vyskytuje AppScreenStateReducer, který je volán v předchozí ukázce. Tento Reducer stejně jako v případě předchozí ukázky přijímá aktuální stav a akci. Následně na základě stavu je zavolán Reducer, který odpovídá danému stavu neboli dané obrazovce. Výsledek Reduceru je vrácen jako AppScreenState. Obecně tento Reducer se stará o volání dalších Reducerů jednotlivých obrazovek, které aktualizují jejích stav na základě akce. Tímto je zaručena aktualizace a koordinace stavu napříč aplikací.

```
extension AppScreenState {
    static let reducer: Reducer<Self> = { state, action in
        switch state {
        case .serviceEventList(let state):
            return .serviceEventList(
            ServiceEventListState.reducer(state, action))
        case .serviceEvent(let state):
            return .serviceEvent(
            ServiceEventDetailState.reducer(state, action))
        case .customer(let state):
            return .customer(
            CustomerDetailState.reducer(state, action))
        case .serviceEventCompletion(let state):
            return .serviceEventCompletion(
            ServiceEventCompletionState.reducer(state,
            action))
        case .tabBar(let state):
            return .tabBar(
            TabBarState.reducer(state, action))
        case .accountView(let state):
            return .accountView(
            AccountViewState.reducer(state, action))
        case .loginView(let state):
            return .loginView(
            LoginViewState.reducer(state, action))
        }
    }
}
```

Další důležitou částí je rozšíření AppState o funkci, která nám na základě hodnoty typu ActiveAppScreen vrátí State. Toto je využíváno téměř v každé obrazovce, kdy nám je navrácen aktuální State obrazovky. Obecně řekneme funkci, že chceme získat stav pro obrazovku loginView a funkce nám vrátí stav LoginViewState.

```
extension AppState {
    func screenState<State>(for screen: ActiveAppScreen) ->
 State? {
        return activeScreens.screens
            .compactMap {
            switch ($0, screen) {
            case (.serviceEventList(let state),
                     .serviceEventList):
                return state as? State
            case (.serviceEvent(let state),
                 .serviceEvent(let id)) where state.id == id:
                return state as? State
            case (.customer(let state),
                 .customer(let id)) where state.id == id:
                return state as? State
            case (.serviceEventCompletion(let state),
                 .serviceEventCompletion(let id))
                 where state.id == id:
                return state as? State
            case (.tabBar(let state), .tabBar):
                return state as? State
            case (.accountView(let state), .accountView):
                return state as? State
            case (.loginView(let state), .loginView):
                return state as? State
        default:
                return nil
        }
    }.first
  }
}
```

Nyní se lze přesunout na konkrétní popis implementace, jelikož klíčové prvky implementace architektury, které jsou využívány byly popsány. Modely v architektuře MVVM jsou identické s architekturou Redux a nejsou tedy již znova popisovány.

8.3 Vstupní bod aplikace

Na začátku zdrojového kódu je inicializován globální store, který je v této architektuře hlavním prvkem. Při inicializaci má nastaven počáteční stav, Reducer a jsou zaregistrovány jednotlivé Middleware. Dále se vytvářejí instance Coordinates a Phone Middleware. Tyto Middlewares jsou popsány později. Hlavním vstupním bodem aplikace je struktura serviceMan_reduxApp. Tato struktura v proměnné body vrací BaseView a zároveň do této struktury nastavuje jako environmentObject store. Tím je zaručené sdílení store v aplikaci.

```
let store = Store(
    initial: AppState(),
    reducer: AppState.reducer,
    middlewares: [Middlewares.eventList, Middlewares.logger, Middlewares.user]
)

let coordinatesMiddleware = CoordinatesMiddleware()
let phoneMiddleware = PhoneMiddleware()

@main
struct serviceMan_reduxApp: App {
    var body: some Scene {
        WindowGroup {
            BaseView()
                .environmentObject(store)
        }
    }
}

struct BaseView: View {
    @EnvironmentObject var store: Store<AppState>
    @State var isLoggedIn = true

    var body: some View {
        Group {
            if (store.state.screenState(for: .loginView) as LoginViewState?)?.loggedUser != nil || store.state.screenState(for: .tabBar) as TabBarState? != nil {
             MainTabView()
             .environmentObject((store.state.screenState(for: .loginView) as LoginViewState?)?.loggedUser ??
                   User.sampleData)
                    .onAppear( perform: {
store.dispatch(ActiveScreensStateAction.showScreen(.tabBar))
                    })
            } else {
                LoginView()
            }
        }
    }
}

```

V struktuře BaseView se na základě stavu ve store vrací struktura MainTabView nebo LoginView. První zmíněná struktura je zobrazena přihlášenému uživateli a druhá slouží pro nepřihlášeného uživatele. Zároveň je předán do MainTabView přihlášený uživatel jako environmentObject. Při zobrazení MainTabView je zavolána dispatch funkce, která se nachází ve Store, která oznamuje akci, že byla zobrazena obrazovka tabBar.

8.4 Přihlašovací obrazovka

Každá obrazovka má v Redux architektuře svůj vlastní stav. Stav je většinou reprezentován jako struktura. Ukázku stavu lze vidět v následujícím kódu. Stav obsahuje proměnnou typu UserCredentials, sloužící pro uložení přihlašovacích dat a následné odeslání, proměnnou zda probíhá načítání a proměnnou typu User, která obsahuje přihlášeného uživatele

```
struct LoginViewState {
    var credentials: UserCredentials
    var isLoading: Bool
    var loggedUser: User?
}

extension LoginViewState {
    init() {
        isLoading = false
        credentials = UserCredentials(name: "", password: "")
        loggedUser = nil
    }
}
```

Pro přihlašovací obrazovku jsou definovány následující akce, které zde mohou nastat. Jedná se například o aktualizaci přihlašovacích údajů, které uživatel zadá nebo o zaslání požadavku na přihlášení do externího systému. V případě hodnoty updateNameText(String) tato akce reprezentuje změnu v textovém poli pro přihlašovací jméno a přijímá hodnotu typu String, která reprezentuje daný textový řetězec. Implementačně se jedná opět o výčtový typ, který implementuje protokol Action.

```
enum LoginViewStateAction: Action {
    case postLogin(UserCredentials)
    case updateNameText(String)
    case updatePasswordText(String)
    case didLogin(User)
}
```

Další částí je LoginViewStateReducer, který zpracovává akce typu LoginViewStateAction. Jak lze vidět například při stavu postLogin, tak je navrácen nový stav a proměnná isLoading je nastavena na true. Tím je změněn stav na základě příchozí akce. Další možnou ukázkou je akce updateNameText, kdy je do stavu zapsána hodnota, kterou uživatel zadal to uživatelského vstupu. Pro každou akci je zde vytvořen nový stav.

```
extension LoginViewState {
    static let reducer: Reducer<Self> = { state, action in
        guard let action = action as? LoginViewStateAction
            else {
             return state
            }
        switch action {
        case .postLogin(let crendetials):
            return LoginViewState(credentials: crendetials,
                      isLoading: true)
        case .updateNameText(let text):
            let credentials = UserCredentials(name: text,
                      password: state.credentials.password)
            return LoginViewState(credentials: credentials,
                      isLoading: false)
        case .updatePasswordText(let text):
            let credentials = UserCredentials(name:
                 state.credentials.name, password: text)
            return LoginViewState(credentials: credentials,
                    isLoading: false)
        case .didLogin(let user):
            return LoginViewState(credentials:
                    state.credentials,
                    isLoading: false, loggedUser: user)
        }
    }
}
```

Dále je potřeba implementovat Middleware, který nám bude zajišťovat samotnou logiku pro přihlášení. Middleware na základě akce provede akci a vrátí novou akci. Middleware v případě příchozí akce postLogin zavolá akci logIn, která se nachází ve třídě UserRepository a vrátí novou akci didLogin s hodnotou přihlášeného uživatele. Ze. Stavu postLogin je tedy změněn stav aplikace na didLogin.

```
extension Middlewares {
    private static let userRepository = UserRepository()

    static let user: Middleware<AppState> = { state,
                                        action in
    switch action {
    case LoginViewStateAction.postLogin(let credentials):
        return userRepository
                .logIn(credentials: credentials)
                .map { LoginViewStateAction.didLogin($0) }
                .ignoreError()
                .eraseToAnyPublisher()
    default:
    return Empty()
            .eraseToAnyPublisher()
    }
  }
}
```

Třída UserRepository je určena k získání přihlášeného uživatele. Funkce simuluje dotaz na externí systém a vrací uživatele po 1 vteřině.

```
final class UserRepository: ObservableObject {
    func logIn(credentials: UserCredentials) ->
                AnyPublisher<User, RepositoryError> {
            return Just(User.sampleData)
                    .delay(for: 1.0, scheduler: RunLoop.main)
                    .setFailureType(to: RepositoryError.self
                    .eraseToAnyPublisher()
    }
}
```

V případě chyby je vrácena hodnota z výčtového typu RepositoryError.

```
enum RepositoryError: Error {
    case unknown
    case notFound
}
```

Po implementaci těchto částí lze implementovat samotné uživatelské rozhraní. Na začátku struktury je deklarována proměnná store jako EnvirnemntObject, tím je získána instance store. Následně obsahuje struktura počítanou proměnnou state typu LoginViewState. Ta nám získává aktuální stav této obrazovky ze store. Následně uživatelské rozhraní pracuje s tímto stavem. Jak lze vidět na řádku číslo 7, tak je prvně zjištěno pomocí optional binding, zda hodnota state obsahuje hodnotu. Poté je na základě hodnoty ze stavu zobrazen indikátor.

Uživatelské prvky jako je například TextField jsou následně propojeny pomocí Binding se stavem obrazovky. Při uživatelském zápisu do TextField je informován store pomocí funkce dispatch, kdy je zavolán s parametrem akce updateNameText a textovým řetězcem. Tím je store notifikován o změně stavu na této obrazovce. Data, jako je uživatelem zapsané jméno je získáno ze stavu z proměnné credentials. Tímto způsobem komunikují prvky, které se starají o uživatelské rozhraní se store.

```
struct LoginView: View {
    @EnvironmentObject var store: Store<AppState>
    var state: LoginViewState? {
            store.state.screenState(for: .loginView)
         }
    var body: some View {
        if let state {
            if state.isLoading {
// ...
        }
// ...
        TextField(
            "Jméno",
            text: Binding(get: { state.credentials.name },
                          set: {
                            store.dispatch(
                        LoginViewStateAction.updateNameText($0)
            )})
        )
    }
// ...
}
```

Jakmile je zobrazeno LoginView, tak je store notifikován o zobrazení této obrazovky pomocí funkce vázané na onAppear. Při schování této obrazovky je zase store notifikován o skrytí. Každé zobrazení a skrytí obrazovky musí notifikovat store.

```
.onAppear(perform: {
            store.dispatch(
            ActiveScreensStateAction.showScreen(.loginView))
        })
.onDisappear(perform: {
         store.dispatch(
         ActiveScreensStateAction.dismissScreen(.loginView))
        })
```

Tato zmíněná implementace obrazovky pro přihlášení ukazuje kompletní proces toho, jak Redux architektura funguje a co vše je potřeba implementovat.

8.5 TabView

Implementace TabView je téměř identická s MVVM implementací. Jediným rozdílem je absence bindované proměnné isLogged, která v MVVM sloužila pro odhlášení uživatele.

8.6 Seznam servisních úkolů

Pro seznam servisních úkolů je definován stav, který obsahuje seznam všech servisních událostí a informaci, zda probíhá načítání.

```
struct ServiceEventListState {
    let serviceEvents: [ServiceEvent]
    let isLoading: Bool
}
```

Dále jsou definovány akce pro získání servisních události a akce, která reprezentuje získání servisních událostí.

```
enum ServiceEventListStateAction: Action {
    case fetchEvents
    case didReceiveEvents([ServiceEvent])
}
```

ServiceEventListStateReducer obsahuje implementaci dvou stavů, které zde mohou nastat a navrací nový stav. V případě příchozí akce fetchEvents je do stavu a proměnné serviceEvents nastaveno prázdné pole. V případu druhé akce jsou do stavu uloženy hodnoty příchozích servisních událostí. Pro získání těchto dat existuje ServiceEventListMiddleware.

```
extension ServiceEventListState {
    static let reducer: Reducer<Self> = { state, action in
        switch action {
        case ServiceEventListStateAction.fetchEvents:
            return ServiceEventListState(
                serviceEvents: [],
                isLoading: true
                )
         case ServiceEventListStateAction
                    .didReceiveEvents(let events):
            return ServiceEventListState(
                serviceEvents: events,
                isLoading: false
                )
         default:
                  return state
        }
     }
}
```

```
extension Middlewares {
    private static let serviceEventListRepository = ServiceEventListRepository()
    private static let serviceEventDetailRepository = ServiceEventDetailRepository()
    private static let customerDetailRepository = CustomerDetailRepository()

    static let eventList: Middleware<AppState> = { state, action in
        switch action {
        case ServiceEventListStateAction.fetchEvents:
        return serviceEventListRepository
            .fetchEvents()
            .map { ServiceEventListStateAction.didReceiveEvents($0) }
            .ignoreError()
            .eraseToAnyPublisher()
        case ServiceEventDetailAction.getServiceEvent(let id):
        return serviceEventDetailRepository
            .fetchEvent(id: id)
            .map { ServiceEventDetailAction.didReceiveEvent($0) }
            .ignoreError()
            .eraseToAnyPublisher()
        case CustomerDetailAction.getCustomer(let id):
        return customerDetailRepository
            .fetchCustomer(id: id)
            .map { CustomerDetailAction.didReceiveCustomer($0) }
            .ignoreError()
            .eraseToAnyPublisher()
        case ServiceEventCompletionAction.postCompletion(let completion):
        return serviceEventDetailRepository
                .postCompletion(completion)
                .map { _ in ServiceEventCompletionAction.didFinishCompletion(true) }
                .ignoreError()
                .eraseToAnyPublisher()
        default:
            return Empty().eraseToAnyPublisher()
        }
    }
}
```

Na začátku kódu jsou vytvořeny instance repozitářů pro seznam událostí, detail servisní události a pro detail zákazníka. Tento Middleware slouží pro seznam událostí, detail události a pro detail zákazníka. Na základě příchozí akce jsou volány další funkce v repozitářích.

V případě seznamu servisních událostí a příchozí akce fetchEvents je zavolána funkce fetchEvents, která se nachází ve třídě ServiceEventListRepository. Následně při získání dat je vrácen stav didReceiveEvents a jako parametr je pole servisních událostí.

Následně je store propojen s uživatelským rozhraním. Stejně jako v případě všech obrazovek v architektuře Redux je deklarována proměnná store a vytvořena počítaná proměnná state, která vrací stav ze store typu ServiceEventListState. Následně jsou na řádku číslo 14 pomocí List komponenty vykresleny servisní události na základě dat ze stavu a proměnné serviceEvents. Poté je vytvořena navigace na detail servisní události.

```
1  struct ServiceEventList: View {
2      @EnvironmentObject var store: Store<AppState>
3      var state: ServiceEventListState? {
4          store.state.screenState(for: .serviceEventList)
5      }
6
7      var body: some View {
8          NavigationView {
9              Group {
10                 if let state {
11                     if state.isLoading {
12                         ProgressView("Načítání")
13                     } else {
14                 List(state.serviceEvents) { serviceEvent in
15 NavigationLink(destination: ServiceEventDetailView(eventId:
    serviceEvent.id)){
16                 ServiceEventCellView(event: serviceEvent)
17                 }
18             }
19             .listStyle(PlainListStyle())
20             }
21         }
22     }
23     .navigationBarTitle(Text("Servisní události"))
24   }
25   .onAppear(perform: {
26    store.dispatch(ActiveScreensStateAction
27    .showScreen(.serviceEventList))
28    store.dispatch(ServiceEventListStateAction.fetchEvents)
29  })
30  .onDisappear(perform: {
31    store.dispatch(ActiveScreensStateAction
32    .dismissScreen(.serviceEventList))
33  })
34  .environment(\.colorScheme, .dark)
35  }
36 }
```

Za zmínku v tomto kódu stojí ještě metoda onAppear, kdy se v této funkci po zobrazení notifikuje store o zobrazení této obrazovky a zároveň je volána akce fetchEvents. Tím je

zajištěno načtení dat při zobrazení. Při skrytí obrazovky je opět zavolána akce dismissScreen. Store je hlavním prvkem této architektury, a proto musí být o všem informován a data by měla pocházet ze stavu který je získán ze store.

8.7 Detail servisního úkolu

Stav detailu servisního úkolu obsahuje id a objekt servisní události.

```
struct ServiceEventDetailState {
    let id: String
    let serviceEvent: ServiceEvent?
}
```

Pro tuto obrazovku je implementováno několik akcí. Kromě akce pro získání a informace o získání dat obsahuje tento výčtový typ i akce pro otevření nativní aplikace Mapy a zavolání na číslo.

```
enum ServiceEventDetailAction: Action {
    case getServiceEvent(id: String)
    case didReceiveEvent(ServiceEvent)
    case openAppleMaps(ServiceEvent)
    case callToNumber(ServiceEvent)
}
```

Samozřejmostí je implementace ServiceEventDetailStateReducer. Zde stojí za zmínku akce openAppleMaps a callToNumber. V těchto dvou případech se stavem aplikace nic neděje a je vrácen stav ve stejném stavu.

```
extension ServiceEventDetailState {
    static let reducer: Reducer<Self> = { state, action in
        guard let action = action as?
                            ServiceEventDetailAction
         else { return state }
        switch action {
        case .getServiceEvent(let id):
            return ServiceEventDetailState(id: id,
                    serviceEvent: nil)
        case .didReceiveEvent(let event):
            return ServiceEventDetailState(id: event.id,
                    serviceEvent: event)
        case .openAppleMaps(let event):
            return ServiceEventDetailState(id: event.id,
                    serviceEvent: event)
        case .callToNumber(let event):
            return ServiceEventDetailState(id: event.id,
                    serviceEvent: event)
        }
    }
}
```

Následně je v případě akce fetchEvent volána funkce fetchEvent třídy ServiceEventDetailRepository skrz Middleware a tím jsou získána data pro detail.

```
class ServiceEventDetailRepository: ObservableObject {

    func fetchEvent(id: String) ->
AnyPublisher<ServiceEvent, RepositoryError> {
        let events = ServiceEvent.sampleData.first(where:
{$0.id == id }) ?? ServiceEvent.sampleData[1]

        return Just(events)
            .setFailureType(to: RepositoryError.self)
            .eraseToAnyPublisher()
    }

    func postCompletion(_ completion:
ServiceEventCompletion) -> AnyPublisher<Bool,
RepositoryError> {
        return Just(true)
            .delay(for: 1.0, scheduler: RunLoop.main)
            .setFailureType(to: RepositoryError.self)
            .eraseToAnyPublisher()
    }
}
```

Zároveň se v této třídě nachází funkce postCompletion, která bude zmíněna posléze a je využívána při hlášení stavu události.

V podobném formátu jako u předchozích obrazovek je implementováno propojení uživatelského rozhraní se store. Následně budou popsány pouze specifické části implementace této obrazovky.

V případě tlačítka pro volání čísla je po stisku tlačítka zaslána akce callToNumber do store a samotná logika volání je vyřešena pomocí zavolání funkce nad proměnnou phoneMiddleware, která byla již zmíněna na počátku popisu implementace.

```
ButtonInsideButtonView(title:
 serviceEvent.customer.contactName,
 iconString: "phone.fill",
 function: {
    store.dispatch(ServiceEventDetailAction
    .callToNumber(serviceEvent))
    phoneMiddleware.callToNumber(
    serviceEvent.telephoneContact)
    }
)
```

PhoneMiddleware je implementován v následujícím kódu.

```
final class PhoneMiddleware {
    func callToNumber(_ number: String) {
        guard let url = URL(string: "tel://\(number)"),
              UIApplication.shared.canOpenURL(url) else {
            return
        }
        UIApplication.shared.open(url, options: [:],
 completionHandler: nil)
    }
}
```

Stejným principem je i řešeno zobrazení aplikace Mapy po stisku uživatelského rozhraní s mapou. Opět je vyslána akce do store a zavolána funkce na proměnné coordinatesMiddleware.

Při stisku tlačítka pro hlášení stavu je zaslána následující akce do store.

```
Button(action: {
    store.dispatch(ActiveScreensStateAction
        .showScreen(.serviceEventCompletion(id:
 serviceEvent.id)))
 }) {
  Text("Hlášení stavu")
    .frame(maxWidth: .infinity,
     maxHeight: 50)
  }
```

V této struktuře je také vytvořena počítaná proměnná, která vrací stav aplikace pro ServiceEventCompletionState.

```
var stateCommmpletion: ServiceEventCompletionState? {
 store.state.screenState(for: .serviceEventCompletion(id:
eventId))
}
```

Kombinací tohoto chování je velmi zajímavě vyřešeno zobrazovaní modálního okna pro hlášení stavu.

```
.sheet(isPresented: Binding(get: {
    stateCommmpletion != nil
}, set: {
    _ in
})) {
//            ...
}
```

Jak lze vidět v kódu, modální obrazovka je zobrazena pouze v případě, kdy proměnná stateCompletion obsahuje hodnotu. Tato situace nastává, pokud je pomocí tlačítka pro hlášení stavu zaslána akce showScreen s hodnotou serviceEventCompletion. Výsledkem toho je, že proměnná obsahuje hodnotu a je tedy modální okno prezentováno a vše je řízeno stavem store.

Samozřejmostí ServiceEventDetailView je to, že při zobrazení je zaslána akce getServiceEvent do store. Tím je skrz ServiceEventListMiddleware a následně ServiceEventDetailRepository získána konkrétní servisní událost.

8.8 Detail zákazníka

Podobně jako u předchozí obrazovky se ve stavu detailu zákazníka nachází id a objekt zákazníka.

```
struct CustomerDetailState {
    let id: String
    let customer: Customer?
}
```

Mezi evidované akce na této obrazovce patří získání zákazníka, informace o proběhlém získání a otevření nativní aplikace pro mapy.

```
enum CustomerDetailAction: Action {
    case getCustomer(id: String)
    case didReceiveCustomer(Customer)
    case openAppleMaps(Customer)
}
```

Samozřejmostí je implementace CustomerDetailStateReducer, který na základě akce vrací nový stav. Akce didRecieveCustomer uloží do stavu konkrétního načteného zákazníka.

```
extension CustomerDetailState {
    static let reducer: Reducer<Self> = { state, action in
        guard let action = action as? CustomerDetailAction
           else { return state }
        switch action {
        case .getCustomer(let id):
            return CustomerDetailState(id: id,
                    customer: nil)
        case .didReceiveCustomer(let customer):
        return CustomerDetailState(id: customer.id,
                    customer: customer)
        case .openAppleMaps(let customer):
        return CustomerDetailState(id: customer.id,
                customer: customer)
        }
    }
}
```

Za zmínku zde stojí způsob získání konkrétního zákazníka. Při vytváření instance CustomerDetailView je vyžadován parametr customerId. Na základě tohoto id je pak následně získán konkrétní zákazník ze store.

```
struct CustomerDetailView: View {
    @EnvironmentObject var store: Store<AppState>
    var state: CustomerDetailState? {
 store.state.screenState(for: .customer(id: customerId)) }
    let customerId: String
//    ...
}
```

8.9 Hlášení stavu

State je pro hlášení stavu opět v podobném stylu jako u předchozích obrazovek. Je zde evidováno id, objekt typu ServiceEventCompletion, který slouží pro odeslání do externího systému a hodnota zda probíhá načítání.

```
struct ServiceEventCompletionState {
    let id: String
    var completion: ServiceEventCompletion?
    var isLoading: Bool
}
```

V souvislosti s tím, že uživatel zde vyplňuje vícero vstupních dat, tak je nutno zde evidovat více akcí. Jedná se o akci odeslání do externího systému, aktualizaci vstupních dat na základě interakce uživatele a ukončení odeslání do externího systému.

```
enum ServiceEventCompletionAction: Action {
    case postCompletion(ServiceEventCompletion)
    case updateDescriptionText(String)
    case updateIsCompleted(Bool)
    case updateCompletionType(ServiceEventCompletionTypeEnum)
    case didSelectImage(UIImage)
    case didFinishCompletion(Bool)
}
```

Reducer je implementován opět ve stejném stylu. V ukázce je zobrazen Reducer pro první dvě akce. První akce vrací stav na základě postCompletion a druhá nastává při akci updateDescriptionText. V druhém případě lze vidět, jak je zapisován nový stav s novou hodnotou z uživatelského rozhraní do parametru note.

```
extension ServiceEventCompletionState {
    static let reducer: Reducer<Self> = { state, action in
        guard let action = action as? ServiceEventCompletionAction else { return state }
        switch action {
        case .postCompletion(let completion):
        return ServiceEventCompletionState(id: state.id, completion: completion, isLoading: true)
        case .updateDescriptionText(let text):
        var completion: ServiceEventCompletion?
        if let stateCompletion = state.completion {
        completion = ServiceEventCompletion(id: stateCompletion.id, serviceEventId: stateCompletion.serviceEventId, isCompleted: stateCompletion.isCompleted, note: text, selectedItemCompletionType: stateCompletion.selectedItemCompletionType, selectedImage: stateCompletion.selectedImage)
        }
        return ServiceEventCompletionState(id: state.id,
                completion: completion, isLoading: false)
    }
}
```

Za zmínku v případě popisu provázání uživatelského rozhraní reprezentováno strukturou ServiceEventCompletionView stojí provázání vstupu se store, které je řešeno pomocí Binding. Stejný styl provázání byl také v přihlašovací obrazovce.

```
TextEditor(text: Binding(get: {
    serviceEventCompletion.note
}, set: {
    store.dispatch(ServiceEventCompletionAction
            .updateDescriptionText($0))
}))
```

Simulované odeslání do externího systému po stisku tlačítka na odeslání je řešeno jako odeslání akce postCompletion s parametrem typu ServiceEventCompletion, což reprezentuje uživatelem vyplněné údaje. Následně je akce skrz ServiceEventListMiddleware odeslána do ServiceEventDetailRepository, kde se nachází akce pro odeslání dat do externího systému.

Zbylá funkcionalita v této obrazovce je v podobném stylu jako u předchozích obrazovek a není tedy již popisována.

8.10 Účet

Poslední obrazovkou, jejíž implementace nebyla doposud popsána je účet. Data o přihlášeném uživateli jsou zobrazovány stejně jako u architektury MVVM pomocí EnvironmentObject. Velmi elegantním řešením je vyřešeno odhlášení. Jak již bylo popsáno u vstupního bodu aplikace, tak pro odhlášení a zobrazení přihlašovací obrazovky stačí pouze nastavit stav do store. V akci pro tlačítko odhlášení je tedy zavolána funkci dispatch s akcí showScreen(.loginView) a díky tomu je zobrazena obrazovka pro přihlášení. Oproti architektuře MVVM, kde toto bylo řešeno pomocí Binding proměnné, která se musela předávat velmi příjemné řešení.

```
Button(action:
{
   store.dispatch(ActiveScreensStateAction
     .showScreen(.loginView))
}
) {
   Text("Odhlásit se")
   .frame(maxWidth: .infinity, maxHeight: 50)
}
```

9 SROVNÁNÍ IMPLEMENTACÍ

Tato kapitola popisuje rozdílnost implementací architektury MVVM a Redux.

9.1 Obecná náročnost implementace

Obecně lze říct, že pro každou obrazovku v architektuře MVVM byl vytvořen ViewModel. Správné rozdělení kódu do daných částí architektury není příliš náročné. Většina logiky se nachází právě ve ViewModelu. Tím je zaručeno jednoduché pravidlo, kdy je většina kódu související s logikou v této třídě. Architektura umožňuje velmi rychlou adaptaci vývojáře na tuto architekturu, jelikož se jedná o jednu ze spíše jednodušších architektur. K implementaci se tedy může rychle zapojit i juniorní vývojář.

Naopak v případě Redux architektury, tak je pro každou obrazovku vyžadováno několik podpůrných funkcionalit. Pro každou obrazovku je nutno vytvořit minimálně stav, akce a reducer. Zde v počátku implementace jsem měl velmi problém s rozdělením, kdy, kde a co implementovat. V případě MVVM nám vystačí jeden ViewModel a zde je nutno mít vícero souborů a tím je nutno aby vývojář více přemýšlel. Díky tomu může být zapojení vývojáře do této architektury náročnější. Z osobního pohledu lze říct, že Redux architektura je náročnější na implementaci. V případě této architektury považuji za velkou nevýhodu to, že vývojář může zapomenou implementovat určitou část ale projekt lze překladačem přeložit. Během implementace Redux architektury nastávaly situace, kdy aplikace byla úspěšně přeložena ale nefungovala, jak bylo očekáváno. Problém nastával například v rozšíření AppState o funkci screenState. Mezi další nevýhodu patří nutná registrace vytvořeného Middleware do Store. Vývojář může opomenout tuto nutnost a následně Middleware nebude volán.

MVVM je tedy snazší na pochopení a implementaci.

9.2 Počet položek

V architektuře MVVM pro implementaci bylo vytvořeno 58 položek. Pro architekturu Redux bylo potřeba 110 položek. Velikostně tedy architektura Redux vyžaduje více položek. Tím je architektura Redux pracnější, jelikož vývojář musí vytvořit mnohem více souborů. Během implementace aplikace v architektuře Redux velmi často nastával problém s indexováním dat v uživatelském prostředí Xcode. Často se objevovaly upozornění a chyby, které posléze zmizely. Toto není způsobeno přímo architekturou Redux ale kombinací Xcode a většího množství položek v projektu. Jedná se spíše o nedokonalost Xcode, která se

zde projevila. Velký počet položek lze vysvětlit nutností stavu, akce a reduceru pro každou obrazovku, což obecně v MVVM stačí pro každou obrazovku pouze ViewModel.

Za zmínku zde stojí to, že i pro relativně menší aplikaci zde bylo nutno vytvořit tolik souborů, což by v případě větší aplikace mohlo být spíše nepříjemné.

9.3 Počet řádků

V Redux implementaci bylo napsáno 2112 řádků. V MVVM architektuře to bylo 1313. Tento výsledek opět poukazuje na větší pracnost architektury Redux. Počet řádků v architektuře Redux lze i vysvětlit tím, že je nutno Store a všem notifikovat. Příkladem je zobrazení a skrytí obrazovky, což je nutno mít implementováno pro každou obrazovku a už jenom těmito akcemi je počet řádku zvyšován, což v architektuře MVVM není potřeba.

9.4 Testovatelnost

Každá architektura má rozdílnou testovatelnost. V této kapitole jsou ukázány rozdíly v testování.

9.4.1 UI Testy

Pro demonstraci UI testování byla v demo aplikaci otestována obrazovka pro přihlášení. Kód lze vidět v následující ukázce. Po spuštění aplikace je otestováno, zda je v uživatelském rozhraní vykreslen text Jméno a Heslo. Následně se testuje, zda existuje TextField s názvem Jméno a Heslo. Do obou z těchto vstupních prvků je vložen text. Poté je ověřena existence tlačítka pro přihlášení a je toto tlačítko stisknuto.

```swift
let app = XCUIApplication()
app.launch()

XCTAssertTrue(app.staticTexts["Jméno"].exists)
XCTAssertTrue(app.staticTexts["Heslo"].exists)

let nameTextField = app.textFields["Jméno"]
XCTAssertTrue(nameTextField.exists)

nameTextField.tap()
nameTextField.typeText("pavelodstrcilik")

let passwordSecureTextField = app.secureTextFields["Heslo"]
XCTAssertTrue(passwordSecureTextField.exists)
passwordSecureTextField.tap()
passwordSecureTextField.typeText("123456789")

let logInButton = app.buttons["Přihlásit se"]
XCTAssertTrue(logInButton.exists)
logInButton.tap()
```

Závislost tohoto kódu není na architektuře. Tím je tedy možné tento testovací kód použít jak v architektuře MVVM, tak i v Redux.

9.4.2 Unit testy

Odlišným příkladem jsou unit testy. Pro demonstraci testování kódu pomocí unit testů byla opět zvolena přihlašovací obrazovka.

V architektuře MVVM byl vytvořen následující test.

```swift
let isLoading = false
let isLoggedIn = false
let firstName = ""
let lastName = ""
let username = ""

let viewModel = LoginViewModel()

XCTAssertFalse(isLoading)
XCTAssertFalse(isLoggedIn)
XCTAssertEqual(viewModel.loggedUser.firstName, firstName)
XCTAssertEqual(viewModel.loggedUser.lastName, lastName)
XCTAssertEqual(viewModel.loggedUser.username, username)

viewModel.logIn()
XCTAssertTrue(viewModel.isLoading)
```

Tento unit test vytvoří instanci třídy LoginViewModel a následně testuje, zda proměnné této třídy mají očekávanou hodnotu. Je například otestováno, že není aktivní načítání nebo zda není přihlášen uživatel. Poté je zavolána funkce logIn a je otestováno, zda je aktivní načítání.

V případě architektury Redux lze testovat jednotlivé akce reduceru. V ukázce je testován LoginViewReducer. Na začátku testu je vytvořen počáteční stav. Poté jsou vytvořeny proměnné s přihlašovacími údaji a je vytvořena postLogin akce. Tato vytvořená akce je spolu s počátečním stavem poslána do LoginViewReducer a je vrácen nový stav. Následně je otestováno, že se přihlašovací údaje správně uložily a zda probíhá načítání. Tím je ověřena funkčnost pro tuto akci. V další ukázce unit testování je otestována akce didLogin. Prvně je vytvořena proměnná s hodnotou demo uživatele. Následně je vytvořena akce didLogin s demo uživatelem jako parametrem. Poté je získán nový stav z LoginViewReducer na základě původní hodnoty a didLogin akce. Následně je otestováno, zda se přihlášený uživatel ve vráceném stavu rovná s uživatelem, který byl do akce didLogin zaslán. Ve výsledku je tedy testováno zaslání akce postLogin a akce, kdy je získán přihlášený uživatel.

```
let initialState =
 LoginViewState(credentials: UserCredentials(name: "",
                                             password: ""),
                isLoading: false)

let credentailName = "pavel"
let credentailPassword = "password"
let postLoginAction =
LoginViewStateAction.postLogin(
                UserCredentials(name: credentailName,
                password: credentailPassword))
let postLoginState = LoginViewState
                        .reducer(initialState, postLoginAction)
XCTAssertEqual(postLoginState.credentials.name,
                                 credentailName)
XCTAssertEqual(postLoginState.credentials.password,
                                 credentailPassword)
XCTAssertTrue(postLoginState.isLoading)

let user = User.sampleData
let didLoginAction = LoginViewStateAction.didLogin(user)
let postDidLoginAction = LoginViewState.reducer(initialState,
 didLoginAction)
XCTAssertFalse(postDidLoginAction.isLoading)
XCTAssertEqual(postDidLoginAction.loggedUser?.firstName,
                user.firstName)
XCTAssertEqual(postDidLoginAction.loggedUser?.lastName,
                user.lastName)
```

Ve zmíněných ukázkách lze vidět odlišnosti testování v obou architekturách. V případě MVVM je testován ViewModel. V architektuře Redux je testován Reducer. Z ukázky lze vidět, že je architektura Redux mnohem lépe testovatelná, jelikož umožňuje testovat každou akci relativně jednoduše. Toto má souvislost i s tím, že v architektuře Redux je přesně dané, jak se aplikace chová.

9.5 Logování událostí

Výhodou architektury Redux je jednoduchá možnost logování událostí napříč aplikací. Toto vyplývá z faktu, že na většinu událostí je nutno mít akci. V následující ukázce je LoggerMiddleware, který stačí zaregistrovat ve Store a následně jsou veškeré akce, které se dějí v aplikaci vypisovány do konzole. Toto je velkou výhodou oproti MVVM architektuře.

```
extension Middlewares {
    static let logger: Middleware<AppState> = { state, action
 in
        print("Action: \(action)")
        print("State: \(state)")

        return Empty().eraseToAnyPublisher()
    }
}
```

9.6 Stavy

Další výhodou v Redux architektuře je funkčnost aplikace dle stavů. Tím nastává možnost, kdy lze aplikaci uvést do konkrétního stavu relativně bez problémů. To může být výhodou, kdy je implementována nová funkcionalita do aplikace, která je závislá na předchozím průběhu aplikace. V případě Redux lze tento stav vyvolat a není potřeba v aplikaci manuálně procházet do konkrétního stavu pomocí uživatelského rozhraní. Další výhodou je možnost zjištění důvodu konkrétní chyby, kdy jsou veškeré akce logovány a následně tak lze zjistit proces, který vedl k chybě.

9.7 Rozšiřitelnost

V MVVM lze bez problémů rozšiřovat konkrétní ViewModel. V architektuře Redux lze zase rozšiřovat akce a reducery. Obě tyto architektury lze bez větších potíží rozšířit a lze říct, že poskytují stejnou možnost rozšiřitelnosti, jelikož obě architektury vhodně oddělují jednotlivé vrstvy.

9.8 Přehlednost

V obou architekturách se View vrstva stará pouze o zobrazení dat. U architektury MVVM se většina logiky nachází ve ViewModelu. Tím je zaručeno, že pokud se vývojář podívá do konkrétního ViewModelu, tak zde najde související logiku. V případě architektury Redux lze sledovat a nahlédnout do Reducer komponent, kde vývojář přímo vidí konkrétní akce a změny. Výhodou přehlednosti architektury Redux může být i to, že obsahuje definované akce pro konkrétní obrazovku. Obě zmíněné architektury jsou přehledné, nicméně architektura Redux má větší přehlednost ale za nutnosti více souborů.

9.9 Podpora platformy

Dalším důležitým aspektem architektury je to, zda zapadá do celkového principu platformy Apple. Architektura MVVM nevyžaduje příliš tvorbu podpůrných tříd na rozdíl od Redux, kde je například potřeba naimplementovat Store. Historicky vývojáři v kombinaci s knihovnou UIKit většinou využívali různé architektury založené na MVC, jelikož lze říct, že tato architektura byla i podporována společností Apple. V případě knihovny SwiftUI neexistuje konkrétní a ověřená architektura, která by byla doporučována, a proto se zde nachází prostor na různé architektury typu Redux, které by mohly do deklarativní syntaxe SwiftUI zapadat. V závěru lze říct, že obě architektury zapadají do knihovny SwiftUI.

9.10 Komunitní podpora

Každá architektura je více nebo méně populární. Výsledkem je podpora a dostupnost materiálů, které slouží ať v počátcích implementace architektury nebo v průběhu. Pokud existuje mnoho materiálů a příkladů k dané architektuře, tak je práce usnadněna, jelikož lze najít příklady, ze kterých se lze inspirovat. Zde pokládám za velkou výhodu architektury MVVM, jelikož k ní lze najít materiály. U architektury Redux v kombinaci se SwiftUI příliš materiálů není a mnoho základních principů, které jsou použity v praktické práci bylo nutné vymyslet, čímž je zvětšena časová náročnost implementace.

9.11 Doporučení

V závěru lze říct, že při výběru architektury je nutno zvážit několik faktorů. Důležitá je velikost projektu a zkušenosti jednotlivých vývojářů. Dalším aspektem jsou i osobní preference. Možností je i kombinace vícero architektur. Ve výsledku lze architekturu MVVM doporučit pro vývoj menších a středně velkých aplikací. Výhodou je také

jednoduchost architektury, kterou si mohou i méně zkušení vývojáři rychle osvojit. Další výhodou je i rychlost implementace. Architekturu Redux lze doporučit pro komplexnější projekt, ve kterém je potřeba sledovat tok dat, jednotlivé stavy a změny v aplikaci. Výhodou architektury je jasná předvídatelnost a konzistence chování. Nicméně tato architektura vyžaduje více zdrojového kódu a delší čas implementace.

ZÁVĚR

Cílem této diplomové práce bylo představení vybraných architektur a návrhových vzorů, které lze využít k vývoji mobilních aplikací pro Apple platformu.

V první kapitole teoretické části této práce byla obecně popsána knihovna SwiftUI. Tato multiplatformní knihovna od společnosti Apple umožňuje deklarativní přístup při tvorbě uživatelského rozhraní. V této kapitole jsou mimo jiné popsány obaly proměnných, které tato knihovna poskytuje.

V druhé kapitole je čtenář uveden do problematiky architektur. Následně jsou popsány specifika jednotlivých architektur, které lze využít primárně k vývoji mobilních aplikací pro operační systém iOS s využitím jazyka Swift. Postupně jsou popsány architektury MVC, MVVM, MVVM-C, MVP, Clean architecture, VIPER a Redux.

V následující kapitole jsou charakterizovány vybrané návrhové vzory, které se využívají v Apple platformě. Mezi popsané vzory patří Singleton, Abstract Factory, Adapter, Command, Target-Action, Key-value observing a Repository pattern.

V poslední kapitole teoretické části jsou popsány možnosti testování kódu v operačním systému iOS. V této kapitole se nachází popis UI testů, Unit testů, performance testů a statické analýzy kódu.

V první kapitole praktické části byly popsány funkcionální a nefunkcionální požadavky na demo aplikaci. Další kapitola se věnuje obecnému popisu demo aplikace, kde jsou popsány jednotlivé obrazovky a jejich význam. Samozřejmostí je popis využitých technologií pro tvorbu demo aplikace.

V následující kapitole je popsána implementace demo aplikace v architektuře MVVM. Tato architektura je často používána pro vývoj mobilních aplikací pro platformu Apple. V této části je podrobně vyobrazena implementace aplikace za použití architektury MVVM. Dále je čtenář podrobně seznámen s modely, které aplikace využívá.

Další kapitola popisuje implementaci aplikace v architektuře Redux. Tato architektura není běžně využívanou v kombinaci s programovacím jazykem Swift a knihovnou SwiftUI. V kapitole jsou popsány veškeré prvky této architektury. Dále je vyobrazena implementace aplikace pomocí této architektury.

V závěru práce se nachází kapitola, která srovnává vytvořené implementace. V úvodu je popsána obecná náročnost implementace a jsou zmíněné problémy, které nastávaly při

implementaci aplikací. Poté jsou implementace srovnány z hlediska počtu položek a řádků. Z tohoto srovnání vyplývá, že architektura Redux potřebuje téměř dvojnásobné množství položek a řádků kódu oproti architektuře MVVM. V další části je popsán ukázkový UI test přihlašovací obrazovky, který není závislý na architektuře a byl tak použit v obou architekturách. Poté jsou vyobrazeny možnosti testování kódu pomocí Unit testů v architektuře MVVM a Redux. Každá architektura umožňuje odlišné možnosti testování pomocí Unit testů. Dále jsou popsány možnosti logování událostí, které lze v architektuře Redux velmi rychle implementovat. Dalším srovnaným bodem je rozšiřitelnost a přehlednost. Následně je popsáno, jak daná architektura zapadá do platformy Apple. V předposledním bodu je porovnána komunitní podpora srovnávaných architektur. V závěru se nachází doporučení pro výběr ze srovnávaných architektur.

SEZNAM POUŽITÉ LITERATURY

[1] JACOBS, Bart. What Is SwiftUI. Cocoacasts [online]. Code Foundry [cit. 2023-05-23]. Dostupné z: https://cocoacasts.com/swiftui-fundamentals-what-is-swiftui

[2] SwiftUI. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/

[3] SwiftUI Overview. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/xcode/swiftui/

[4] Declaring a custom view. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/declaring-a-custom-view

[5] App. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/app

[6] Configuring views. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/configuring-views

[7] ROY, Manisha. SwiftUI: Property wrappers explained in simplest way. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://medium.com/globant/swiftui-property-wrappers-explained-in-simplest-way-28cb580c6408

[8] State. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/state

[9] What is the @State property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-state-property-wrapper

[10] HUDSON, Paul. What is the @StateObject property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-stateobject-property-wrapper

[11] StateObject. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/stateobject

[12] Binding. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/binding

[13] HUDSON, Paul. What is the @Binding property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-binding-property-wrapper

[14] HUDSON, Paul. What is the @Published property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-published-property-wrapper

[15] HUDSON, Paul. What is the @ObservedObject property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-observedobject-property-wrapper

[16] Environment. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swiftui/environment

[17] HUDSON, Paul. What is the @EnvironmentObject property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z:

https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-environmentobject-property-wrapper

[18] HUDSON, Paul. What is the @AppStorage property wrapper?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-appstorage-property-wrapper

[19] Model-View-Controller. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[20] Model-View-Controller. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html

[21] PASQUIER, Benoit. How to implement MVVM pattern in Swift from scratch. Benoit Pasquier [online]. Benoit Pasquier [cit. 2023-05-23]. Dostupné z: https://benoitpasquier.com/ios-swift-mvvm-pattern/

[22] SHARMA, Krupanshu. MVVM in iOS – A Quick Walkthrough. Clarion Technologies [online]. Clarion Technologies [cit. 2023-05-23]. Dostupné z: https://www.clariontech.com/blog/mvvm-in-ios-a-quick-walkthrough

[23] VALDÉS, Daniel Lozano. IOS Architecture: MVVM-C, Coordinators (3/6). Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://medium.com/sudo-by-icalia-labs/ios-architecture-mvvm-c-coordinators-3-6-3960ad9a6d85

[24] OULLADI, Saad El. IOS Swift : MVP Architecture. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3

[25] KUDINOV, Oleh. Clean Architecture and MVVM on iOS. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3

[26] TOZRI, Ghazi. IOS: Clean Architecture using SwiftUI, Combine, and Dependency Injection. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://betterprogramming.pub/ios-clean-architecture-using-swiftui-combine-and-dependency-injection-for-dummies-2e44600f952b

[27] MAHMUDUL ALAM, Sayed. VIPER Design Pattern in Swift for iOS Application Development. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6

[28] KULIK, Wojciech. Redux architecture and mind-blowing features. Wojciech Kulik [online]. Wojciech Kulik [cit. 2023-05-23]. Dostupné z: https://wojciechkulik.pl/ios/redux-architecture-and-mind-blowing-features

[29] ŽOLTÁ, Lucie. Návrhové vzory. Lucie Žoltá [online]. Lucie Žoltá [cit. 2023-05-23]. Dostupné z: http://lucie.zolta.cz/index.php/softwarove-inzenyrstvi/45-navrhove-vzory

[30] HUDSON, Paul. What is a singleton?. Hacking with Swift [online]. Hudson Heavy Industries [cit. 2023-05-23]. Dostupné z: https://www.hackingwithswift.com/example-code/language/what-is-a-singleton

[31] Abstract Factory in Swift. Refactoring.Guru [online]. Refactoring.Guru [cit. 2023-05-23]. Dostupné z: https://refactoring.guru/design-patterns/abstract-factory/swift/example#lang-features

[32] Adapter in Swift. Refactoring.Guru [online]. Refactoring.Guru [cit. 2023-05-23]. Dostupné z: https://refactoring.guru/design-patterns/adapter/swift/example
[33] Cocoa Design Patterns. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW35
[34] Using Key-Value Observing in Swift. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/swift/using-key-value-observing-in-swift#Annotate-a-Property-for-Key-Value-Observing
[35] YATES, Phil. Repository Pattern in Swift. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://blog.devgenius.io/repository-pattern-in-swift-a8eda25b515d
[36] Xcode UI Testing in Swift – Code Examples. CodeWithChris [online]. CodeWithChris [cit. 2023-05-23]. Dostupné z: https://codewithchris.com/xcode-ui-testing-swift/
[37] XCTest. Apple Developer [online]. Apple [cit. 2023-05-23]. Dostupné z: https://developer.apple.com/documentation/xctest
[38] Getting started with Unit Tests in Swift. SwiftLee [online]. SwiftLee [cit. 2023-05-23]. Dostupné z: https://www.avanderlee.com/swift/unit-tests-best-practices/
[39] Unit Testing. Swift by Sundell [online]. Sundell [cit. 2023-05-23]. Dostupné z: https://www.swiftbysundell.com/basics/unit-testing/
[40] JAGTAP, Shashikant. Continuous Performance Testing of an iOS Apps using XCTest. Medium [online]. Medium [cit. 2023-05-23]. Dostupné z: https://medium.com/xcblog/continuous-performance-testing-of-an-ios-apps-using-xctest-811df87fe227
[41] SwiftLint. SwiftLint [online]. JP Simard [cit. 2023-05-23]. Dostupné z: https://realm.github.io/SwiftLint/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
MVC	Model-View-Controler
MVVM	Model-View-ViewModel
MVVM-C	Model-View-ViewModel-Coordinator
MVP	Model-View-Presenter
JSON	JavaScript Object Notation

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH